БИБЛИОТЕКА

ПРОГРАММИСТА



**Дэн Бейдер**



**чистым**

**PYTHON**

**ТОНКОСТИ ПРОГРАММИРОВАНИЯ  
ДЛЯ ПРОФИ**

С^пптт»’

РЗ^Ьоп Tricks: The Book

Dan Bader

**БИБЛИОТЕКА**

ПРОГРАММИСТА



Дэн Бейдер

тонкости ПРОГРАММИРОВАНИЯ  
**ДЛЯ** ПРОФИ

**І^птЕР’** Санкт-Петербург • Москва • Екатеринбург • Воронеж  
^ Нижний Новгород • Ростов-на-Дону • Самара • Минск

**ББК 32.973.2-018.1  
УДК 004.43  
Б41**

**Бейдер Д.**

**Б41 Чистый Python. Тонкости программирования для профи. — СПб.: Питер,  
2018. — 288 с.: ил. — (Серия «Библиотека программиста»).**

**ISBN 978-5-4461-0803-9**

**Изучение всех возможностей Python — сложная задача, а с этой книгой вы сможете сосредо-  
точиться на практических навыках, которые действительно важны. Раскопайте «скрытое золото»  
в стандартной библиотеке Python и начните писать чист^ій код уже сегодня.**

**Если у вас есть опыт работы со старыми версиями Python, вы сможете ускорить работу с со-  
временными шаблонами и функциями, представленными на Python 3.**

**Если вы работали с другими языками программирования и хотите перейти на Python, то найдете  
практические советы, необходимые для того, чтобы стать эффективным питонистом.**

**Если вы хотите научиться писать чистый код, то найдете здесь самые интересные примеры  
и малоизвестные трюки.**

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 Daniel Bader. Все права защищены. Никакая часть данной книги  
не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев ав-  
торских 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 доступ-  
ность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию  
все ссылки на интернет-ресурсы были действующими.**

**ISBN 978-1775093305 англ.  
ISBN 978-5-4461-0803-9**

**© Dan Bader (dbader.org), 2016-2017**

**© Перевод на русский язык ООО Издательство «Питер», 2018  
© Издание на русском языке, оформление ООО Издательство  
«Питер», 2018**

**© Серия «Библиотека программиста», 2018**

**Оглавление**

[**Предисловие 16**](#bookmark3)

[**Комментарии переводчика 18**](#bookmark4)

[**Базовый набор библиотек для разработчика 18**](#bookmark5)

[**От издательства 19**](#bookmark6)

**Глава 1 . Введение 20**

1. [**. Что такое идиома Python 20**](#bookmark9)
2. [**. Чем эта книга будет полезна 22**](#bookmark10)
3. [**.Как читать эту книгу 23**](#bookmark11)
4. **. 4 . Тонкости Python: цифровой комплект инструментов в качестве бонуса . . 24**

[**Глава 2 . Шаблоны для чистого Python 25**](#bookmark13)

1. [**. Прикрой свой з\*\* инструкциями assert 25**](#bookmark14)

[**Инструкция assert в Python — пример 26**](#bookmark15)

[**Почему просто не применить обычное исключение? 27**](#bookmark16)

[**Синтаксис инструкции Python assert 28**](#bookmark17)

**Распространенные ловушки, связанные с использованием инструкции**

**assert в Python 30**

**Предостережение № 1: не используйте инструкции assert для проверки**

**данных 30**

**Предостережение № 2: инструкции assert, которые никогда не дают сбоя 32**

[**Инструкции assert — резюме 34**](#bookmark21)

[**Ключевые выводы 34**](#bookmark22)

1. [**. Беспечное размещение запятой 34**](#bookmark23)

[**Ключевые выводы 38**](#bookmark25)

1. [**. Менеджеры контекста и инструкция with 38**](#bookmark26)

[**Поддержка инструкции with в собственных объектах 40**](#bookmark27)

[**Написание красивых API с менеджерами контекста 42**](#bookmark28)

[**Ключевые выводы 44**](#bookmark29)

1. [**. Подчеркивания, дандеры и другое 44**](#bookmark30)
2. [**. Одинарный начальный символ подчеркивания: \_var 45**](#bookmark31)
3. [**. Одинарный замыкающий символ подчеркивания: var\_ 47**](#bookmark32)
4. [**. Двойной начальный символ подчеркивания: var 48**](#bookmark33)

[**Экскурс: что такое дандеры? 52**](#bookmark34)

1. **. Двойной начальный и замыкающий символ подчеркивания: var . . . 53**
2. [**. Одинарный символ подчеркивания: \_ 54**](#bookmark36)

[**Ключевые выводы 55**](#bookmark37)

1. [**. Шокирующая правда о форматировании строковых значений 56**](#bookmark38)

[**№ 1 . «Классическое» форматирование строковых значений 57**](#bookmark39)

[**№ 2 . «Современное» форматирование строковых значений 58**](#bookmark40)

[**№ 3 . Интерполяция литеральных строк (Python 3.6+) 60**](#bookmark41)

[**№ 4 . Шаблонные строки 62**](#bookmark42)

[**Какой метод форматирования строк мне использовать? 63**](#bookmark43)

[**Ключевые выводы 64**](#bookmark44)

1. [**. Пасхалка «Дзен Python» 64**](#bookmark45)

[**Дзен Python от Тима Питерса 65**](#bookmark46)

**Глава 3 . Эффективные функции 66**

[**3 . 1 . Функции Python — это объекты первого класса 66**](#bookmark48)

[**Функции — это объекты 67**](#bookmark49)

[**Функции могут храниться в структурах данных 68**](#bookmark50)

[**Функции могут передаваться другим функциям 69**](#bookmark51)

[**Функции могут быть вложенными 70**](#bookmark52)

[**Функции могут захватывать локальное состояние 72**](#bookmark53)

[**Объекты могут вести себя как функции 74**](#bookmark54)

[**Ключевые выводы 75**](#bookmark55)

1. [**. Лямбды — это функции одного выражения 75**](#bookmark56)

[**Лямбды в вашем распоряжении 77**](#bookmark57)

[**А может, не надо 78**](#bookmark58)

[**Ключевые выводы 80**](#bookmark59)

1. [**. Сила декораторов 80**](#bookmark60)

[**Основы декораторов Python 82**](#bookmark61)

[**Декораторы могут менять поведение 84**](#bookmark62)

[**Короткая пауза 86**](#bookmark63)

[**Применение многочисленных декораторов к функции 86**](#bookmark64)

[**Декорирование функций, принимающих аргументы 88**](#bookmark65)

[**Ключевые выводы 91**](#bookmark66)

1. [**. Веселье с \*args и \*\*kwargs 92**](#bookmark67)

[**Переадресация необязательных или именованных аргументов 93**](#bookmark68)

[**Ключевые выводы 95**](#bookmark69)

1. [**. Распаковка аргументов функции 96**](#bookmark70)

[**Ключевые выводы 98**](#bookmark71)

1. [**. Здесь нечего возвращать 98**](#bookmark72)

[**Ключевые выводы 101**](#bookmark73)

**Глава 4. Классы и ООП 102**

[**4 .1 . Сравнения объектов: is против == 102**](#bookmark74)

1. [**. Преобразование строк (каждому классу по repr ) 104**](#bookmark75)

[**Метод str против repr 107**](#bookmark76)

[**Почему каждый класс нуждается в repr 110**](#bookmark77)

[**Отличия Python 2 .x: Unicode 112**](#bookmark78)

[**Ключевые выводы 113**](#bookmark80)

1. [**.Определение своих собственных классов-исключений 114**](#bookmark79)

[**Ключевые выводы 117**](#bookmark85)

1. [**. Клонирование объектов для дела и веселья 118**](#bookmark81)

[**Создание мелких копий 119**](#bookmark82)

[**Создание глубоких копий 121**](#bookmark83)

[**Копирование произвольных объектов 122**](#bookmark84)

[**Ключевые выводы 125**](#bookmark87)

1. **. Абстрактные базовые классы держат наследование**

**под контролем 125**

[**Ключевые выводы 128**](#bookmark93)

1. [**. Чем полезны именованные кортежи 129**](#bookmark88)

[**Именованные кортежи спешат на помощь 130**](#bookmark89)

[**Создание производных от Namedtuple подклассов 133**](#bookmark90)

[**Встроенные вспомогательные методы 134**](#bookmark91)

[**Когда использовать именованные кортежи 135**](#bookmark92)

[**Ключевые выводы 135**](#bookmark96)

1. **Переменные класса против переменных экземпляра:**

**подводные камни 136**

[**Пример без собак 139**](#bookmark95)

[**Ключевые выводы 141**](#bookmark104)

**4.8 . Срыв покровов с методов экземпляра, методов класса**

[**и статических методов 142**](#bookmark100)

[**Методы экземпляра 143**](#bookmark98)

[**Методы класса 143**](#bookmark99)

**Статические методы 144**

[**Посмотрим на них в действии! 144**](#bookmark101)

[**Фабрики аппетитной пиццы с @classmethod 147**](#bookmark102)

[**Когда использовать статические методы 149**](#bookmark103)

[**Ключевые выводы 151**](#bookmark113)

[**Глава 5 . Общие структуры данных Python 153**](#bookmark105)

1. [**. Словари, ассоциативные массивы и хеш-таблицы 155**](#bookmark106)

[**diet — ваш дежурный словарь 156**](#bookmark107)

[**collections . OrderedDict — помнят порядок вставки ключей 157**](#bookmark108)

**coNections . defaultdict — возвращает значения, заданные по умолчанию**

**для отсутствующих ключей 158**

**collections . ChainMap — производит поиск в многочисленных словарях**

**как в одной таблице соответствия 159**

**types . MappingProxyType — обертка для создания словарей только**

**для чтения 159**

[**Словари в Python: заключение 160**](#bookmark112)

[**Ключевые выводы 161**](#bookmark121)

1. [**.Массивоподобные структуры данных 161**](#bookmark114)

[**list — изменяемые динамические массивы 162**](#bookmark115)

[**tuple — неизменяемые контейнеры 163**](#bookmark116)

[**array, array — элементарные типизированные массивы 164**](#bookmark117)

[**str — неизменяемые массивы символов Юникода 165**](#bookmark118)

[**bytes — неизменяемые массивы одиночных байтов 167**](#bookmark119)

[**bytearray — изменяемые массивы одиночных байтов 168**](#bookmark120)

**Ключевые выводы 169**

1. [**. Записи, структуры и объекты переноса данных 170**](#bookmark122)

[**dict — простые объекты данных 171**](#bookmark123)

[**tuple — неизменяемые группы объектов 172**](#bookmark124)

**Написание собственного класса — больше работы, больше контроля . . . . 174**

[**collections . namedtuple — удобные объекты данных 175**](#bookmark126)

[**typing . NamedTuple — усовершенствованные именованные кортежи 177**](#bookmark127)

[**struct . Struct — сериализованные С-структуры 178**](#bookmark128)

[**types . SimpleNamespace — причудливый атрибутивный доступ 179**](#bookmark129)

[**Ключевые выводы 180**](#bookmark130)

1. [**. Множества и мультимножества 181**](#bookmark131)

[**set — ваше дежурное множество 182**](#bookmark132)

[**frozenset — неизменяемые множества 183**](#bookmark133)

[**collections . Counter — мультимножества 183**](#bookmark134)

[**Ключевые выводы 184**](#bookmark135)

1. [**. Стеки (с дисциплиной доступа LIFO) 185**](#bookmark136)

[**list — простые встроенные стеки 186**](#bookmark137)

[**collections . deque — быстрые и надежные стеки 187**](#bookmark138)

**deque . LifoQueue — семантика блокирования для параллельных**

**вычислений 188**

[**Сравнение реализаций стека в Python 189**](#bookmark140)

[**Ключевые выводы 190**](#bookmark141)

1. [**. Очереди (с дисциплиной доступа FIFO) 190**](#bookmark142)

[**list — ужасно меееедленная очередь 192**](#bookmark143)

[**collections . deque — быстрые и надежные очереди 193**](#bookmark144)

**queue Queue — семантика блокирования для параллельных**

**вычислений 194**

[**multiprocessing . Queue — очереди совместных заданий 195**](#bookmark146)

[**Ключевые выводы 196**](#bookmark147)

1. [**Очереди с приоритетом 196**](#bookmark148)

[**list — поддержание сортируемой очереди вручную 197**](#bookmark149)

[**heapq — двоичные кучи на основе списка 198**](#bookmark150)

[**queue . PriorityQueue — красивые очереди с приоритетом 199**](#bookmark151)

[**Ключевые выводы 200**](#bookmark152)

**Глава 6 . Циклы и итерации 201**

1. [**. 1 .Написание питоновских циклов 201**](#bookmark153)

[**Ключевые выводы 204**](#bookmark154)

* 1. [**. Осмысление включений 205**](#bookmark155)

[**Ключевые выводы 208**](#bookmark156)

* 1. [**. Нарезки списков и суши-оператор 209**](#bookmark157)

[**Ключевые выводы 211**](#bookmark158)

* 1. [**. Красивые итераторы 212**](#bookmark159)

[**Бесконечное повторение 213**](#bookmark160)

[**Как циклы for-in работают в Python? 215**](#bookmark161)

[**Более простой класс-итератор 218**](#bookmark162)

[**Кто же захочет без конца выполнять итерации 219**](#bookmark163)

[**Совместимость с Python 2 . x 223**](#bookmark164)

[**Ключевые выводы 224**](#bookmark165)

* 1. [**. Генераторы — это упрощенные итераторы 224**](#bookmark166)

[**Бесконечные генераторы 225**](#bookmark167)

[**Генераторы, которые прекращают генерацию 227**](#bookmark168)

[**Ключевые выводы 231**](#bookmark169)

* 1. [**. Выражения-генераторы 231**](#bookmark170)

[**Выражения-генераторы против включений в список 233**](#bookmark171)

[**Фильтрация значений 235**](#bookmark172)

[**Встраиваемые выражения-генераторы 236**](#bookmark173)

[**Слишком много хорошего 236**](#bookmark174)

[**Ключевые выводы 238**](#bookmark175)

1. [**Цепочки итераторов 238**](#bookmark176)

[**Ключевые выводы 241**](#bookmark177)

**Глава 7 . Трюки со словарем 242**

[**7.1 . Значения словаря, принимаемые по умолчанию 242**](#bookmark178)

[**Ключевые выводы 245**](#bookmark179)

[**7. 2 . Сортировка словарей для дела и веселья 245**](#bookmark180)

[**Ключевые выводы 248**](#bookmark181)

[**7. 3 . Имитация инструкций выбора на основе словарей 248**](#bookmark182)

[**Ключевые выводы 253**](#bookmark183)

[**7. 4 . Самое сумасшедшее выражение-словарь на западе 253**](#bookmark184)

[**Ключевые выводы 260**](#bookmark185)

[**7. 5 . Так много способов объединить словари 260**](#bookmark186)

[**Ключевые выводы 263**](#bookmark187)

[**7. 6 . Структурная печать словаря 263**](#bookmark188)

[**Ключевые выводы 265**](#bookmark189)

**Глава 8 . Питоновские методы**

**повышения производительности 266**

1. [**. 1 . Исследование модулей и объектов Python 266**](#bookmark193)

[**Ключевые выводы 269**](#bookmark194)

* 1. [**. Изоляция зависимостей проекта при помощи Virtualenv 270**](#bookmark195)

[**Виртуальные среды спешат на помощь 271**](#bookmark196)

[**Ключевые выводы 274**](#bookmark197)

* 1. [**. По ту сторону байткода 275**](#bookmark198)

[**Ключевые выводы 279**](#bookmark199)

**Глава 9 . Итоги 280**

1. [**. 1 . Бесплатные еженедельные советы для разработчиков на Python 281**](#bookmark202)
   1. [**. PythonistaCafe: сообщество разработчиков на Python 282**](#bookmark203)

**Что питонисты говорят о книге «Чистый Python .  
Тонкости программирования для профи»**

***«Мне эта книга безумно нравится. Она похожа на репетитора, ко-  
торый разъясняет^ ну, типа, всякие трюки или идиомы. Я изучаю  
Python на работе и перешел на него с оболочки командной строки  
Powershell, с которой я познакомился там же,* — *масса нового и фан-  
тастического материала. Всякий раз, когда я попадаю в тупик  
с Python (обычно с шаблонным кодом Flask) или когда чувствую, что  
мой исходный код мог бы выглядеть как-то более по-питоновски,  
я направляю вопросы в нашу внутреннюю дискуссионную группу  
Python.***

***Я часто восхищаюсь некоторыми ответами, которые дают мне  
коллеги. Их отзывы пестрят терминами типа «включения в слова-  
ри», «лямбды» и «генераторы». Я всегда впечатлен и даже поражен  
тем, насколько же Python мощный, когда вы владеете этими при-  
емами и можете их правильно реализовать.***

***Ваша книга стала именно той, которая нужна, чтобы превра-  
титься из запутавшегося скриптера Powershell в того, кто знает,  
как и когда применять эти питоновские «штучки», о которых все  
говорят.***

***Как человеку, у которого нет ученой степени в области Computer  
Science, мне приятно иметь учебное пособие, объясняющее вещи,  
которые другие, возможно, узнали, получая академическое образо-  
вание. Для меня огромное удовольствие читать эту книгу. И кроме  
того, я также подписан на электронную рассылку, что и помогло  
мне выйти на это издание».***

**I** Даниэль Мейер **(Daniel Meyer),  
старший администратор локальных систем в Tesla Inc.**

***«Впервые о вашей книге я услышал от коллеги, который хотел за-  
путать меня примером вашего кода с построением словаря. Я был  
почти на сто процентов уверен, почему словарь в итоге получился  
и более простой, и меньшего размера, но, должен признаться, что  
такого результата я не ожидал.:)***

***Он показал мне книгу по видеосвязи* — *и я как бы просмотрел ее,  
когда он пролистывал мне страницы. Я сразу же загорелся узнать  
больше.***

***Уже к полудню я купил собственный экземпляр книги и продолжил  
читать о том, как в Python создаются словари. Спустя несколько  
часов, когда я встретил другого коллегу за чашкой кофе, то исполь-  
зовал похожий трюк уже с ним.:)***

***Затем коллега поднял вопрос по той же самой теме, и благодаря  
тому, как вы все объяснили в книге, я мог не теряться в догадках,  
а верно ответить, каким будет результат. Это значит, что вы  
прекрасно объяснили материал.:)***

***В Python я не новичок, и некоторые из концепций в отдельных гла-  
вах для меня тоже не новы, но, признаюсь, читая книгу, я все время  
узнаю что-нибудь новое из каждой главы, поэтому респект за на-  
писание очень приличной книги и за фантастическую работу по  
объяснению принципов, лежащих в основе всех тонкостей! С боль-  
шим нетерпением жду обновлений и, безусловно, познакомлю своих  
друзей и коллег с вашей книгой».***

**—** Ог Масьел **(Og Maciel),  
разработчик на Python в Red Hat**

***«с великим удовольствием читал книгу Дэна. Он раскрывает важ-  
ные аспекты Python на ясных примерах (в одном из них используя  
кошек-близнецов, чтобы объяснить операторы* is *и ==).***

***Речь не просто о примерах исходного кода. В издании всесто-  
ронне обсуждаются соответствующие детали реализации. По-  
настоящему важно то, что эта книга реально позволяет вам пи-  
сать программный код на Python лучше!***

***Благодаря этой книге я вдохновился новейшими практическими при-  
емами программирования на Python: например, стал использовать  
собственные исключения и абстрактные классы (когда искал их,  
нашел и блог Дэна). Одни только эти новые знания оправдывают  
цену книги».***

**—** Боб Бельдербос **(Bob Belderbos),  
инженер Oracle и соучредитель PyBites**

**Предисловие**

**Прошло почти десять лет с тех пор, как я впервые познакомилась с языком  
программирования Python. Когда много лет назад я впервые попробовала  
заняться им, то, признаюсь, начала с неохотой. До того я программировала  
на другом языке, и совсем неожиданно на работе меня определили в ту  
команду, где все использовали Python. Это стало началом моего собствен-  
ного путешествия по миру Python.**

**Когда меня впервые познакомили с языком Python, то сказали, что все  
будет легко и я освою его очень быстро. Когда же я спросила коллег о ре-  
сурсах по изучению Python, мне дали всего одну-единственную ссылку  
на официальную документацию. Чтение ее поначалу сбивало с толку,  
и ушло достаточно много времени, прежде чем я научилась уверенно в ней  
ориентироваться, не говоря уже о том, чтобы разбираться. Нередко мне  
приходилось искать решения на веб-сайте** StackOverflow**.**

**Придя из другого языка программирования, я не просто нуждалась в ка-  
ком-нибудь источнике, посвященном обучению программированию или  
дающем пояснения по поводу классов и объектов. Я искала конкретные  
ресурсы, которые научили бы меня функциональным средствам языка  
Python, объяснили разницу между ним и другими языками и то, как написа-  
ние исходного кода на Python отличается от написания его на другом языке.**

**Я потратила немало лет, чтобы полностью осознать ценность этого языка.  
Читая книгу Дэна, я досадовала, что у меня не было ее тогда, когда много  
лет назад я только начала изучать Python.**

**Например, одним из многих уникальных функциональных средств языка  
Python, которое поначалу меня удивило больше всего, была конструкция  
включения в список. Как Дэн отмечает в своей книге, обычной реакцией  
тех, кто только перешел на Python с другого языка, становятся слова «Так  
вот как они используют циклы** for**!». Помню один из первых коммен-  
тариев с обзором исходного кода, который я получила, когда начинала**

**программировать на Python: «Почему бы здесь не применить включение  
в список?» Дэн четко разъясняет это понятие в главе 6, начиная с показа  
организации цикла в чисто питоновском стиле и постепенно достраивая  
его до итераторов и генераторов.**

**В разделе 2.5 Дэн рассматривает различные способы форматирования  
строковых значений в Python. Форматирование строковых значений —  
это одна из тех вещей, которые бросают вызов Дзену языка Python, гла-  
сящему, что должен существовать один и желательно только один очевид-  
ный способ сделать это. Дэн показывает разные способы, в том числе мое  
любимое новое дополнение к языку, f-строки, а также объясняет плюсы  
и минусы каждого метода.**

**Глава «Питоновские методы повышения производительности» представ-  
ляет собой еще один великолепный ресурс. Она охватывает аспекты, ле-  
жащие за пределами языка программирования Python, а также содержит  
советы о том, как отлаживать свои программы, как управлять библиоте-  
ками, от которых они зависят, и дает вам возможность заглянуть внутрь  
байткода Python.**

**Для меня большая честь и удовольствие представить книгу «Чистый  
Python. Тонкости программирования для профи» моего друга Дэна Бейдера.**

**Участвуя в развитии языка Python в качестве разработчика ядра CPython,  
я общаюсь со многими членам сообщества. На своем пути я встретила  
много наставников, помощников и завела много новых друзей. Они напо-  
минают мне о том, что Python — это не только исходный код, но прежде  
всего — сообщество.**

**Чтобы освоить программирование на Python, нужно не только понимать  
теоретические аспекты языка. Для достижения этой цели придется по-  
нять и принять общие правила и самые лучшие практические приемы,  
используемые сообществом.**

**Книга Дэна поможет вам в этом путешествии. Я убеждена, что, прочитав  
ее, вы почувствуете себя увереннее в написании программ на Python.**

**—** Мариатта Виджайя **(Mariatta Wijaya),  
разработчик ядра Python (**mariatta . ca**)**

**Комментарии переводчика**

**Весь материал настоящей книги протестирован в среде Windows 10. При  
тестировании исходного кода за основу взят Python версии 3.6.4 (время  
перевода — апрель 2018 года).**

**Хотя в настоящей книге установка и применение сторонних библиотек  
практически не рассматривается, тем не менее в комментарии переводчи-  
ка включена информация о базовом наборе инструментов, необходимых  
для дальнейшей работы. Эта информация ни к чему не обязывает, но  
служит прекрасной отправной точкой для всех, кто интересуется про-  
граммированием на Python.**

**Базовый набор библиотек для разработчика**

**В обычных условиях библиотеки Python можно скачать и установить из  
каталога библиотек Python** PyPi **(**[**https://pypi**](https://pypi) . python . org/**) с помощью ме-  
неджера пакетов** pip**. Однако следует учесть, что в ОС Windows для ра-  
боты некоторых библиотек, в частности SciPy, Scikit-learn и Scikit-image,  
требуется, чтобы в системе была установлена библиотека Numpy+MKL.  
Библиотека** Numpy+MKL **привязана к библиотеке Intel® Math Kernel  
Library и включает в свой состав необходимые динамические библи-  
отеки (DLL) в каталоге** numpy. core**. Библиотеку Numpy+MKL следует  
скачать из хранилища whl-файлов на веб-странице Кристофа Голька из  
лаборатории динамики флуоресценции Калифорнийского университета  
в г. Ирвайн (**[**http://www**](http://www) . Ifd . uci . edu/~gohlke/pythonlibs/**) и установить при  
помощи менеджера пакетов** pip **как** whl **(соответствующая процедура  
установки пакетов в формате** whl **описана ниже). Например, для 64-раз-  
рядной операционной системы Windows и среды Python 3.6 команда  
будет такой:**

**pip install numpy-1.14.2+mkl-cp36-cp36m-win\_amd64.whl**

**Такой режим установки также касается библиотек** scipy**,** scikit-image  
**и** scikit-learn**. Стоит также отметить, что эти особенности установки не  
относятся к ОС Linux и Mac. Далее приводятся сведения об основопола-  
гающих библиотеках.**

* NumPy **— основополагающая библиотека, необходимая для научных  
  вычислений на Python.**
* SciPy **— библиотека, используемая в математике, естественных науках  
  и инженерном деле. Требует наличия** numpy+mkl**.**
* Matplotlib **— библиотека для работы с двумерными графиками.**
* Pandas **— инструмент для анализа структурных данных и временных  
  рядов. Требует наличия** numpy **и некоторых других. Для чтения файлов  
  Excel требует установки библиотеки** xlrd**.**
* Scikit-learn **— интегратор классических алгоритмов машинного об-  
  учения. Требует наличия** numpy+mkl**.**
* Scikit-image **— коллекция алгоритмов для обработки изображений.  
  Требует наличия** numpy+mkl**.**
* Jupyter **— интерактивная онлайновая вычислительная среда.**
* PyQt5 **— библиотека инструментов для программирования графиче-  
  ского интерфейса пользователя, требуется для работы инструменталь-  
  ной среды программирования** Spyder**.**
* Spyder **— инструментальная среда программирования на Python.**

**От издательства**

**Ваши замечания, предложения, вопросы отправляйте по адресу** comp@  
piter com **(издательство «Питер», компьютерная редакция).**

**Мы будем рады узнать ваше мнение!**

**На веб-сайте издательства** www . piter com **вы найдете подробную инфор-  
мацию о наших книгах.**

**1**

**Введение**

1. **. Что такое идиома Python**

Идиома Python **(Python Trick) — короткий фрагмент исходного кода на  
Python, используемый как инструмент обучения . Идиома Python обуча-  
ет отдельному свойству языка Python путем простой иллюстрации либо  
служит в качестве мотивирующего примера, который дает возможность  
копнуть глубже и развить интуитивное понимание**

**Книга «Чистый Python. Тонкости программирования для профи» на-  
чалась как серия скриншотов с фрагментами кода, которыми я делился  
в Твиттере в течение одной недели. К моему удивлению, они получили  
отклики, а потом еще несколько дней подряд их продолжали распростра-  
нять и ретвитить.**

**Разработчики все чаще и чаще стали обращаться ко мне с вопросом, как  
«получить всю серию». На самом деле у меня на очереди было еще лишь  
несколько таких идиом, которые охватывали целый ряд тем, связанных  
с Python. И за ними не было никакого плана. Они были просто забавным  
экспериментом в Твиттере.**

**Но из этих запросов я понял то, что мои краткие и четкие примеры кода  
стоит рассматривать и как инструмент для обучения. В конце концов  
я занялся созданием еще ряда идиом Python и начал ими делиться в се-  
рии почтовых рассылок. В течение нескольких дней на мою рассылку**

**подписались несколько сотен разработчиков на Python, и я был просто  
в восторге от этого.**

**В следующие дни и недели ко мне нескончаемым потоком стали об-  
ращаться разработчики на Python. Они благодарили за то, что я довел  
до них ту часть языка, которая оставалась для них камнем преткнове-  
ния. Ощущение от этих отзывов было потрясающим. Я-то считал, что  
эти идиомы Python являлись простыми снимками экрана с примерами  
кода, но оказалось, что для многих людей они стали неоценимой по-  
мощью.**

**Именно тогда я решил удвоить ставку на моем эксперименте с идиомами  
Python и довел его до серии из порядка 30 электронных сообщений. Каж-  
дое из них представляло собой заголовок и снимок экрана с примером,  
и вскоре я осознал пределы этого формата. Примерно в ту же пору на мой  
электронной ящик пришло письмо от незрячего разработчика на Python,  
разочарованного тем, что эти идиомы поставлялись как изображения,  
которые он не мог прочитать с помощью экранного диктора.**

**Стала очевидной необходимость уделить этому проекту больше времени,  
чтобы сделать его привлекательнее и доступнее для более широкой ауди-  
тории. Так что я засел за воссоздание всей серии электронных сообщений  
с идиомами Python в текстовом формате и с надлежащей подсветкой  
синтаксиса на основе HTML-разметки. Переиздание моей книги о Python  
было встречено одобрительно. По откликам я понял, что разработчики  
обрадовались тому, что наконец смогли копипастить примеры кода и экс-  
периментировать с ними.**

**По мере того как все больше разработчиков подписывалось на электрон-  
ную рассылку этой серии, я начал замечать закономерности в откликах  
и вопросах, которые я получал. Некоторые идиомы хорошо работали  
именно в качестве мотивационных примеров. Однако что касается более  
сложных из них, то не хватало рассказчика, который направлял бы чита-  
телей или подсказывал им дополнительные ресурсы, где можно было бы  
расширить свое понимание.**

**Скажем так: этот аспект был еще одной большой областью для улучше-  
ния. Основная задача моего веб-сайта** dbader org **состоит в том, чтобы по-  
могать разработчикам на Python становиться еще более потрясающими, —  
и очевидно, что этот аспект предоставлял возможность приблизиться  
к этой цели.**

**Я решил взять из своего почтового курса самые лучшие и самые ценные  
трюки и идиомы и на их основе приступил к написанию книги нового  
типа по Python:**

* **книги, которая обучает самым крутым аспектам языка с помощью ко-  
  ротких и легких для усвоения примеров;**
* **книги, в которой хранятся потрясающие функциональные средства  
  языка Python (класс!) и которая поддерживает мотивацию на высоком  
  уровне;**
* **книги, которая берет вас за руку, ведет по правильному пути и помогает  
  углубить свое понимание языка Python.**

**Для меня эта книга — результат моих любимых занятий и большой экс-  
перимент . Надеюсь, что вы получите удовольствие от ее прочтения и по  
ходу узнаете еще что-то о Python!**

**— Дэн Бейдер**

1. **. Чем эта книга будет полезна**

**Цель этой книги в том, чтобы сделать вас лучшим — более эффективным,  
более осведомленным, более практичным — разработчиком на языке  
Python. Вы, вероятно, задаетесь вопросом: а как чтение этой книги по-  
может мне всего этого достигнуть?**

**«Чистый Python» — это не пошаговое учебное пособие по Python. И это  
не курс языка Python начального уровня. Если вы находитесь на началь-  
ных стадиях изучения этого языка, то в одиночку эта книга не превратит  
вас в профессионального разработчика на Python. Ее чтение, безусловно,  
окажет на вас благотворное влияние, но при этом вам обязательно нужно  
поработать с другими ресурсами, которые сформируют ваши основопо-  
лагающие навыки программирования на Python.**

**Вы извлечете из этой книги максимальную пользу, если в той или иной  
степени владеете языком Python и хотите перейти на следующий уровень.  
Она прекрасно поможет, если вы уже некоторое время программируете  
на Python и готовы пойти дальше, чтобы придать своим познаниям за-  
конченный вид и сделать свой программный код более питоновским.**

**Чтение книги «Чистый Python. Тонкости программирования для профи»  
также окажет бесценную помощь, если у вас уже имеется опыт работы  
с другими языками программирования и вы надеетесь поскорее разобрать-  
ся в тонкостях языка Python. Вы обнаружите массу практических советов  
и шаблонов проектирования, которые сделают вас более эффективным  
и квалифицированным программистом на Python.**

1. **. Как читать эту книгу**

**Оптимальный способ чтения книги «Чистый Python. Тонкости программи-  
рования для профи» — рассматривать ее как копилку потрясающих функ-  
циональных средств языка Python. Каждая приводимая в книге идиома  
Python — самодостаточна, и поэтому ничего страшного, если вы будете  
обращаться к тем из них, которые вызывают у вас наибольший интерес.  
На самом деле именно это я вам и рекомендую делать.**

**Разумеется, вы также можете прочитать всю книгу «Чистый Python.  
Тонкости программирования для профи» от начала до конца. И когда вы  
дойдете до заключительной страницы, вы не пропустите ни одной идиомы  
и шаблона и будете знать, что ознакомились со всем.**

**Некоторые из этих идиом легко понять сразу, и вы не испытаете никаких  
затруднений при их внедрении в повседневную работу, просто прочитав  
главу. Чтобы разобраться в других идиомах, может потребоваться немного  
больше времени.**

**Если вы испытываете затруднения в том, чтобы та или иная идиома за-  
работала в ваших программах, то, как правило, помогает возможность  
поэкспериментировать с каждым примером кода в сеансе интерпретатора  
Python.**

**Если и это не расставит все на свои места, то, пожалуйста, не стесняйтесь  
обращаться ко мне, чтобы я смог выручить вас и дать более подробное  
объяснение. В конечном счете это принесет пользу не только вам, но  
и всем питонистам, которые читают эту книгу.**

**1 . 4 . Тонкости Python: цифровой комплект  
инструментов в качестве бонуса**

**Эта книга сопровождается коллекцией бонусных ресурсов, которые я на-  
зываю «Тонкости Python: цифровой комплект инструментов»^.**

**Среди всего прочего этот комплект инструментов включает 12 видео-  
уроков HD-качества общей продолжительностью более двух часов. Эти  
видеопособия тесно связаны с отдельными главами книги и помогут вам  
быстрее усвоить и закрепить знания, акцентировав внимание на ключевых  
моментах.**

**Включенные в этот комплект инструментов ресурсы стоят 100 $, но при  
покупке этой книги вы получаете их без всякой дополнительной оплаты.**

**Доступ к копии цифрового комплекта инструментов можно получить  
онлайн на моем веб-сайте по адресу dbader org/python-tricks-toolkit .**

**См.** [**https://dbader**](https://dbader)**. org/python-tricks-toolkit**

**Шаблоны для чистого  
Python**

**2 . 1 . Прикрой свой з\*\* инструкциями assert**

**Иногда по-настоящему полезное функциональное средство языка при-  
влекает меньше внимания, чем оно того заслуживает. По некоторым  
причинам это именно то, что произошло со встроенной в Python инструк-  
цией** assert**.**

**В этой главе я собираюсь дать вам представление об использовании**assert **в Python. Вы научитесь ее применять для автоматического об-  
наружения ошибок в программах Python. Эта инструкция сделает ваши  
программы надежнее и проще в отладке.**

**В этом месте вы, вероятно, заинтересуетесь: «Что такое** assert **и в чем ее  
прелесть?» Позвольте дать вам несколько ответов.**

**По своей сути инструкция Python** assert **представляет собой средство  
отладки, которое проверяет условие. Если условие утверждения** assert  
**истинно, то ничего не происходит и ваша программа продолжает выпол-  
няться как обычно. Но если же вычисление условия дает результат ложно,  
то вызывается исключение** AssertionError **с необязательным сообщением  
об ошибке.**

**Инструкция assert в Python — пример**

**Вот простой пример, чтобы дать вам понять, где утверждения** assert  
**могут пригодиться. Я попытался предоставить вам некоторое подобие  
реальной задачи, с которой вы можете столкнуться на практике в одной  
из своих программ.**

**Предположим, вы создаете интернет-магазин с помощью Python. Вы рабо-  
таете над добавлением в систему функциональности скидочного купона,  
и в итоге вы пишете следующую функцию** apply\_discount**:**

**def apply\_discount(product, discount):**

**price = int(product['цена'] \* (1.0 - discount))  
assert 0 <= price <= product['цена']**

**return price**

**Вы заметили, что здесь есть инструкция** assert**? Она будет гарантировать,  
что, независимо от обстоятельств, вычисляемые этой функцией снижен-  
ные цены не могут быть ниже 0 $ и они не могут быть выше первоначаль-  
ной цены товара.**

**Давайте убедимся, что эта функция действительно работает как заду-  
мано, если вызвать ее, применив допустимую скидку. В этом примере  
товары в нашем магазине будут представлены в виде простых слова-  
рей. И скорее всего, в реальном приложении вы примените другую  
структуру данных, но эта безупречна для демонстрации утверждений**assert**. Давайте создадим пример товара — пару симпатичных туфель  
по цене 149,00 $:**

**>>> shoes = {'имя': 'Модные туфли', 'цена': 14900}**

**Кстати, заметили, как я избежал проблем с округлением денежной цены,  
использовав целое число для представления цены в центах? В целом не-  
плохое решение^ Но я отвлекся. Итак, если к этим туфлям мы применим  
25 %-ную скидку, то ожидаемо придем к отпускной цене 111,75 $:**

**>>> apply\_discount(shoes, 0.25)**

**11175**

**Отлично, функция сработала безупречно. Теперь давайте попробуем при-  
менить несколько недопустимых скидок. Например, 200 %-ную «скидку»,  
которая вынудит нас отдать деньги покупателю:**

**>>> apply\_discount(shoes, 2.0)**

**Traceback (most recent call last):**

**File "<input>", line 1, in <module>  
apply\_discount(prod, 2.0)**

**File "<input>", line 4, in apply\_discount  
assert 0 <= price <= product['price']**

***AssertianErrar***

**Как вы видите, когда мы пытаемся применить эту недопустимую скидку,  
наша программа останавливается с исключением** AssertionError**. Это  
происходит потому, что 200 %-ная скидка нарушила условие утверждения**assert**, которое мы поместили в функцию** apply\_discount**.**

**Вы также можете видеть отчет об обратной трассировке этого исключения  
и то, как он указывает на точную строку исходного кода, содержащую вы-  
звавшее сбой утверждение. Если во время проверки интернет-магазина  
вы (или другой разработчик в вашей команде) когда-нибудь столкнетесь  
с одной из таких ошибок, вы легко узнаете, что произошло, просто по-  
смотрев на отчет об обратной трассировке исключения.**

**Это значительно ускорит отладку и в дальнейшем сделает ваши про-  
граммы удобнее в поддержке. А в этом, дружище, как раз и заключается  
сила** assert**!**

**Почему просто не применить обычное исключение?**

**Теперь вы, вероятно, озадачитесь, почему в предыдущем примере я просто  
не применил инструкцию** if **и исключение.**

**Дело в том, что инструкция** assert **предназначена для того, чтобы сооб-  
щать разработчикам о неустранимых ошибках в программе. Инструкция**assert **не предназначена для того, чтобы сигнализировать об ожидаемых  
ошибочных условиях, таких как ошибка «Файл не найден», где пользо-  
ватель может предпринять корректирующие действия или просто попро-  
бовать еще раз.**

**Инструкции призваны быть внутренними самопроверками (internal self-  
checks) вашей программы. Они работают путем объявления неких усло-  
вий, возникновение которых в вашем исходном коде невозможно. Если  
одно из таких условий не сохраняется, то это означает, что в программе  
есть ошибка.**

**Если ваша программа бездефектна, то эти условия никогда не возникнут.  
Но если же они возникают, то программа завершится аварийно с исклю-  
чением** AssertionError**, говорящим, какое именно «невозможное» усло-  
вие было вызвано. Это намного упрощает отслеживание и исправление  
ошибок в ваших программах. А мне нравится все, что делает жизнь легче.  
Надеюсь, вам тоже.**

**А пока имейте в виду, что инструкция** assert **— это средство отладки,  
а не механизм обработки ошибок времени исполнения программы. Цель  
использования инструкции** assert **состоит в том, чтобы позволить разра-  
ботчикам как можно скорее найти вероятную первопричину ошибки. Если  
в вашей программе ошибки нет, то исключение** AssertionError **никогда  
не должно возникнуть.**

**Давайте взглянем поближе на другие вещи, которые мы можем делать  
с инструкцией** assert**, а затем я покажу две распространенные ловушки,  
которые встречаются во время ее использования в реальных сценариях.**

**Синтаксис инструкции Python assert**

**Прежде чем вы начнете применять какое-то функциональное средство  
языка, всегда неплохо подробнее познакомиться с тем, как оно практиче-  
ски реализуется в Python. Поэтому давайте бегло взглянем на синтаксис  
инструкции** assert **в соответствии с документацией Python[[1]](#footnote-1):**

**инструкция\_assert ::= "assert" выражениеі ["," выражение2]**

**В данном случае** выражениеі **— это условие, которое мы проверяем, а не-  
обязательное** выражение2 **— это сообщение об ошибке, которое выводится  
на экран, если утверждение дает сбой. Во время исполнения программы  
интерпретатор Python преобразовывает каждую инструкцию** assert **при-  
мерно в следующую ниже последовательность инструкций:**

**if debug :**

**if not выражениеі:**

**raise AssertionError(выражение2)**

**В этом фрагменте кода есть две интересные детали.**

**Перед тем как данное условие инструкции** assert **будет проверено, про-  
водится дополнительная проверка глобальной переменной** debug **. Это**

**встроенный булев флажок, который при нормальных обстоятельствах  
имеет значение** True**, — и значение** False**, если запрашивается оптимиза-  
ция. Мы поговорим об этом подробнее чуть позже в разделе, посвященном  
«распространенным ловушкам».**

**Кроме того, вы можете применить** выражение2**, чтобы передать необязатель-  
ное сообщение об ошибке, которое будет показано в отчете об обратной  
трассировке вместе с исключением** AssertionError**. Это может еще больше  
упростить отладку. Например, я встречал исходный код такого плана:**

**>>> if cond == 'x':  
do\_x()**

**elif cond == 'y':  
do\_y()  
else:**

**assert False, (**

**'Это никогда не должно произойти, и тем не менее это '  
'временами происходит. Сейчас мы пытаемся выяснить'  
'причину. Если вы столкнетесь с этим на практике, то '  
'просим связаться по электронной почте с dbader. Спасибо!')**

**Разве это не ужасно? Конечно, да. Но этот прием определенно допустим  
и полезен, если в одном из своих приложений вы сталкиваетесь с плава-  
ющей ошибкой Гейзенбаг[[2]](#footnote-2).**

**Распространенные ловушки, связанные с использованием  
инструкции assert в Python**

**Прежде чем вы пойдете дальше, есть два важных предостережения, на  
которые я хочу обратить ваше внимание. Они касаются использования  
инструкций** assert **в Python.**

**Первое из них связано с внесением в приложения ошибок и рисков, свя-  
занных с нарушением безопасности, а второе касается синтаксической  
причуды, которая облегчает написание бесполезных инструкций** assert**.**

**Звучит довольно ужасно (и потенциально таковым и является), поэтому  
вам, вероятно, следует как минимум просмотреть эти два предостереже-  
ния хотя бы бегло.**

**Предостережение № 1: не используйте инструкции assert  
для проверки данных**

**Самое большое предостережение по поводу использования утверждений  
в Python состоит в том, что утверждения могут быть глобально отклю-  
чены[[3]](#footnote-3) переключателями командной строки** -O **и** -OO**, а также переменной  
окружения** PYTHONOPTIMIZE **в СPython.**

**Это превращает любую инструкцию** assert **в нулевую операцию: утверж-  
дения** assert **просто компилируются и вычисляться не будут, это означа-  
ет, что ни одно из условных выражений не будет выполнено[[4]](#footnote-4).**

**Это преднамеренное проектное решение, которое используется схожим  
образом во многих других языках программирования. В качестве побоч-  
ного эффекта оно приводит к тому, что становится чрезвычайно опасно  
использовать инструкции** assert **в виде быстрого и легкого способа про-  
верки входных данных.**

**Поясню: если в вашей программе утверждения** assert **используются для  
проверки того, содержит ли аргумент функции «неправильное» или не-  
ожиданное значение, то это решение может быстро обернуться против вас  
и привести к ошибкам или дырам с точки зрения безопасности.**

**Давайте взглянем на простой пример, который демонстрирует эту про-  
блему. И снова представьте, что вы создаете приложение Python с интер-  
нет-магазином. Где-то среди программного кода вашего приложения есть  
функция, которая удаляет товар по запросу пользователя.**

**Поскольку вы только что узнали об** assert**, вам не терпится применить  
их в своем коде (я бы точно так поступил!), и вы пишете следующую  
реализацию:**

**def delete\_product(prod\_id, user):**

**assert user.is\_admin(), 'здесь должен быть администратор'  
assert store.has\_product(prod\_id), 'Неизвестный товар'  
store. get\_product(prod\_id).delete()**

**Приглядитесь поближе к функции** delete\_product**. Итак, что же произой-  
дет, если инструкции** assert **будут отключены?**

**В этом примере трехстрочной функции есть две серьезные проблемы,  
и они вызваны неправильным использованием инструкций** assert**:**

1. Проверка полномочий администратора инструкциями assert несет  
   в себе опасность. **Если утверждения** assert **отключены в интерпрета-  
   торе Python, то проверка полномочий превращается в нулевую опера-  
   цию. И поэтому теперь любой пользователь может удалять товары.  
   Проверка полномочий вообще не выполняется. В результате повы-  
   шается вероятность того, что может возникнуть проблема, связанная  
   с обеспечением безопасности, и откроется дверь для атак, способных  
   разрушить или серьезно повредить данные в нашем интернет-магазине.  
   Очень плохо. [[5]](#footnote-5)**

**Например, если приложение магазина аварийно завершается при по-  
пытке стороннего лица удалить неизвестный товар, то, скорее всего, это  
произошло потому, что взломщик смог завалить его недопустимыми  
запросами на удаление и вызвать сбой в работе сервера.**

**Каким образом можно избежать этих проблем? Ответ таков: никогда не  
использовать утверждения** assert **для выполнения валидации данных.  
Вместо этого можно выполнять проверку обычными инструкциями** if  
**и при необходимости вызывать исключения валидации данных, как по-  
казано ниже:**

**def delete\_product(product\_id, user):  
if not user.is\_admin():**

**raise AuthError('Для удаления необходимы права админа')  
if not store.has\_product(product\_id):**

**raise VaiueErrar('Идентификатор неизвестного товара')  
store.get\_product(product\_id).delete()**

**Этот обновленный пример также обладает тем преимуществом, что вме-  
сто того, чтобы вызывать неопределенные исключения** AssertionError**,  
он теперь вызывает семантически правильные исключения, а имен-  
но** ValueError **или** AuthError **(которые мы должны были определить  
сами).**

**Предостережение № 2: инструкции assert,  
которые никогда не дают сбоя**

**Удивительно легко случайно написать инструкцию** assert**, которая всегда  
при вычислении возвращает истину. Мне самому в прошлом довелось по-  
нести ощутимый ущерб. Вкратце проблема в следующем.**

**Когда в инструкцию** assert **в качестве первого аргумента передается  
кортеж,** assert **всегда возвращает** True **и по этой причине выполняется  
успешно.**

**Например, это утверждение никогда не будет давать сбой:  
assert(1 == 2, 'Это утверждение должно вызвать сбой')**

**Эта ситуация связана с тем, что в Python непустые кортежи всегда явля-  
ются истинными. Если вы передаете кортеж в инструкцию** assert**, то это  
приводит к тому, что условие** assert **всегда будет истинным, что, в свою  
очередь, приводит к тому, что вышеупомянутая инструкция** assert **ста-  
нет бесполезной, потому что она никогда не сможет дать сбой и вызвать  
исключение.**

**По причине такого, в общем-то, не интуитивного поведения относительно  
легко случайно написать плохие многострочные инструкции** assert**. На-  
пример, в одном из моих комплектов тестов я с легким сердцем написал  
группу преднамеренно нарушенных тестовых случаев, которые внушали  
ложное чувство безопасности. Представьте, что в одном из ваших модуль-  
ных тестов имеется приведенное ниже утверждение:**

**assert (**

**counter == 10,**

**'Это должно было сосчитать все элементы'**

**)**

**На первый взгляд этот тестовый случай выглядит абсолютно приемле-  
мым. Однако он никогда не выловит неправильный результат: это ут-  
верждение** assert **всегда будет давать истину, независимо от состояния  
переменной** counter**. И в чем же тут дело? А в том, что оно подтверждает  
истинность объекта-кортежа.**

**Как я уже сказал, благодаря этому довольно легко выстрелить себе в ногу  
(моя все еще побаливает). Хорошая контрмера, с помощью которой  
можно избежать неприятностей от этой синтаксической причуды, — ис-  
пользовать линтер (linter), инструмент статического анализа кода[[6]](#footnote-6). Кроме  
того, более свежие версии Python 3 для таких сомнительных инструкций**assert **показывают синтаксическое предупреждение.**

**Между прочим, именно поэтому вам также всегда следует выполнять  
быстрый тест «на дым» при помощи своих модульных тестовых случаев.  
Прежде чем переходить к написанию следующего, убедитесь, что они  
действительно не срабатывают.**

**Инструкции assert — резюме**

**Несмотря на данные выше предостережения, я полагаю, что инструкции  
assert являются мощным инструментом отладки, который зачастую не-  
достаточно используется разработчиками Python.**

**Понимание того, как работают инструкции assert и когда их применять,  
поможет писать программы Python, которые будет легче сопровождать  
и отлаживать.**

**Это великолепный навык, который стоит освоить, чтобы прокачать зна-  
ния Python до более качественного уровеня и стать всесторонним пи-  
тонистом. Мне это позволило сэкономить бесконечные часы, которые  
приходилось тратить на отладку.**

**Ключевые выводы**

* **Инструкция Python assert — это средство отладки, которое проверяет  
  условие, выступающее в качестве внутренней самопроверки вашей  
  программы.**
* **Инструкции assert должны применяться только для того, чтобы по-  
  могать разработчикам идентифицировать ошибки. Они не являются  
  механизмом обработки ошибок периода исполнения программы.**
* **Инструкции assert могут быть глобально отключены в настройках  
  интерпретатора.**

1. **. Беспечное размещение запятой**

**Вот вам полезный совет, когда вы добавляете и удаляете элементы из кон-  
станты списка, словаря или множества в Python: завершайте все строки  
запятой.**

**Не поняли, о чем это я? Тогда вот вам примерчик. Предположим, что  
в вашем исходном коде есть вот такой список имен:**

**>>> names = ['Элис', 'Боб', 'Дилберт']**

**Всякий раз, когда вы вносите изменения в этот список, будет трудно ска-  
зать, что именно было изменено, к примеру, в ситуации, когда вы будете  
смотреть на результат команды** Git diff**. Большинство систем управления  
исходным кодом строчно-ориентированы и с трудом справляются с вы-  
делением многочисленных изменений, вносимых в одной-единственной  
строке.**

**Быстрым решением будет принятие стиля оформления кода, при котором  
вы разворачиваете константы списка, словаря или множества на несколь-  
ких строках, как показано ниже:**

**>>> names = [**

**... 'Элис',**

**... 'Боб',**

**... 'Дилберт'**

**... ]**

**Благодаря этому получится один элемент на строку, и во время просмотра  
результатов команды** diff **в своей системе управления исходным кодом  
станет предельно ясно, какой из них был добавлен, удален или изменен.  
Я обнаружил, что это небольшое изменение помогло мне избежать глупых  
ошибок. Оно также расширило возможности моих коллег просматривать  
изменения в исходном коде.**

**Нужно сказать, что два случая редактирования по-прежнему могут вы-  
зывать некоторое недоразумение. Всякий раз, когда вы добавляете новый  
элемент в конец списка или удаляете последний элемент, вам придется  
вручную обновлять размещение запятой для получения единообразного  
форматирования.**

**Допустим, что в этот список вы хотите добавить еще одно имя (Джейн).  
Если вы добавите Джейн, то, чтобы избежать дурацкой ошибки, вам нуж-  
но исправить размещение запятой после строки Дилберт:**

**>>> names = [**

**... 'Элис',**

**... 'Боб',**

**... 'Дилберт' *# <- Пропущенная запятая!***

**... 'Джейн'**

**]**

**После того как вы проинспектируете содержимое этого списка, будьте  
готовы удивиться:**

**>>> names**

**['Элис', 'Боб', 'ДилбертДжейн']**

**Как видите, Python объединил строковые литералы Дилберт и Джейн  
в ДилбертДжейн. Такое поведение, которое называется «конкатенаци-  
ей строковых литералов», является преднамеренным и задокументи-  
рованным. И оно также предоставляет фантастическую возможность  
выстрелить себе в ногу, внося в ваши программы трудноотлавливаемые  
ошибки:**

**Применение многочисленных смежных строковых или байтовых лите-  
ралов (разделенных пробелом), в некоторых случаях с использованием  
разных согласованных правилами оформления кавычек, допустимо, и их  
значение идентично их конкатенации1 .**

**Вместе с тем в некоторых случаях конкатенация строковых литералов яв-  
ляется полезным функциональным средством языка. Например, ее можно  
использовать для сокращения количества обратных слешей (косых),  
необходимых для разбиения длинных строковых констант на несколько  
строк кода:**

**my\_str = ('Это супердлинная строковая константа, '**

**'развернутая на несколько строк. '**

**'И обратите внимание - не требуется никаких обратных косых!')**

**С другой стороны, мы только что увидели, как это же самое функциональ-  
ное средство языка может быстро превратиться в помеху. Итак, каким же  
образом эту ситуацию можно исправить?**

**Добавление пропущенной запятой после Дилберт не дает объединить два  
строковых литерала в один:  
>>> names = [**

**... 'Элис',**

**... 'Боб',**

**... 'Дилберт',**

**... 'Джейн'**

**]**

**Но теперь мы совершили полный круг и вернулись к изначальной про-  
блеме. Мне пришлось изменить две строки кода, чтобы добавить в список  
новое имя. Это снова затрудняет просмотр командой** Git diff **того, что  
было изменено^ Добавил ли кто-то новое имя? Изменил ли кто-то имя  
Дилберта?**

**К счастью, синтаксис языка Python допускает небольшую свободу ма-  
невра, тем самым позволяя решить проблему размещения запятой раз  
и навсегда. Прежде всего, вам просто нужно привыкнуть применять  
стиль оформления кода, который ее избегает. Давайте я покажу, как это  
делается.**

**В Python запятая может размещаться после каждого элемента в константе  
списка, словаря или множества, включая последний элемент. В силу этого  
вам просто нужно не забывать всегда заканчивать строки запятой, и таким  
образом вы избежите жонглирования с размещением запятых, которое  
требовалось бы в противном случае.**

**Вот как будет выглядеть окончательный пример:**

**>>> names = [**

**... 'Элис',**

**... 'Боб',**

**... 'Дилберт',**

**... ]**

**Вы заметили запятую после строкового литерала Дилберт? Этот трюк  
сделает добавление или удаление новых элементов проще и избавит от  
необходимости обновлять размещение запятой. Он унифицирует ваши  
строки, очистит результаты команды** diff **в системе управления исходным  
кодом, а рецензенты вашего кода станут счастливее. Иногда волшебство  
кроется в мелочах, не правда ли?**

**Ключевые выводы**

* **Продуманное форматирование и размещение запятой может упростить  
  обслуживание ваших констант списка, словаря или множества.**
* **Конкатенация строковых литералов как функциональное средство  
  Python может работать как на вас, так и против, внося в код трудноот-  
  лавливаемые ошибки.**

1. **. Менеджеры контекста и инструкция with**

**Некоторые разработчики считают инструкцию Python with малопонят-  
ным функциональным средством языка. Но когда вы заглянете за кулисы,  
то увидите, что никаких танцев с бубнами там нет и она действительно  
является весьма полезным функциональным средством, которое содей-  
ствует написанию более чистого и более удобочитаемого программного  
кода Python.**

**Итак, в чем же прелесть инструкции with? Она помогает упростить не-  
которые распространенные шаблоны управления ресурсами, абстраги-  
руясь от их функциональности и позволяя выделять их и использовать  
повторно.**

**Один из хороших способов увидеть эффективное применение данного  
функционального средства языка — посмотреть на примеры в стандарт-  
ной библиотеке Python. Встроенная функция open() предоставляет пре-  
восходный вариант ее применения:**

**with open('hello.txt', 'w') as f:  
f.write('привет, мир!')**

**Существует общая рекомендация открывать файлы, используя ин-  
струкцию with. Это связано с тем, что она гарантирует автоматическое  
закрытие дескрипторов открытых файлов после того, как выполнение  
программы покидает контекст инструкции with. На внутреннем уровне  
вышеупомянутый пример кода сводится примерно к следующему фраг-  
менту кода:**

**f = open('hello.txt', 'w')  
try:**

**f.write('привет, мир!')  
finally:**

**f.close()**

**Вы сразу можете сказать, что он довольно многословен. Обратите вни-  
мание: инструкция** try^finally **имеет важное значение. Просто написать  
что-то типа этого было бы недостаточно:**

**f = open('hello.txt', 'w')  
f.write('привет, мир!')  
f.close()**

**Если во время вызова** f.write() **случится исключение, то такая реали-  
зация не будет гарантировать, что файл будет закрыт, и поэтому наша  
программа может допустить утечку дескриптора файла. Вот почему ин-  
струкция** with **имеет столь важное значение. Она превращает надлежащее  
получение и высвобождение ресурсов в пустяковую работу.**

**Еще одним хорошим примером, где инструкция** with **эффективно исполь-  
зуется в стандартной библиотеке Python, является класс** threading.Lock**:**

**some\_lock = threading.Lock()**

* ***Вредно:***

**some\_lock.acquire()**

**try:**

* ***Сделать что-то...*finally:**

**some\_lock.release()**

* ***Лучше:***

**with some\_lock:**

* ***Сделать что-то...***

**В обоих случаях использование инструкции** with **позволяет абстрагиро-  
ваться от большей части логики обработки ресурса. Вместо необходимо-  
сти всякий раз писать явную инструкцию** try^finally**, инструкция** with  
**выполняет эту работу за нас.**

**Инструкция** with **может сделать программный код, который работает  
с системными ресурсами, более удобочитаемым. Она также помогает из-**

**бежать ошибок или утечек, делая практически невозможными ситуации,  
когда по разным причинам забывают выполнить очистку или высвобож-  
дение ресурсов после того, как они стали больше ненужными.**

**Поддержка инструкции with в собственных объектах**

**Нужно сказать, что в функции** open() **или классе** threading. Lock **нет  
ничего особенного или чудесного, равно как и в том, что они могут при-  
меняться вместе с инструкцией** with**. Ту же самую функциональность  
можно обеспечить в собственных классах и функциях путем реализации  
так называемых менеджеров контекста (context managers)1.**

**Что такое менеджер контекста? Это простой «протокол» (или интерфейс),  
который ваш объект должен соблюдать для того, чтобы поддерживать  
инструкцию** with**. В сущности, если вы хотите, чтобы объект функциони-  
ровал как менеджер контекста, от вас требуется только одно — добавить**

**в него методы** enter **и** exit **. Python будет вызывать эти два метода**

**в соответствующих случаях в цикле управления ресурсом.**

**Давайте посмотрим, как это выглядит на практике. Вот пример простой  
реализации контекстного менеджера** open()**:**

**class ManagedFile:**

**def init (self, name):**

**self.name = name**

**def enter (self):**

**self.file = open(self.name, 'w')  
return self.file**

**def exit (self, exc\_type, exc\_val, exc\_tb):**

**if self.file:**

**self.file.close()**

**Наш класс** ManagedFile **подчиняется протоколу менеджера контекста и те-  
перь поддерживает инструкцию** with **точно так же, как и первоначальный  
пример с функцией** open()**:  
>>> with ManagedFile('hello.txt') as f:**

**... f.write('привет, мир!')**

**... f.write('a теперь, пока!')**

**Python вызывает** enter **, когда поток исполнения входит в контекст**

**инструкции** with **и наступает момент получения ресурса. Когда поток**

**исполнения снова покидает контекст, Python вызывает** exit **, чтобы**

**высвободить этот ресурс.**

**Написание менеджера контекста на основе класса не является единствен-  
ным способом поддержки инструкции** with **в Python. Служебный модуль**contextlib**1 стандартной библиотеки обеспечивает еще несколько аб-  
стракций, надстроенных поверх базового протокола менеджера контекста.  
Он может слегка облегчить вашу жизнь, если ваши варианты применения  
совпадают с тем, что предлагается модулем** contextlib**.**

**Например, вы можете применить декоратор** contextlib. contextmanager**,  
чтобы определить для ресурса фабричную функцию на основе генератора,  
которая затем будет автоматически поддерживать инструкцию** with**. Вот  
как выглядит пример нашего контекстного менеджера** ManagedFile**, пере-  
писанный в соответствии с этим приемом:**

**from contextlib import contextmanager**

**@contextmanager  
def managed\_file(name):  
try:**

**f = open(name, 'w')  
yield f  
finally:**

**f.close()**

**>>> with managed\_file('hello.txt') as f:**

**... f.write('привет, мир!')**

**... f.write('a теперь, пока!')**

**В данном случае** managed\_fileQ **является генератором, который сначала  
получает ресурс. После этого он временно приостанавливает собственное  
исполнение и передает ресурс инструкцией** yield**, чтобы его использовал  
источник вызова. Когда источник вызова покидает контекст** with**, гене-  
ратор продолжает выполняться до тех пор, пока не произойдут любые  
оставшиеся шаги очистки, после чего ресурс будет высвобожден и воз-  
вращен системе.**

**Реализации на основе класса и на основе генератора по своей сути экви-  
валентны. Вы можете предпочесть тот или иной вариант в зависимости  
от того, какой подход вы считаете более удобочитаемым.**

**Оборотной стороной реализации на основе** @contextmanager **может яв-  
ляться то, что такая реализация требует некоторого вникания в продви-  
нутые понятия языка Python, такие как декораторы и генераторы. Если  
чувствуете, что вам необходимо в них разобраться, то не стесняйтесь по-  
менять маршрут и перейти к соответствующим главам книги.**

**Повторю еще раз: правильный выбор реализации здесь сводится к тому,  
с какой из них вы и ваша команда чувствуете себя комфортно и какую из  
них вы считаете наиболее удобочитаемой.**

**Написание красивых API с менеджерами контекста**

**Менеджеры контекста обладают достаточной гибкостью, и если к при-  
менению инструкции** with **подойти творчески, то для своих модулей  
и классов вы сможете определять удобные API.**

**Например, что, если «ресурсом», которым мы хотели бы управлять, яв-  
ляются уровни отступа текста в некоей программе — генераторе отчетов?  
Что, если бы для этого мы смогли написать исходный код, который вы-  
глядит вот так:**

**with Indenter() as indent:  
indent.print('привет!')  
with indent:**

**indent.print('здорово')  
with indent:**

**indent.print('6oHxyp')**

**indent.print('эй')**

**Он читается почти как предметно-ориентированный язык (DSL) для  
расстановки отступов. Кроме того, обратите внимание, как этот код не-  
сколько раз входит в тот же самый менеджер контекста и покидает, чтобы  
изменить уровни отступа. Выполнение этого фрагмента кода должно при-  
вести к указанному ниже результату и распечатке в консоли аккуратно  
отформатированного текста:**

**привет!**

**здорово**

**бонжур**

**эи**

**Итак, каким образом вы реализовали бы менеджер контекста, который  
поддерживал бы эту функциональность?**

**Между прочим, для вас это может быть прекрасным упражнением, чтобы  
понять, как именно работают менеджеры контекста. Поэтому перед тем,  
как вы обратитесь к моей реализации, приведенной ниже, возьмите паузу  
и попробуйте реализовать это самостоятельно в качестве задания.**

**Если вы готовы взглянуть на мою реализацию, то ниже показано, как  
можно воплотить эту функциональность, использовав менеджер контек-  
ста на основе класса:**

**class Indenter:**

**def init (self):**

**self.level = 0**

**def enter (self):**

**self.level += 1  
return self**

**def exit (self, exc\_type, exc\_val, exc\_tb):**

**self.level -= 1**

**def print(self, text):**

**print(' ' \* self.level + text)**

**Неплохо, правда? Надеюсь, что сейчас вы чувствуете себя увереннее при  
использовании менеджеров контекста Python и инструкции** with **в соб-**ственных программах. Это превосходное функциональное средство языка,  
позволяющее решать задачи по управлению ресурсами в гораздо более  
питоновском и удобном в сопровождении стиле.

**Если вы ищете другое упражнение, чтобы понять тему глубже, то попро-  
буйте реализовать менеджер контекста, измеряющий время исполнения  
блока программного кода с использованием функции time.time. Обяза-  
тельно напишите его в двух вариантах: на основе декоратора и на основе  
класса, чтобы усвоить разницу между ними.**

**Ключевые выводы**

* **Инструкция with упрощает обработку исключений путем инкапсуля-  
  ции стандартных случаев применения инструкций try/finally в так  
  называемые менеджеры контекста.**
* **Чаще всего менеджер контекста используется для управления без-  
  опасным получением и высвобождением системных ресурсов. Ресурсы  
  выделяются при помощи инструкции with и высвобождаются автома-  
  тически, когда поток исполнения покидает контекст with.**
* **Эффективное применение инструкции with помогает избежать утечки  
  ресурсов и облегчает ее восприятие.**

1. **. Подчеркивания, дандеры и другое**

**У символов одинарного и двойного подчеркивания в Python есть осо-  
бый смысл в именах переменных и методов. Отчасти этот смысл суще-  
ствует исключительно по договоренности и предназначен в качестве  
подсказки программисту — и частично он обеспечивается интерпрета-  
тором Python.**

**Если вам интересно, каков смысл символов одинарного и двойного под-  
черкивания в именах переменных и методов, то здесь я приложу все уси-  
лия, чтобы ответить на ваш вопрос. В этом разделе мы обсудим следую-  
щие ниже шаблоны подчеркивания и согласованные правила именования  
и то, как они влияют на поведение ваших программ Python:**

* **Одинарный начальный символ подчеркивания: \_var.**
* **Одинарный замыкающий символ подчркивания: var\_.**
* **Двойной начальный символ подчеркивания: var.**
* **Двойной начальный и замыкающий символ подчеркивания: var .**
* **Одинарный символ подчеркивания: \_.**

1. **. Одинарный начальный символ подчеркивания: \_var**

**В том, что касается имен переменных и методов, префикс, состоящий из  
одинарного символа подчеркивания, имеет свой смысл только по дого-  
воренности и представляет собой подсказку программисту — он означает  
то, что он должен означать по общему согласию сообщества Python, но на  
поведение ваших программ он не влияет.**

**Префикс, состоящий из символа подчеркивания, подразумевается как  
подсказка, которая должна сообщить другому программисту, что пере-  
менная или метод, начинающиеся с одинарного символа подчеркивания,  
предназначаются для внутреннего пользования. Эта договоренность  
определена в PEP 8, руководстве по стилю оформления наиболее широко  
применяемого исходного кода Python[[7]](#footnote-7).**

**Однако эта договоренность не обеспечивается интерпретатором Python.  
В Python отсутствует строгое разграничение между «приватными» и «пу-  
бличными» переменными, как в Java. Добавление одинарного символа  
подчеркивания перед именем переменной больше похоже на размещение  
крошечного подстрочного предупреждающего знака, который говорит:  
«Послушай, эта переменная точно не предназначена быть частью от-  
крытого интерфейса этого класса. Оставь-ка ее в покое».**

**Взгляните на приведенный ниже пример:**

**class Test:**

**def init (self):**

**self.foo = 11  
self. bar = 23**

**Что случится, если создать экземпляр этого класса и попробовать полу-  
чить доступ к атрибутам** foo **и** \_bar**, определенным в его конструкто-  
ре** init **?**

**Давайте узнаем:**

**>>> t = Test() >>> t.foo  
11**

**>>> t.\_bar  
23**

**Как видите, одинарный начальный символ подчеркивания в** \_bar **не поме-  
шал нам «залезть» в класс и получить доступ к значению этой переменной.**

**Все потому, что в Python префикс, состоящий из одинарного подчерки-  
вания, представляет собой просто согласованную договоренность — по  
крайней мере в том, что касается имен переменных и методов. Вместе  
с тем начальные символы подчеркивания влияют на то, как имена импор-  
тируются из модулей. Предположим, что у вас есть модуль** my\_module **и в  
нем есть следующий фрагмент кода:**

***# my\_moduLe.py:***

**def external\_func():  
return 23**

**def \_internal\_func():  
return 42**

**Так вот, если для импорта всех имен из модуля вы будете использовать  
подстановочный импорт (wildcard import) (\*), то Python не будет импор-  
тировать имена с начальным символом подчеркивания (если только в мо-  
дуле не определен список** all **, который отменяет такое поведение[[8]](#footnote-8)):  
>>> from my\_module import \***

**>>> external\_func()**

**23**

**name**

**internal func' is not defined"**

**>>> \_internal\_func() NameError:**

**К слову сказать, подстановочного импорта следует избегать, поскольку он  
вносит неясность в то, какие имена присутствуют в пространстве имен[[9]](#footnote-9).  
Ради ясности лучше придерживаться обычного импорта. Согласованные  
правила именования с одинарным подчеркиванием, в отличие от подста-  
новочного импорта, обычный импорт не затрагивает:**

**>>> import my\_module**

**>>> my\_module.external\_func()**

**23**

**>>> my\_module.\_internal\_func()**

**42**

**Понимаю, этот момент может показаться немножко запутанным. Если  
вы придерживаетесь рекомендаций PEP 8 в том, что подстановочного  
импорта следует избегать, то все, что действительно необходимо запом-  
нить, состоит в следующем:**

**Одинарные символы подчеркивания являются в Python согласованным  
правилом именования, которое говорит о том, что то или иное имя пред-  
назначается для внутреннего использования . Это договорное правило  
обычно интерпретатором Python не обеспечивается и предназначено  
для программиста только в качестве подсказки**

1. **. Одинарный замыкающий символ  
   подчеркивания: var\_**

**Иногда самое подходящее имя переменной уже занято ключевым словом  
языка Python. По этой причине такие имена, как** class **или** def**, в Python  
нельзя использовать в качестве имен переменных. В этом случае можно  
в конец имени добавить символ одинарного подчеркивания, чтобы из-  
бежать конфликта из-за совпадения имен:  
>>> def make\_object(name, class):**

**SyntaxErrar: "invalid syntax"**

**>>> def make\_object(name, class\_):**

**... pass**

**В общих чертах, замыкающий одинарный символ подчеркивания (пост-  
фикс) используется по договоренности, чтобы избежать конфликтов из-  
за совпадения имен с ключевыми словами Python. Эта договоренность  
определена и объяснена в PEP 8.**

1. **. Двойной начальный символ подчеркивания: var**

**Шаблоны именования, которые мы рассмотрели к этому моменту, по-  
лучают свой смысл только из согласованной договоренности. В случае  
атрибутов (переменных и методов) класса Python, которые начинаются  
с двойных символов подчеркивания, все немного по-другому.**

**Префикс, состоящий из двойного символа подчеркивания, заставляет  
интерпретатор Python переписывать имя атрибута для того, чтобы в под-  
классах избежать конфликтов из-за совпадения имен.**

**Такое переписывание также называется искажением имени (name  
mangling) — интерпретатор преобразует имя переменной таким образом,  
что становится сложнее создать конфликты, когда позже класс будет  
расширен.**

**Я знаю, звучит довольно абстрактно. Вот почему я подобрал этот неболь-  
шой пример кода, который мы сможем использовать для эксперименти-  
рования:**

**class Test:**

**def \_\_init\_\_(self):  
self.foo = 11  
self.\_bar = 23  
self. baz = 23**

**Давайте взглянем на атрибуты объекта, использовав встроенную функ-  
цию** dipQ**:  
>>> t = Test()**

**>>> dir(t)**

**['\_Test baz', ' class ' delattr ' diet**

**' dir ', ' doc ', ' eq ', ' format ', ' ge ',**

**' getattribute ', ' gt ', ' hash ', ' init ',**

**' le ', ' lt ', ' module ', ' ne ', ' new ',**

**' reduce ', ' reduce\_ex ', ' repr ',**

**' setattr ', ' sizeof ', ' str ',**

**' subclasshook ', ' weakref ', '\_bar', 'foo']**

**Результат показывает список с атрибутами объекта. Давайте возьмем  
этот список и отыщем наши первоначальные имена переменных foo, \_bar,  
и baz. Обещаю, вы обнаружите несколько интересных изменений.**

**Прежде всего, в списке атрибутов переменная self.foo появляется неиз-  
мененной как foo.**

**Далее, self.\_bar ведет себя таким же образом — она обнаруживается  
в классе как \_bar. Как уже было отмечено, в данном случае начальный  
символ подчеркивания — это просто договоренность, подсказка про-  
граммисту.**

**Однако с атрибутом self. baz все выглядит немного по-другому. Когда**

**вы попытаетесь отыскать в списке атрибут \_\_baz, вы увидите, что пере-  
менной с таким именем там нет.**

**Так что же произошло с** baz**?**

**Если вы приглядитесь, то увидите, что в этом объекте имеется атрибут**

**с именем** \_Test baz**. Это и есть искажение имени, которое применяет**

**интерпретатор Python. Это делается, чтобы защитить переменную от  
переопределения в подклассах.**

**Давайте создадим еще один класс, который расширяет класс** Test **и пы-  
тается переопределить его существующие атрибуты, добавленные в кон-  
структоре:**

**class ExtendedTest(Test):  
def \_\_init\_\_(self):**

**super(). init ()**

**self.foo = 'переопределено'  
self.\_bar = 'переопределено'  
self. baz = 'переопределено'**

**Итак, какими, по вашему мнению, будут значения** foo**,** \_bar **и** baz **в эк-  
земплярах класса** ExtendedTest**? Давайте посмотрим:**

**>>> t2 = ExtendedTest()**

**>>> t2.foo  
*'переопределено'***

**>>> t2.\_bar  
*'переопределено'***

**>>> t2. baz**

***AttributeErrar*:**

***"'ExtendedTest' abject has na attribute '* *baz'"***

**Постойте, почему при попытке проверить значение t2. baz мы получаем**

**исключение AttributeError? Искажение имени наносит очередной удар!  
Оказывается, что этот объект вообще не имеет атрибута baz:**

**>>> dir(t2)**

**['\_ExtendedTest baz', '\_Test baz', ' class ',**

**' delattr ', ' dict ', ' dir ', ' doc ',**

**' eq ', ' format ', ' ge ', ' getattribute ',**

**' gt ', ' hash ', ' init ', ' le ', ' lt ',**

**' module ', ' ne ', ' new ', ' reduce ',**

**' reduce\_ex ', ' repr ', ' setattr ',**

**' sizeof ', ' str ', ' subclasshook ',**

**' weakref ', '\_bar', 'foo', 'get\_vars']**

**Как видите, имя baz превратилось в \_ExtendedTest baz, чтобы предот-  
вратить случайное изменение. Но первоначальное имя \_Test baz по-**

**прежнему на месте:**

**>>> t2.\_ExtendedTest baz**

***'переопределено'***

**>>> t2.\_Test baz**

**42**

**Искажение имени с двойным символом подчеркивания для программи-  
ста совершенно очевидно. Взгляните на следующий пример, который это  
подтверждает:**

**class ManglingTest:**

**def \_\_init\_\_(self):**

**self. mangled = 'Привет'**

**def get\_mangled(self):**

**return self. mangled**

**>>> ManglingTest().get\_mangled()  
'Привет'**

**>>> ManglingTest(). mangled**

***AttributeErrar*:**

***'* *mangied'***

***"'MangiingTest' abject has na attribute***

**Распространяется ли искажение на имена методов? Конечно! Искажение  
имен затрагивает все имена, которые в контексте класса начинаются с двух  
символов подчеркивания (или «дандеров»):  
class MangledMethod:**

**def method(self):**

**return 42**

**def call\_it(self):**

**return self. method()**

**>>> MangledMethod(). method()**

***AttributeErrar*:**

**method'**

**"'MangledMethod' object has no attribute  
>>> MangledMethod().call\_it()**

**42**

**Вот еще один, пожалуй, вызывающий удивление, пример искажения имен  
в действии:**

**\_MangledGlobal mangled = 23**

**class MangledGlobal:  
def test(self):**

**return \_\_mangled**

**>>> MangledGlobal().test()**

**23**

**В этом примере я назначил** \_MangledGlobal mangled **глобальной пе-**

**ременной. Затем к этой переменной я обратился в контексте класса**MangledGlobal**. Из-за искажения имен я смог сослаться на глобальную  
переменную** \_MangledGlobal\_\_mangled **просто как на** \_\_mangled **внутри  
метода** test() **класса.**

**Интерпретатор Python автоматически расширил имя** mangled **до**

\_MangledGlobal mangled**, потому что оно начинается с двух символов**

**подчеркивания. Это показывает, что искажение имен точно не связано  
с атрибутами класса. Оно относится к любому имени, начинающемуся  
с двух символов подчеркивания, которое используется в контексте  
класса.**

**Уф-ф! Многовато, надо переварить.**

**Буду с вами честен: я написал эти примеры и объяснения не сразу из  
головы. Чтобы это сделать, мне потребовалось время на исследование  
и редактирование. Я использую Python много лет, однако правила и вот  
такие особые случаи, как этот, не крутятся у меня в мозгу постоянно.**

**Для программиста иногда самым важным навыком является умение  
«распознавать шаблоны» (образы, паттерны) и понимать, где их нужно  
искать. Если в этом месте вы чувствуете себя несколько подавленными,  
не волнуйтесь. Отдохните и поэкспериментируйте с несколькими при-  
мерами из этого раздела.**

**Пусть эти принципы впитаются как следует, чтобы вы наконец осознали  
общую идею искажения имен и некоторые другие формы поведения, ко-  
торые я вам показал. Если однажды вы столкнетесь с ними «в полях», то  
хотя бы будете знать, что именно искать в документации.**

**Экскурс: что такое дандеры?**

**Если вы слышали разговор опытных питонистов о Python или при-  
сутствовали при обсуждении на конференциях, то, возможно, слышали  
термин дандер (dunder). Вам интересно, что же это такое? Ладно, вот  
ответ.**

**В сообществе Python двойные символы подчеркивания часто называют  
«дандерами» (dunders — это сокращение от англ. double underscores). При-  
чина в том, что в исходном коде Python двойные символы подчеркивания  
встречаются довольно часто, и, чтобы не изнурять свои жевательные  
мышцы, питонисты нередко сокращают термин «двойное подчеркивание»,  
сводя его до «дандера».**

**Например, переменная** baz **будет произноситься как «дандер baz». Ана-  
логичным образом, метод** init **звучит как «дандер init», хотя будет**

**логичным предположить, что так: «дандер init дандер».**

**Но это всего лишь еще одна из причуд среди прочих согласованных пра-  
вил именования. Для разработчиков Python это все равно что секретное  
рукопожатие.**

**4 1-1 W W \j \j**

**. Двойной начальный и замыкающий символ  
подчеркивания: var**

**Пожалуй, это удивляет, но искажение имен не применяется, если имя  
начинается и заканчивается двойными символами подчеркивания. Ин-  
терпретатор Python не трогает переменные, окруженные префиксом  
и постфиксом, которые состоят из двойных символов подчеркивания:**

**class PrefixPostfixTest:**

**def init (self):**

**self. bam = 42**

**>>> PrefixPostfixTest(). bam**

**42**

**Однако имена, у которых есть начальный и замыкающий двойной символ  
подчеркивания, в языке зарезервированы для специального применения.**

**Это правило касается таких имен, как метод** init **для конструкторов**

**объектов или метод** call **, который делает объекты вызываемыми.**

**Эти дандер-методы часто упоминаются как магические методы, однако  
в сообществе Python многим разработчикам, включая меня, это слово не  
нравится. Такое название подразумевает, что применение дандер-методов  
не приветствуется, и это абсолютно не соответствует действительности.  
В Python они представляют собой ключевое функциональное средство  
и должны применяться по мере необходимости. В них нет ничего «маги-  
ческого» или тайного.**

**Тем не менее в контексте согласованных правил именования лучше воз-  
держаться от использования имен, которые начинаются и заканчиваются**

**двойными символами подчеркивания, в своих собственных программах  
во избежание конфликтов с последующими версиями языка Python.**

**5 . Одинарный символ подчеркивания: \_**

**По договоренности одинарный автономный символ подчеркивания ино-  
гда используется в качестве имени, чтобы подчеркнуть, что эта перемен-  
ная временная или незначительная.**

**Например, в приведенном ниже цикле нам не нужен доступ к нарастающе-  
му индексу, и мы можем применить «**\_**», чтобы показать, что этот символ  
подчеркивания является лишь временным значением:**

**>>> for \_ in range(32):**

**... рпіпі('Привет, Мир.')**

**Одинарные символы подчеркивания также можно применять в распа-  
ковке выражений, обозначая таким образом «неважную» переменную,  
чтобы проигнорировать отдельные значения. И снова: смысл одинарного  
подчеркивания существует только по договоренности, и оно не запускает  
особых форм поведения в синтаксическом анализаторе Python. Одинар-  
ный символ подчеркивания — это просто имя допустимой переменной,  
которое иногда используется с этой целью.**

**В следующем ниже примере исходного кода я распаковываю кортеж в от-  
дельные переменные, но я заинтересован только в значениях полей** color  
**и** mileage**. Однако для того, чтобы выражение распаковки было успешным,  
мне нужно назначить переменным все содержащиеся в кортеже значения.  
Именно тут в качестве переменной-заполнителя пригодится символ «**\_**»:**

**>>> car = ('красный', 'легковой автомобиль', 12, 3812.4)**

**>>> color, \_, \_, mileage = car**

**>>> color**

***'красный'***

**>>> mileage**

**3812.4**

**>>> \_**

**12**

**Помимо его применения в качестве временной переменной, символ «\_»  
является специальной переменной в большинстве интерпретаторов Python,  
работающих в цикле чтение-вычисление-печать (REPL). Она представляет  
в них результат последнего выражения, вычисленного интерпретатором.**

**Это удобно, если вы работаете в сеансе интерпретатора и хотите получить  
доступ к результату предыдущего вычисления:**

**>>> 20 + 3  
23**

**>>> \_**

**23**

**>>> print(\_)**

**23**

**Это также удобно, если вы конструируете объекты на лету и хотите взаи-  
модействовать с ними, не назначая им имени перед этим:**

**>>> list()**

**[]**

**>>> \_.append(1)**

**>>> \_.append(2)**

**>>> \_.append(3)**

**>>> \_**

**[1, 2, 3]**

**Ключевые выводы**

* Одинарный начальный символ подчеркивания \_var**: согласованное  
  правило именования, указывающее на то, что имя предназначается  
  для внутреннего использования. Обычно не обеспечивается интерпре-  
  татором Python (за исключением подстановочного импорта) и нужно  
  только как подсказка программистам.**
* Одинарный замыкающий символ подчеркивания var\_**: используется  
  по договоренности, чтобы избежать конфликтов с ключевыми словами  
  Python, которые могут возникнуть из-за совпадения имен.**
* **Двойной начальный символ подчеркивания var: запускает механизм**

**искажения имен при использовании в контексте класса. Обеспечива-  
ется интерпретатором Python.**

* **Двойной начальный и замыкающий символ подчеркивания var :**

**указывает на специальные методы, определенные языком Python.  
Следует избегать этой схемы именования для своих собственных  
атрибутов.**

* Одинарный символ подчеркивания \_**: иногда используется в качестве  
  имени временных или незначительных переменных («неважных»).  
  Кроме того, он представляет результат последнего выражения в сеансе  
  интерпретатора REPL Python.**

1. **. Шокирующая правда о форматировании  
   строковых значений**

**Помните про Дзен Python и про то, как должен существовать «один —  
и, желательно, только один — очевидный способ сделать это»? Вы, воз-  
можно, почешете затылок, когда узнаете, что в Python существует четыре  
основных способа форматирования строковых значений.**

**В этом разделе я покажу, как эти четыре подхода к форматированию строк  
работают и каковы их соответствующие достоинства и недостатки. Я так-  
же покажу вам свое простое «эмпирическое правило» в отношении того,  
как я подбираю наилучший универсальный подход к форматированию  
строк.**

**Сразу перейдем к делу, так как нам многое нужно рассмотреть. Чтобы  
получить простой игрушечный пример для экспериментов, предположим,  
что у нас есть представленные ниже переменные (или на самом деле кон-  
станты), с которыми мы будем работать:**

**>>> errno = 50159747054  
>>> name = 'Боб'**

**И на основе этих переменных мы хотели бы сгенерировать выходное  
строковое значение с сообщением об ошибке:**

***'Эй, Боб! Вот ошибка SxbadcSffee!'***

**Надо сказать, что такая ошибка и впрямь испортит разработчику утро  
понедельника! Но сегодня мы здесь собрались, чтобы обсудить формати-  
рование строк. Поэтому приступим к работе.**

**№ 1 . «Классическое» форматирование  
строковых значений**

**Строковые значения в Python имеют уникальную встроенную операцию,  
к которой можно обратиться через оператор** %**. Этот оператор представ-  
ляет собой краткую форму, которая позволяет очень легко выполнять  
простое позиционное форматирование. Если вы когда-либо имели дело  
с функцией** printf **в языке C, то вы сразу же поймете, как эта операция  
работает. Ниже дан простой пример:**

**>>> 'Привет, %s' % name  
'Привет, Боб'**

**Здесь я использую спецификатор формата** %s**, чтобы сообщить Python, где  
подставить значение переменной** name**, представленной в виде строкового  
значения. Этот способ называется «классическим» форматированием  
строк[[10]](#footnote-10).**

**В классическом форматировании строк существуют и другие специфика-  
торы формата, служащие для того, чтобы дать вам возможность управлять  
выводимым строковым значением. Например, имеется возможность пре-  
образовывать числа в шестнадцатеричную форму записи или заполнять  
пробелами для генерирования безупречно отформатированных таблиц  
и отчетов.**

**Ниже я использую спецификатор формата** %x**, чтобы преобразовать цело-  
численное значение в строковое и представить его как шестнадцатеричное  
число:**

**>>> '%x' % errno  
'badceffee'**

**Синтаксис «классического» форматирования строк слегка изменится,  
если вы захотите выполнить многочисленные подстановки в одном-  
единственном строковом значении. Поскольку оператор** % **принимает  
всего один аргумент, вам необходимо обернуть правую часть в кортеж,  
как здесь:**

**>>> 'Эй, %s! Вот ошибка 0x%x!' % (name, errno)**

***'Эй, Боб! Вот ошибка SxbadcSffee!'***

**Кроме того, к подстановкам переменных в своей форматной строке можно  
обращаться по имени. В этом случае в оператор** % **следует передать словарь  
с отображением имен на соответствующие им значения:**

**>>> 'Эй, %(name)s! Вот ошибка 0x%(ennno)x!' % {**

**... "name": name, "errno": errno }**

***'Эй, Боб! Вот ошибка SxbadcSffee!'***

**Это облегчает поддержку ваших строк и их модификацию в будущем. Вам  
не нужно волноваться о том, что порядок, в котором вы передаете значе-  
ния, совпадает с порядком, на который ссылаются значения в форматной  
строке. Разумеется, оборотной стороной этого приема является то, что он  
требует набирать чуть больше текста.**

**Я уверен, вы спросите, почему такое форматирование в стиле** printf  
**называется «классическим» форматированием строк. Что ж, давайте  
расскажу. Дело в том, что оно технически было заменено на «современ-  
ное» форматирование, о котором мы собираемся поговорить уже через  
минуту. Но несмотря на то что «классическому» форматированию стали  
придавать меньшее значение, оно не было объявлено нерекомендуемым  
для использования. И в последних версиях Python оно по-прежнему под-  
держивается.**

**№ 2 . «Современное» форматирование строковых значений**

**Python 3 ввел новый способ форматирования строк, который позднее  
был также перенесен в Python 2.7. Это «современное» форматирование  
строк избавляется от специального синтаксиса с использованием опера-  
тора % и делает синтаксис форматирования строк более упорядоченным.**

**Форматирование теперь обрабатывается вызовом функции** format () **со  
строковым объектом[[11]](#footnote-11).**

**Функция** format() **может применяться для выполнения простого позици-  
онного форматирования, точно так же, как вы могли поступать в случае  
с «классическим» форматированием:**

**>>> 'Привет, {}'.format(name)**

***'Привет, Боб'***

**Либо вы можете обращаться к подстановкам переменных по имени и ис-  
пользовать их в любом порядке, в котором вы захотите. Это довольно  
мощное функциональное средство языка, поскольку оно позволяет из-  
менять порядок следования отображаемых элементов, не изменяя аргу-  
менты, переданные в функцию форматирования:**

**>>> 'Эй, {name}! Вот ошибка 0x{ennno:x}!'.fonmat(**

**... name=name, errno=errno)**

***'Эй, Боб! Вот ошибка SxbadcSffee!'***

**Этот пример также показывает, как изменился синтаксис форматирова-  
ния целочисленной переменной в виде шестнадцатеричной строки. Теперь  
мы должны передавать спецификацию формата (format spec) путем до-  
бавления суффикса «**:х**» после имени переменной.**

**В целом синтаксис форматной строки стал мощнее, не усложнив при этом  
более простые варианты использования. Время, потраченное на подробное  
изучение документации Python по мини-языку форматирования строк,  
с лихвой окупится[[12]](#footnote-12).**

**В Python 3 «современному» форматированию строк отдается предпо-  
чтение по сравнению с форматированием с использованием** %**. Однако,  
начиная с Python 3.6, появился еще более оптимальный способ форма-  
тирования строковых значений. И об этом способе я вам расскажу в сле-  
дующем разделе.**

**№ 3 . Интерполяция литеральных строк (Python 3.6+)**

**Python 3.6 добавляет еще один способ форматирования строк, который  
называется форматированными строковыми литералами (Formatted  
String Literals). Этот новый способ форматирования строк позволяет ис-  
пользовать выражения Python, которые встраиваются в строковые кон-  
станты. Ниже дан простой пример, который поможет вам проникнуться  
этим функциональным средством языка:**

**>>> ^Привет, {name}!'**

***'Привет, Боб!'***

**В новом синтаксисе форматирования заложена большая мощь. Поскольку  
он позволяет встраивать произвольные выражения Python, вы даже може-  
те выполнять локальные арифметические действия, как показано ниже:**

**>>> a = 5  
>>> b = 10**

**>>> ^Пять плюс десять равняется {а + b}, а не {2 \* (а + b)}.'**

***'Пять плюс десять равняется 15, а не 30.'***

**За кадром форматированные строковые литералы представляют собой  
функциональное средство синтаксического анализатора Python, которое  
преобразовывает** f**-строки в серию строковых констант и выражений.  
Затем они объединяются, формируя окончательное строковое значение.**

**Предположим, что у нас есть следующая функция** greet()**, которая со-  
держит** f**-строку:**

**>>> def greet(name, question):**

**... return f"Привет, {name}! Как {question}?"**

**>>> greet('Боб', 'дела')  
"Привет, Боб! Как дела?"**

**Если эту функцию дизассемблировать и проинспектировать то, что про-  
исходит за кадром, то мы увидим, что в этой функции f-строка преобра-  
зовывается в нечто наподобие следующего:  
>>> def greet(name, question):**

**... return ("Привет, " + name + "! Как " +  
question + "?")**

**На практике реализация слегка быстрее, чем эта, потому что в ней в ка-  
честве оптимизации используется код операции** BUILD\_STRING**1. Но функ-  
ционально они одинаковы:**

**>>> import dis**

**2**

|  |  |  |  |
| --- | --- | --- | --- |
| **dis** | **(greet)** |  |  |
| **0** | **LOAD\_CONST** | **1** | **('Привет, ')** |
| **2** | **LOAD\_FAST** | **0** | **(name)** |
| **4** | **FORMAT\_VALUE** | **0** |  |
| **6** | **LOAD\_CONST** | **2** | **("! Как ")** |
| **8** | **LOAD\_FAST** | **1** | **(question)** |
| **10** | **FORMAT\_VALUE** | **0** |  |
| **12** | **LOAD\_CONST** | **3** | **('?')** |
| **14** | **BUILD\_STRING** | **5** |  |
| **16** | **RETURN\_VALUE** |  |  |

**Строковые литералы также поддерживают существующий синтаксис  
форматных строк метода** str.format()**. Это позволяет решать те же са-  
мые задачи форматирования, которые мы обсудили в предыдущих двух  
разделах:**

**>>> f"Эй, {name}! Вот ошибка {errno:#x}!"**

***"Эй, Боб! Вот ошибка SxbadcSffee!"***

**Новые форматированные строковые литералы Python аналогичны ша-  
блонным литералам JavaScript, добавленным в ES2015. Убежден, что они  
являются довольно приятным дополнением к языку, и я уже начал их  
использовать в своей повседневной работе с Python 3. Подробнее о фор-  
матированных строковых литералах вы можете узнать из официальной  
документации Python[[13]](#footnote-13) [[14]](#footnote-14).**

**№ 4 . Шаблонные строки**

**Еще один прием форматирования строк в Python представлен шаблон-  
ными строками. Этот механизм более простой и менее мощный, но в не-  
которых случаях он может оказаться именно тем, что вы ищете.**

**Давайте взглянем на простой пример приветствия:**

**>>> from string import Template  
>>> t = Template('Эй, $name!')**

**>>> t.substitute(name=name)**

***'Эй, Боб!'***

**Здесь вы видите, что нам приходится импортировать класс** Template **из  
встроенного модуля Python** string**. Шаблонные строки не являются  
ключевым функциональным свойством языка, но они обеспечиваются  
модулем стандартной библиотеки.**

**Еще одно отличие состоит в том, что шаблонные строки не допускают  
спецификаторы формата. Поэтому, чтобы заставить пример со строковой  
ошибкой работать, мы должны сами преобразовать целочисленный код  
ошибки в шестнадцатеричное строковое значение:**

**>>> templ\_string = 'Эй, $name! Вот ошибка $error!'**

**>>> Template(templ\_string).substitute(**

**... name=name, error=hex(errno))**

***'Эй, Боб! Вот ошибка SxbadcSffee!'***

**Пример сработал отлично, но вы, вероятно, интересуетесь, в каких случа-  
ях использовать шаблонные строки в программах на Python. По-моему,  
самый лучший вариант применения шаблонных строк наступает тогда,  
когда вы обрабатываете форматные строки, сгенерированные пользова-  
телями программы. Благодаря их уменьшенной сложности, шаблонные  
строки являются более безопасным вариантом выбора.**

**Более сложные мини-языки форматирования для других приемов фор-  
матирования строк могут вносить уязвимости в ваши программы с точки  
зрения безопасности. Например, форматные строки могут получать до-  
ступ к произвольным переменным в программе.**

**Это означает, что если злонамеренный пользователь может передать  
форматную строку, то он также может потенциально раскрыть секретные  
ключи и другую ценную информацию! Вот простое доказательство идеи  
о том, как такая атака могла бы использоваться:  
>>> SECRET = 'это - секрет'**

**>>> class Error:**

**... def init (self):**

**... pass**

**>>> err = Error()**

**globals\_\_[SECRET]}'**

**>>> user\_input = '{error. init .**

***# Ой-ей-ей...***

**>>> user\_input.format(error=err)**

***'это - секрет'***

**Заметили, как гипотетический взломщик смог извлечь нашу секретную**

**строку, обратившись из форматной строки к словарю** globals **? Жутко,**

**да! Шаблонные строки закрывают это направление атаки, и это делает их  
более безопасным выбором, если вы обрабатываете форматные строки,  
генерируемые из данных, вводимых пользователем:**

**>>> user\_input = '${error. init . globals [SECRET]}'**

**>>> Template(user\_input).substitute(error=err)**

**ValueError:**

***"InvaLid pLacehoLder in string: Line 1, col 1"***

**Какой метод форматирования строк мне использовать?**

**Я вполне понимаю, что, имея такой широкий выбор способов форматиро-  
вания своих строковых значений в Python, вы можете испытывать заме-  
шательство. Здесь не помешало бы соорудить какую-нибудь инфографику  
в виде блок-схемы.**

**Но я этого не сделаю. Вместо этого я попытаюсь все свести к простому  
эмпирическому правилу, которое я применяю, когда пишу на Python.**

**Поехали! Вы можете применять это эмпирическое правило в любой ситу-  
ации, когда вы испытываете затруднения при принятии решения, какой  
метод форматирования использовать; все зависит от обстоятельств.**

**ЭМПИРИЧЕСКОЕ ПРАВИЛО ДЭНА, КАСАЮЩЕЕСЯ  
ФОРМАТИРОВАНИЯ СТРОК PYTHON:**

**Если форматирующие строки поступают от пользователей, то исполь-  
зуйте шаблонные строки, чтобы избежать проблем с безопасностью .  
В противном случае используйте интерполяцию литеральных строк при  
условии, что вы работаете с Python 3.6+, и «современное» форматиро-  
вание строк — если нет .**

**Ключевые выводы**

* **Пожалуй, это удивляет, но в Python существует более одного способа  
  форматирования строк.**
* **У каждого метода есть свои индивидуальные за и против. Ваш вари-  
  ант применения будет влиять на то, какой метод вам следует исполь-  
  зовать.**
* **Если вы затрудняетесь в выборе метода форматирования строк, то  
  попробуйте применить мое эмпирическое правило форматирования  
  строк.**

1. **. Пасхалка «Дзен Python»**

**Я знаю, что далее приводится привычная картина, если говорить о кни-  
гах по Python. И впрямь, нет никаких шансов пройти мимо свода правил  
«Дзен Python» Тима Питерса. За прошедшие годы я не раз извлекал  
пользу из перечитывания этих правил, и думаю, что слова Тима сделали  
из меня более совершенного кодера. Будем надеяться, что они смогут  
сделать то же самое и для вас.**

**Кроме того, можно сказать, что Дзен Python является «большой шишкой»,  
потому что этот свод правил включен в качестве пасхалки в сам язык.  
Просто запустите сеанс интерпретатора Python и выполните следующую  
команду:**

**>>> import this**

**Дзен Python от Тима Питерса**

**Красивое лучше, чем уродливое.**

**Явное лучше, чем неявное.**

**Простое лучше, чем сложное.**

**Сложное лучше, чем запутанное.**

**Плоское лучше, чем вложенное.**

**Разреженное лучше, чем плотное.**

**Читаемость имеет значение.**

**Особые случаи не настолько особые, чтобы нарушать правила.**

**При этом практичность важнее безупречности.**

**Ошибки никогда не должны замалчиваться.**

**Если не замалчиваются явно.**

**Встретив двусмысленность, отбрось искушение угадать.**

**Должен существовать один — и желательно только один — очевидный  
способ сделать это.**

**Хотя он поначалу может быть и не очевиден, если вы не голландец[[15]](#footnote-15).  
Сейчас лучше, чем никогда.**

**Хотя никогда зачастую лучше, чем прямо сейчас.**

**Если реализацию сложно объяснить — идея плоха.**

**Если реализацию легко объяснить — идея, возможно, хороша.  
Пространства имен — отличная вещь! Давайте будем делать их больше!**

**Эффективные функции**

**3**

1. **. Функции Python — это объекты  
   первого класса**

**Функции Python относятся к объектам первого класса. Их можно при-  
сваивать переменным, хранить их в структурах данных, передавать их  
в качестве аргументов другим функциям и даже возвращать их в качестве  
значений из других функций.**

**Глубокое осмысление этих понятий на интуитивном уровне намного упро-  
стит понимание расширенных функциональных средств языка Python,  
в частности лямбда-функций и декораторов. Это также направит вас по  
верному пути к методам функционального программирования.**

**На следующих нескольких страницах я проведу вас через несколько при-  
меров, чтобы помочь вам развить это интуитивное понимание. Примеры  
будут строиться друг поверх друга, поэтому, возможно, вам стоит читать  
их по порядку и даже попробовать некоторые из них в сеансе интерпре-  
татора Python по мере продвижения.**

**Чтобы осмыслить понятия, которые мы будем здесь рассматривать, потре-  
буется немного больше времени, чем вы ожидаете. Не волнуйтесь — это  
абсолютно нормально. Со мной было точно так же. Вполне возможно, что  
вы будете ощущать, как бьетесь головой о стену, а затем в один прекрас-  
ный момент, когда вы будете готовы, вас внезапно осенит, и все встанет  
на свои места.**

**На протяжении всей этой главы я буду в демонстрационных целях ис-  
пользовать приведенную ниже функцию** yell**. Это простой игрушечный  
пример с легко распознаваемым результатом:**

**def yell(text):**

**return text.upper() + '!'**

**>>> уеіі('привет')**

***'ПРИВЕТ!'***

**Функции — это объекты**

**Все данные в программе Python представляются объектами или связями  
между объектами[[16]](#footnote-16). Символьные последовательности (строки), списки,  
модули и функции — все эти явления языка представляют собой объекты.  
Что касается функций, то в Python они ничем не отличаются. Они — тоже  
объекты.**

**Поскольку функция** yell **в Python является объектом, вы можете ее при-  
своить еще одной переменной, точно так же, как это происходит с любым  
другим объектом:**

**>>> bark = yell**

**Эта строка кода не вызывает функцию. Она берет объект-функцию, на  
который ссылается имя** yell**, и создает второе имя,** bark**, которое на него  
указывает. Теперь вы можете исполнить тот же самый объект-функцию,  
который лежит в его основе, вызвав** bark**:**

**>>> Ьагк('гав')**

***'ГАВ!'***

**Объекты-функции и их имена — это две отдельные компетенции. Вот еще  
одно доказательство. Вы можете удалить первоначальное имя функции  
(**yell**), и, поскольку еще одно имя (**bark**) по-прежнему указывает на ле-  
жащую в основе функцию, вы все так же можете через него вызвать эту  
функцию:**

**>>> del yell  
>>> уе11('Привет?')**

**NameError: "name 'yell' is not defined"**

**>>> Ьагк('эй')**

***'ЭЙ!'***

**Кстати, Python прикрепляет к каждой функции строковый идентифи-  
катор. Это делается для отладочных целей во время создания функции.  
К этому внутреннему идентификатору можно получить доступ посред-  
ством атрибута** name ^**:**

**>>> bark. name**

***'yeii'***

**Нужно сказать, что хотя атрибут** name **функции по-прежнему «yell»,**

**это не влияет на то, каким образом вы получаете доступ к объекту-функ-  
ции из вашего программного кода. Идентификатор имени является просто  
средством отладки. Указывающая на функцию переменная и сама функция  
обладают совершенно разными компетенциями.**

**Функции могут храниться в структурах данных**

**Поскольку функции — это объекты первого класса, их можно хранить  
в структурах данных точно так же, как это делается с другими объектами.  
Например, вы можете добавить функции в список:**

**>>> funcs = [bark, str.lower, str.capitalize]**

**>>> funcs**

**[<function yell at 0x10ff96510>,**

**<method 'lower' of 'str' objects>,**

**<method 'capitalize' of 'str' objects>] [[17]](#footnote-17)**

**Доступ к объектам-функциям, хранящимся внутри списка, осуществля-  
ется точно так же, как это происходит с объектом любого другого типа:**

**>>> for f in funcs:**

**... print(f, f('BceM привет'))**

**<function yell at 0x10ff96510> 'ВСЕМ ПРИВЕТ!'**

**<method 'lower' of 'str' objects> 'всем привет'**

**<method 'capitalize' of 'str' objects> 'Всем привет'**

**Хранящийся в списке объект-функцию даже можно вызвать без необходи-  
мости сначала присваивать его переменной. Для этого можно выполнить  
поиск и затем немедленно назвать результирующий «бестелесный» объ-  
ект-функцию внутри одного-единственного выражения:**

**>>> funcs[0]('приветище')**

***'ПРИВЕТИЩЕ!'***

**Функции могут передаваться другим функциям**

**Поскольку функции являются объектами, их можно передавать в качестве  
аргументов другим функциям. Вот функция greet, которая форматирует  
строковое значение приветствия, используя переданный в нее объект-  
функцию, и затем его печатает:**

**def greet(func):**

**greeting = func('Привет! Я - программа Python')  
print(greeting)**

**На результирующее приветствие можно влиять, передавая различные  
функции. Ниже показано, что происходит, если в функцию greet пере-  
дать функцию bark:**

**>>> greet(bark)**

***'ПРИВЕТ! Я - ПРОГРАММА PYTHON'***

**Разумеется, можно также определить новую функцию, чтобы сгенериро-  
вать приветствие с другим колоритом. Например, следующая функция  
whisper может сработать лучше, если вы не хотите, чтобы ваши програм-  
мы Python походили на Оптимуса Прайма из мультсериалов:**def whisper(text):

**return text.lower() +**

**>>> greet(whisper)**

***'Привет! Я - программа Python...'***

**Способность передавать объекты-функции в другие функции в качестве  
аргументов имеет мощные последствия. Она позволяет в своих програм-  
мах абстрагироваться и раздавать поведение. В этом примере функция  
greet остается прежней, но вы можете влиять на ее результат, передавая  
различные линии поведения приветствия.**

**Функции, которые в качестве аргументов могут принимать другие функ-  
ции, также называются функциями более высокого порядка (higher-order  
functions). Они являются непременным условием функционального стиля  
программирования.**

**Классическим примером функций более высокого порядка в Python  
является встроенная функция map. Она принимает объект-функцию  
и итерируемый объект и затем вызывает эту функцию с каждым эле-  
ментом итерируемого объекта, выдавая результат по мере прохождения  
итерируемого объекта.**

**Ниже показано, как вы могли бы отформатировать всю последователь-  
ность приветствий сразу, применив к ним функцию bark:**

**>>> list(map(bark, ['здравствуй', 'эй', 'привет']))**

**['ЗДРАВСТВУЙ!', 'ЭЙ!', 'ПРИВЕТ!']**

**Как видите, функция более высокого порядка map обошла весь список  
и применила функцию bark к каждому его элементу. В результате у нас  
теперь новый объект-список с измененными приветственными строко-  
выми значениями.**

**Функции могут быть вложенными**

**Быть может, вы удивитесь, но Python допускает определение функций  
внутри других функций. Такие функции нередко называются вложен-  
ными функциями (nested functions), или внутренними функциями (inner  
functions). Приведем пример:**def speak(text):

**def whisper(t):**

**return t.lower() +  
return whisper(text)**

**>>> speak('Привет, Мир')**

***'привет, мир...'***

**Итак, что же тут происходит? Всякий раз, когда вы вызываете функцию**speak**, она определяет новую внутреннюю функцию** whisper **и затем  
после этого немедленно ее вызывает. В этом месте мой мозг начинает  
испытывать легкий зуд, но в целом пока материал относительно после-  
довательный.**

**Правда, вот вам неожиданный поворот — функция** whisper **не существует  
за пределами функции** speak**:**

**>>> whisper('Йоу')**

***NameError:***

***"name 'whisper' is not defined"***

**>>> speak.whisper  
AttributeErrar:**

***"'function' object has no attribute 'whisper'"***

**Но что, если вы действительно хотите получить доступ к этой вложенной  
функции** whisper **за пределами функции** speak**? Не забывайте, функции  
являются объектами — и вы можете вернуть внутреннюю функцию ис-  
точнику вызова родительской функции.**

**Например, ниже приведена функция, определяющая две внутренние  
функции. В зависимости от аргумента, передаваемого в функцию верхнего  
уровня, она выбирает и возвращает источнику вызова одну из внутренних  
функций:**

**def get\_speak\_func(volume):  
def whisper(text):**

**return text.lower() + '...'  
def yell(text):**

**return text.upper() + '!'  
if volume > 0.5:**

**return yell  
else:**

**return whisper**

**Обратите внимание на то, как функция** get\_speak\_func **фактически не  
вызывает ни одну из своих внутренних функций — она просто выбирает  
соответствующую внутреннюю функцию на основе аргумента** volume **и за-  
тем возвращает объект-функцию:**

**>>> get\_speak\_func(0.3)**

**<function get\_speak\_func.<locals>.whisper at 0x10ae18>**

**>>> get\_speak\_func(0.7)**

**<function get\_speak\_func.<locals>.yell at 0x1008c8>**

**Разумеется, вы можете продолжить и вызвать возвращенную функцию  
непосредственно, либо сначала присвоив ее переменной:**

**>>> speak\_func = get\_speak\_func(0.7)**

**>>> speak\_func('Привет')**

***'ПРИВЕТ!'***

**Только подумайте^ Это означает, что функции не только могут прини-  
мать линии поведения через аргументы, но и возвращать линии поведения.  
Здорово, правда?**

**И знаете что? С этого места дела приобретают несколько безумный ха-  
рактер. И прежде чем я продолжу писать, мне срочно требуется перерыв  
на кофе (я предлагаю вам сделать то же самое).**

**Функции могут захватывать локальное состояние**

**Вы только что увидели, что функции могут содержать внутренние функ-  
ции и что даже существует возможность возвращать эти (в других ситуа-  
циях скрытые) внутренние функции из родительской функции.**

**Сейчас лучше всего пристегнуть ремень безопасности, потому что все  
становится еще безумнее — мы собираемся зайти на территорию функ-  
ционального программирования еще дальше. (У вас ведь был перерыв на  
кофе, правда?)**

**Мало того что функции могут возвращать другие функции, эти внутрен-  
ние функции также могут захватывать и уносить с собой часть состояния  
родительской функции. И что же это означает?**

**Чтобы это проиллюстрировать, я собираюсь немного переписать преды-  
дущий пример функции get\_speak\_func. Новая версия сразу принимает  
аргументы «volume» и «text», чтобы немедленно сделать возвращаемую  
функцию вызываемой:**

**def get\_speak\_func(text, volume):  
def whisper():**

**return text.lower() + '...'  
def yell():**

**return text.upper() + '!'  
if volume > 0.5:  
return yell  
else:**

**return whisper**

**>>> get\_speak\_func('Привет, Мир', 0.7)()**

***'ПРИВЕТ, МИР!'***

**Теперь взгляните на внутренние функции whisper и yell. Обрати-  
ли внимание на то, что у них больше нет параметра text? Но каким-  
то непостижимым образом они по-прежнему могут получать доступ  
к этому параметру text, определенному в родительской функции. На  
самом деле они, похоже, захватывают и «запоминают» значение этого  
аргумента.**

**Функции, которые это делают, называются лексическими замыканиями  
(lexical closures) (или, для краткости, просто замыканиями). Замыкание  
помнит значения из своего лексического контекста, даже когда поток  
управления программы больше не находится в этом контексте.**

**В практическом плане это означает, что функции могут не только воз-  
вращать линии поведения, но и предварительно конфигурировать эти  
линии поведения. Ниже приведен еще один скелетный пример, который  
иллюстрирует эту идею:**

**def make\_adder(n):  
def add(x):**

**return x + n**

**return add**

**>>> plus\_3 = make\_adder(3)  
>>> plus\_5 = make\_adder(5)  
>>> plus\_3(4)**

**7**

**>>> plus\_5(4)**

**9**

**В данном примере** make\_adder **служит фабрикой для создания и кон-  
фигурирования функций-«сумматоров». Обратите внимание на то, что  
функции-«сумматоры» по-прежнему могут получать доступ к аргументу** n  
**функции** make\_adder **(объемлющему контексту).**

**Объекты могут вести себя как функции**

**Хотя в Python все функции являются объектами, обратное неверно.  
Объекты не являются функциями. Но они могут быть сделаны вызы-  
ваемыми, что во многих случаях позволяет рассматривать их в качестве  
функций.**

**Если объект является вызываемым, то это означает, что вы можете ис-  
пользовать с ним синтаксис вызова функций с круглыми скобками и даже  
передавать в него аргументы вызова функции. Все это приводится в дей-  
ствие дандер-методом** call **. Ниже приведен пример класса, опреде-**

**ляющего вызываемый объект:**

**class Adder:**

**def init (self, n):**

**self.n = n**

**def \_\_call\_\_(self, x):  
return self.n + x**

**>>> plus\_3 = Adder(3)**

**>>> plus\_3(4)**

**7**

**За кадром «вызов» экземпляра объекта в качестве функции сводится  
к исполнению метода** call **этого объекта.**

**Безусловно, не все объекты будут вызываемыми. Вот почему существует  
встроенная функция callable, которая проверяет, является объект вы-  
зываемым или нет:**

**>>> callable(plus\_3)**

**True**

**>>> callable(yell)**

**True**

**>>> саІІаЬІе('привет')**

**False**

**Ключевые выводы**

* **В Python абсолютно все является объектом, включая функции. Их  
  можно присваивать переменным, хранить в структурах данных и пере-  
  давать или возвращать в другие функции и возвращать из них (функ-  
  ции первого класса).**
* **Функции первого класса позволяют абстрагироваться и раздавать  
  линии поведения в ваших программах.**
* **Функции могут быть вложенными, и они могут захватывать и уносить  
  с собой часть состояния родительской функции. Функции, которые это  
  делают, называются замыканиями.**
* **Объекты можно делать вызываемыми. Во многих случаях это позво-  
  ляет рассматривать их в качестве функций.**

1. **. Лямбды — это функции одного выражения**

**Ключевое слово lambda в Python предоставляет краткую форму для  
объявления небольших анонимных функций. Лямбда-функции ведут  
себя точно так же, как обычные функции, объявляемые ключевым сло-  
вом def. Они могут использоваться всякий раз, когда требуются объ-  
екты-функции.**

**Например, ниже показано определение простой лямбда-функции, вы-  
полняющей сложение:**

**>>> add = lambda x, y: x + y  
>>> add(5, 3)**

**8**

**Та же самая функция** add **может быть определена при помощи ключевого  
слова** def**, но она была бы чуть-чуть многословнее:**

**>>> def add(x, y):**

**... return x + y  
>>> add(5, 3)**

**8**

**Сейчас вы, вероятно, задаетесь вопросом: «Что за шум вокруг этих лямбд?  
Если они нечто иное, чем слегка укороченная версия объявления функций  
при помощи ключевого слова** def**, то что тут такого-то?»**

**Взгляните на приведенный ниже пример и держите слова «функциональ-  
ное выражение» в голове, пока его выполняете:**

**>>> (lambda x, y: x + y)(5, 3)**

**8**

**Ладно, и что же здесь произошло? Я просто использовал** lambda**, чтобы  
определить однострочную функцию «**add**», а затем немедленно вызвал ее  
с аргументами** 5 **и** 3**.**

**Концептуально: лямбда-выражение** lambda x, y: x + y **аналогично объяв-  
лению функции при помощи ключевого слова** def**, только записывается  
в одну строку. Основное отличие здесь в том, что перед его использова-  
нием мне не пришлось связывать объект-функцию с именем. Я просто  
сформулировал выражение, которое хотел вычислить как часть лямбды,  
и затем немедленно его вычислил, вызвав лямбда-выражение как обыч-  
ную функцию.**

**Прежде чем вы продолжите, возможно, вам стоит немного поэк-  
спериментировать с приведенным выше примером кода, чтобы по-  
настоящему усвоить его смысл. Я все еще помню, как не сразу до меня  
это наконец дошло. Поэтому не переживайте по поводу того, что по-  
тратите на него несколько минут в сеансе интерпретатора. Это будет  
стоить того.**

**Существует еще одно синтаксическое различие между определениями  
лямбд и обычных функций. Лямбда-функции ограничены одним-един-  
ственным выражением. Это означает, что в лямбда-функциях не могут  
применяться инструкции или аннотации — и даже инструкция** return**.**

**Тогда каким образом возвращать значения из лямбд? При исполнении  
лямбда-функции ее выражение вычисляется и затем результат выраже-  
ния автоматически возвращается, поэтому всегда существует неявная  
инструкция** return**. Именно поэтому некоторые разработчики называют  
лямбды функциями одного выражения.**

**Лямбды в вашем распоряжении**

**В каких случаях в программном коде следует применять лямбда-функ-  
ции? Технически вы можете применять лямбда-выражение в любой  
ситуации, когда от вас ожидается, что вы предоставите объект-функцию.  
И поскольку лямбды могут быть анонимными, перед этим вам даже не  
нужно назначать им имя.**

**Этот факт обеспечивает удобную и «небюрократическую» краткую форму  
для определения функции в Python. Мой самый частый вариант приме-  
нения лямбд состоит в написании кратких и сжатых функций для сорти-  
ровки итерируемых объектов по альтернативному ключу:**

**>>> tuples = [(1, 'd'), (2, 'b'), (4, 'a'), (3, 'c')]**

**>>> sorted(tuples, key=lambda x: x[1])**

**[(4, 'a'), (2, 'b'), (3, 'c'), (1, 'd')]**

**В приведенном выше примере мы сортируем список кортежей по второму  
значению в каждом кортеже. В этом случае лямбда-функция обеспечивает  
быстрый способ изменить порядок сортировки. Ниже приведен еще один  
пример сортировки, с которым вы можете поиграть:**

**>>> sorted(range(-5, 6), key=lambda x: x \* x)**

**[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]**

**Оба показанных мною примера имеют в Python более сжатые реализации  
с использованием встроенных функций** operator.itemgetter() **и** abs()**.**

**Но, надеюсь, вы заметили, как применение лямбды обеспечивает вам  
гораздо большую гибкость. Хотите отсортировать последовательность  
по некоему произвольно вычисленному ключу? Без проблем. Теперь вы  
знаете, как это сделать.**

**Вот еще одна интересная вещь о лямбдах: как и обычные вложенные  
функции, лямбды работают также и как лексические замыкания.**

**Что такое лексическое замыкание? Это лишь затейливое название для  
функции, которая помнит значения из объемлющего лексического контек-  
ста, даже когда поток управления программы больше не находится в этом  
контексте. Вот (довольно академичный) пример, который иллюстрирует  
эту идею:**

**>>> def make\_adder(n):**

**... return lambda x: x + n**

**>>> plus\_3 = make\_adder(3)**

**>>> plus\_5 = make\_adder(5)**

**>>> plus\_3(4)**

**7**

**>>> plus\_5(4)**

**9**

**В приведенном выше примере лямбда** x + n **по-прежнему может получать  
доступ к значению** n**, несмотря на то что она была определена в функции**make\_adder **(объемлющем контексте).**

**Иногда применение лямбда-функции вместо вложенной функции, объ-  
явленной при помощи ключевого слова** def**, может яснее выражать наме-  
рение программиста. Но, честно говоря, это случается редко — по крайней  
мере, не в том программном коде, который мне нравится писать. Поэтому  
давайте поговорим об этом подробнее.**

**А может, не надо\_**

**С одной стороны, я надеюсь, что эта глава заинтересовала вас исследова-  
нием лямбда-функций Python. С другой стороны, я чувствую, что пора  
озвучить еще одно предостережение: лямбда-функции следует применять  
экономно и с необычайной осторожностью.**

**Уж я-то знаю. Я написал изрядную долю исходного кода с использова-  
нием лямбд, которые выглядели «круто», но на самом деле оказались по-  
мехой для меня и моих коллег. Если вы испытываете желание применить  
лямбда-функцию, потратьте несколько секунд (или минут) и обдумайте,  
будет ли этот способ достижения нужного результата и вправду самым  
чистым и максимально удобным в сопровождении.**

**Например, делать что-то подобное для экономии пары строк кода просто  
глупо. Несомненно, технически все работает, и это вполне приличный  
«трюк». Но он порядочно смутит очередную девчонку или парня, которые  
в жесткие сроки должны отправить багфикс:**

* ***Вредно:***

**>>> class Car:**

**... rev = lambda self: ргіп1('Бум!')**

**... crash = lambda self: ргіп1('Бац!')**

**>>> my\_car = Car()**

**>>> my\_car.crash()**

***'Бац!'***

**Похожие чувства у меня и по поводу запутанных конструкций map()  
и filter() с использованием лямбд. Обычно гораздо лучше использовать  
конструкцию включения в список или в выражение-генератор:**

* ***Вредно:***

**>>> list(filter(lambda x: x % 2 == 0, range(16)))**

**[0, 2, 4, 6, 8, 10, 12, 14]**

* ***Лучше:***

**>>> [x for x in range(16) if x % 2 == 0]**

**[0, 2, 4, 6, 8, 10, 12, 14]**

**Если вы заметили, что с лямбда-выражениями получаете что-то хотя  
бы отдаленно сложное, то лучше обратитесь к определению автономной  
функции с подходящим именем.**

**В конечном счете экономия нескольких нажатий клавиш не будет иметь  
значения, зато ваши коллеги (и вы сами в будущем) оценят чистый и удо-  
бочитаемый код больше, чем лаконичное колдовство.**

**Ключевые выводы**

* **Лямбда-функции — это функции одного-единственного выражения,  
  которые не обязательно привязаны к имени (анонимны).**
* **В лямбда-функциях нельзя использовать обычные инструкции Python,  
  и в них всегда содержится неявная инструкция возврата** return**.**
* **Всегда спрашивайте себя: обеспечит ли применение обычной (именован-  
  ной) функции либо конструкции включения в список большую ясность?**

**з. 3 . Сила декораторов**

**В своей основе декораторы Python позволяют расширять и изменять  
поведение вызываемых объектов (функций, методов и классов) без не-  
обратимой модификации самих вызываемых объектов.**

**Любая достаточно универсальная функциональность, которую можно  
прикрепить к существующему классу или поведению функции, является  
отличным кандидатом для декорирования. Сюда входят:**

* **ведение протокола операций (журналирование);**
* **обеспечение контроля за доступом и аутентификацией;**
* **функции инструментального оформления и хронометража;**
* **ограничение частоты вызова API;**
* **кэширование и др.**

**Итак, зачем осваивать использование декораторов в Python? В конце кон-  
цов, только что упомянутое мною выше выглядит довольно абстрактным,**

**и, вполне возможно, вообще сложно увидеть, какую пользу принесут де-  
кораторы в повседневной работе разработчика. Позвольте мне несколько  
прояснить этот вопрос, предоставив вам вполне реальный пример.**

**Предположим, в вашей программе составления отчетности есть 30 функ-  
ций с бизнес-логикой. Одним дождливым утром понедельника ваш босс  
подходит к вашему столу и заявляет: «Доброго понедельника! Помните  
ту отчетность по TPS? Мне нужно, чтобы вы в каждый шаг генератора  
отчетов добавили ведение протокола входных и выходных операций. Ком-  
пании XYZ это нужно для аудиторских целей. Да, и еще. Я им сказал, что  
к среде мы сможем все отправить».**

**В зависимости от того, насколько вы разбираетесь в декораторах, от этого  
поручения либо у вас подскочит давление, либо вы останетесь в относи-  
тельном спокойствии.**

**Без декораторов следующие три дня вам пришлось бы провести в попыт-  
ках модифицировать каждую из этих 30 функций, приводя их в полный  
беспорядок ручными вызовами операции журналирования. Чудесно, не  
правда ли?**

**А если вы знаете свои декораторы, вы спокойно улыбнетесь своему боссу  
и скажете: «Не беспокойся, Джим. Я сделаю это сегодня к 14:00».**

**Сразу после этого вы наберете исходный код для универсального декора-  
тора @audit\_log (всего порядка 10 строк кода) и быстро вставите его перед  
каждым определением функции. Затем вы зафиксируете код в GitHub  
и перехватите очередную чашечку кофе\_**

**Здесь я драматизирую, но лишь капельку. Декораторы действительно  
бывают такими мощными. Я даже сказал бы, что для любого серьезного  
программиста на Python понимание декораторов является важной вехой.  
Они требуют твердого усвоения нескольких продвинутых концепций  
языка, включая свойства функций первого класса.**

**Уверен, что награда за понимание того, как в Python работают декора-  
торы, может быть огромной.**

**Несомненно, декораторы довольно сложны, чтобы твердо усвоить их  
с первого раза, но они — очень полезное функциональное средство языка,  
с которым вы будете часто сталкиваться в сторонних платформах и в стан-  
дартной библиотеке Python. В любом хорошем обучающем пособии по  
Python задача объяснения декораторов относится к разряду пан или пропал.  
Здесь я приложу все усилия, чтобы шаг за шагом познакомить вас с ними.**

**Правда, прежде чем вы погрузитесь в эту тему, сейчас самый подходящий  
момент, чтобы освежить память относительно свойств функций первого  
класса в Python. В этой книге им посвящен целый раздел, и я рекомен-  
дую вам потратить несколько минут, чтобы его просмотреть. Самыми  
важными для понимания декораторов выводами относительно «функций  
первого класса» являются следующие:**

* **функции являются объектами — их можно присваивать переменным,  
  передавать в другие функции и возвращать из других функций;**
* **функции могут быть определены внутри других функций — и дочер-  
  няя функция может захватывать локальное состояние родительской  
  функции (лексические замыкания).**

**Готовы? Отлично! Тогда приступим.**

**Основы декораторов Python**

**Итак, что же такое декораторы на самом деле? Они «украшают», или  
«обертывают», другую функцию и позволяют исполнять программный  
код до и после того, как обернутая функция выполнится.**

**Декораторы позволяют определять конструктивные блоки многократного  
использования, которые могут изменять или расширять поведение других  
функций. И они позволяют это делать без необратимых изменений самой  
обернутой функции. Поведение функции изменяется, только когда оно  
декорировано.**

**Как могла бы выглядеть реализация простого декоратора? В общих чертах  
декоратор — это вызываемый объект, который на входе принимает один  
вызываемый объект, а на выходе возвращает другой вызываемый объект.**

**Приведенная ниже функция имеет это свойство и может считаться самым  
простым декоратором, который вы могли когда-либо написать:**

**def null\_decorator(func):  
return func**

**Как вы видите, null\_decorator является вызываемым объектом (это  
функция). На входе он принимает еще один вызываемый объект и на  
выходе возвращает тот же самый вызываемый объект без его изменения.**

**Давайте его применим, чтобы декорировать (или обернуть) еще одну  
функцию:**

**def greet():**

**return 'Привет!'**

**greet = null\_decorator(greet)**

**>>> greet()**

***'Привет!'***

**В этом примере я определил функцию** greet **и сразу же ее декорировал,  
пропустив через функцию** null\_decorator**. Понимаю, пока это все вы-  
глядит бесполезным. Я ведь о том, что мы намеренно спроектировали  
пустой декоратор бесполезным, верно? Но через мгновение этот пример  
разъяснит, как работает специальный синтаксис Python, предназначенный  
для декораторов.**

**Вместо того чтобы явным образом вызывать** null\_decorator **с функцией**greet **и затем по-новому присваивать его переменной, удобнее восполь-  
зоваться синтаксисом Python** @ **для декорирования функции:**

**@null\_decorator  
def greet():**

**return 'Привет!'**

**>>> greet()**

***'Привет!'***

**Размещение строки** @null\_decorator **перед определением функции анало-  
гично тому, что функция сначала определяется и затем уже прогоняется  
через декоратор. Синтаксис** @ **является всего лишь синтаксическим саха-  
ром (syntactic sugar) и краткой формой для этого широко применяемого  
шаблона.**

**Обратите внимание: синтаксис** @ **декорирует функцию непосредственно  
во время ее определения. При этом становится трудно получить доступ  
к недекорированному оригиналу без хрупких хакерских фокусов. По этой  
причине вы можете решить вручную декорировать некоторые функции  
для сохранения способности вызвать и недекорированную функцию.**

**Декораторы могут менять поведение**

**Теперь, когда вы чуть ближе познакомились с синтаксисом декораторов,  
давайте напишем еще один декоратор, который действительно что-то  
делает и изменяет поведение декорированной функции.**

**Вот чуть более сложный декоратор, который преобразовывает результат  
декорированной функции в буквы верхнего регистра:**

**def uppercase(func):  
def wrapper():**

**original\_result = func()  
modified\_result = original\_result.upper()  
return modified\_result  
return wrapper**

**Вместо того чтобы просто возвратить входную функцию, как это делал  
пустой декоратор, декоратор** uppercase **на лету определяет новую функ-  
цию (замыкание) и использует ее в качестве обертки входной функции,  
чтобы изменить ее поведение во время вызова.**

**Замыкание** wrapper **имеет доступ к недекорированной входной функции,  
и оно свободно может выполнить дополнительный программный код до  
и после ее вызова. (Технически замыканию вообще не нужно вызывать  
входную функцию.)**

**Заметьте, что вплоть до настоящего момента декорированная функция  
ни разу не была исполнена. На самом деле, в вызове входной функции на  
данном этапе нет никакого смысла — потребность в том, чтобы декоратор  
был в состоянии изменить поведение своей входной функции, возникнет,  
только когда он наконец будет вызван.**

**Возможно, вам нужен минутный перерыв, чтобы переварить услышанное.  
Представляю, каким сложным для вас может казаться этот материал, но  
мы в нем разберемся вместе. Обещаю.**

**Самое время, чтобы взглянуть на декоратор** uppercase **в действии. Что  
произойдет, если продекорировать им оригинальную функцию** greet**?**

**@uppercase  
def greet():**

**return 'Привет!  
>>> greet()**

***'ПРИВЕТ!'***

**Надеюсь, вы ждали именно этот результат. Давайте взглянем поближе  
на то, что здесь только что произошло. В отличие от** null\_decorator**,  
декоратор** uppercase **при декорировании функции возвращает другой  
объект-функцию:**

**>>> greet**

**<function greet at 0x10e9f0950>**

**>>> null\_decorator(greet)**

**<function greet at 0x10e9f0950>**

**>>> uppercase(greet)**

**<function uppercase.<locals>.wrapper at 0x76da02f28>**

**И как вы видели чуть раньше, ему это нужно, чтобы изменить поведение  
декорированной функции, когда он в итоге будет вызван. Декоратор**uppercase **сам является функцией. И единственный способ повлиять на  
«будущее поведение» входной функции, которую он декорирует, состоит  
в том, чтобы подменить (или обернуть) входную функцию замыканием.**

**Вот почему декоратор** uppercase **определяет и возвращает еще одну функ-  
цию (замыкание), которая затем может быть вызвана в дальнейшем, вы-  
полняет оригинальную входную функцию и модифицирует ее результат.**

**Декораторы изменяют поведение вызываемого объекта посредством  
обертки-замыкания, в результате чего вам не приходится необратимо  
модифицировать оригинал. Оригинальный вызываемый объект не из-  
меняется необратимо — его поведение меняется, только когда он деко-  
рирован.**

**Это позволяет прикреплять к существующим функциям и классам кон-  
структивные блоки многократного использования, в частности функ-  
ционал журналирования и другое инструментальное оформление. Этот  
факт делает декораторы настолько мощным функциональным средством  
языка, что они нередко используются в стандартной библиотеке Python  
и в сторонних пакетах.**

**Короткая пауза**

**Кстати, если в этом месте вы почувствовали, что вам нужен короткий  
перерыв на чашечку кофе или на прогулку вокруг квартала, то это абсо-  
лютно нормально. По моему мнению, в Python замыкания и декораторы  
являются одними из самых трудных для понимания концепций.**

**Пожалуйста, не торопитесь и не переживайте по поводу того, что не полу-  
чается разобраться во всем этом сразу же. Переварить услышанное часто  
помогает работа с примерами. Поэкспериментируйте с ними по очереди  
в сеансе интерпретатора.**

**Уверен, что у вас получится!**

**Применение многочисленных декораторов к функции**

**Возможно, вас не удивит, что к функции можно применять более одного  
декоратора. В этом случае их эффекты накапливаются, и именно этот  
факт делает декораторы столь полезными в качестве структурных блоков  
многократного использования.**

**Приведем пример. Представленные ниже два декоратора обертывают  
выходную строку декорированной функции в HTML-теги. Глядя на то,  
как теги вложены, вы видите, в каком порядке Python применяет много-  
численные декораторы:**

**def strong(func):  
def wrapper():**

**return '<strong>' + func() + '</strong>'  
return wrapper**

**def emphasis(func):  
def wrapper():**

**return '<em>' + func() + '</em>'  
return wrapper**

**Теперь давайте возьмем эти два декоратора и одновременно применим их  
к нашей функции greet. Для этого вы можете использовать обычный син-**таксис **@** и просто «уложить» многочисленные декораторы вертикально  
поверх одной-единственной функции:

**@strong  
@emphasis  
def greet():**

**return 'Привет!'**

**Какой результат вы ожидаете увидеть, если выполнить декорированную  
функцию? Сначала декоратор** @emphasis **добавит тег** <em>**? Или же прио-  
ритет имеет тег** @strong**? Когда вы вызываете декорированную функцию,  
происходит вот что:**

**>>> greet()**

***'<strong><em>npueem!</em></strong>'***

**Этот результат ясно показывает, в каком порядке декораторы были приме-  
нены: снизу вверх. Сначала входная функция была обернута декоратором**@emphasis**, и затем результирующая (декорированная) функция снова  
была обернута декоратором** @strong**.**

**Чтобы помочь себе запомнить порядок следования снизу-вверх, мне  
нравится называть такое поведение стековой укладкой декораторов. Стек  
начинают строить снизу и затем продолжают добавлять новые блоки по-  
верх старых, поднимаясь все выше и выше.**

**Если разложить приведенный выше пример и избавиться от синтаксиса** @**,  
который применяют декораторы, то цепочка вызовов функций-декорато-  
ров выглядит так:**

**decorated\_greet = strong(emphasis(greet))**

**И снова вы видите, что сначала применяется декоратор** emphasis **и за-  
тем результирующая обернутая функция снова обертывается декорато-  
ром** strong**.**

**Это также означает, что глубокие уровни стековой укладки декораторов  
в конечном счете скажутся на производительности, потому что они будут  
добавлять все новые вызовы вложенных функций. На практике в этом нет  
проблем, но имейте это в виду, если работаете над вычислительно емким  
программным кодом, в котором декорирование применяется часто.**

**Декорирование функций, принимающих аргументы**

**Все примеры пока что декорировали только простую нульарную функ-  
цию** greet**, которая вообще не принимала никаких аргументов. Вплоть  
до этого момента декораторам, которые вы здесь видели, не было дела до  
переадресации аргументов во входную функцию.**

**Если применить один из этих декораторов к функции, которая прини-  
мает аргументы, то она не заработает правильно. Тогда как декорировать  
функцию, которая принимает произвольные аргументы?**

**Вот где на помощь приходят функциональные средства языка Python**\*args **и** \*\*kwargs **для работы с неизвестными количествами аргументов[[18]](#footnote-18).  
Ниже приведен декоратор** proxy**, в котором задействуется их преимуще-  
ство:**

**def proxy(func):**

**def wrapper(\*args, \*\*kwargs):  
return func(\*args, \*\*kwargs)  
return wrapper**

**С этим декоратором происходят две вещи, заслуживающие внимания:**

* **В определении замыкания** wrapper **он использует операторы** \* **и** \*\***,  
  чтобы собрать все позиционные и именованные аргументы, и помещает  
  их в переменные (**args **и** kwargs**).**
* **Замыкание** wrapper **затем переадресует собранные аргументы в ори-  
  гинальную входную функцию, используя операторы «распаковки  
  аргументов» \* и** \*\***.**

**К сожалению, в Python значение операторов «звездочка» и «двойная  
звездочка» перегружено и меняется в зависимости от контекста, в котором  
они используются, но надеюсь, вы уловили идею.**

**Давайте расширим прием, сформулированный декоратором proxy, в бо-  
лее полезный практический пример. Ниже приведен декоратор trace,  
который регистрирует аргументы функции и итоговые результаты, полу-  
ченные во время исполнения:**

**def trace(func):**

**def wrapper(\*args, \*\*kwargs):**

**print(f'ТРАССИРОВКА: вызвана {func. name }() '**

**f'c {args}, {kwargs}')**

**original\_result = func(\*args, \*\*kwargs)**

**print(f'ТРАССИРОВКА: {func. name }() '**

**ввернула {original\_result!r}')  
return original\_result  
return wrapper**

**При декорировании функции с использованием декоратора trace и после-  
дующем ее вызове, будут выведены переданные в декорированную функ-  
цию аргументы и возвращаемое ею значение. Этот пример по-прежнему  
остается несколько «игрушечным» — но в случае крайней необходимости  
он становится отличным средством отладки:**

**@trace def say(name, line):  
return f'{name}: {line}'**

**>>> say('Джейн', 'Привет, Мир')**

***'ТРАССИРОВКА: вызвана say() с ("Джейн", "Привет, Мир"), {}'***

***'ТРАССИРОВКА: say() вернула "Джейн: Привет, Мир"'***

***'Джейн: Привет, Мир'***

**Если говорить об отладке, то существует две вещи, которые при отладке  
декораторов следует иметь в виду.**

**Как писать «отлаживаемые» декораторы**

**При использовании декоратора вы на самом деле только подменяете одну  
функцию другой. Оборотной стороной этого процесса является то, что он  
«скрывает» некоторые метаданные, закрепленные за оригинальной (не-  
декорированной) функцией.**

**Например, оригинальное имя функции, ее строка документации docstring  
и список параметров скрыты замыканием-оберткой:**

**def greet():**

***"""Вернуть дружеское приветствие."""*return 'Привет!'**

**decorated\_greet = uppercase(greet)**

**При попытке получить доступ к каким-либо из этих метаданных функции  
вместо них вы увидите метаданные замыкания-обертки:**

**>>> greet. name**

**'greet'**

**>>> greet. doc**

***'Вернуть дружеское приветствие.'***

**>>> decorated\_greet. name**

***'wrapper'***

**>>> decorated\_greet. doc**

**None**

**Это делает отладку и работу с интерпретатором Python неуклюжей и тру-  
доемкой. К счастью, существует быстрое решение этой проблемы: деко-  
ратор functools.wraps, включенный в стандартную библиотеку Python[[19]](#footnote-19).**

**Декоратор functools.wraps можно использовать в своих собственных  
декораторах для того, чтобы копировать потерянные метаданные из не-  
декорированной функции в замыкание декоратора. Вот пример:**

**import functools  
def uppercase(func):**

**@functools.wraps(func)  
def wrapper():**

**return func().upper()  
return wrapper**

**Применение декоратора functools.wraps к замыканию-обертке, возвра-  
щаемому декоратором, переносит в него строку документации и другие  
метаданные входной функции:**

**^uppercase def greet():**

***"""Вернуть дружеское приветствие."""*return 'Привет!'**

**>>> greet. name**

***'greet'***

**>>> greet. doc**

***'Вернуть дружеское приветствие.'***

**В качестве оптимального практического приема я порекомендовал бы ис-  
пользовать декоратор functools.wraps во всех декораторах, которые вы  
пишете сами. Это не займет много времени и уменьшит головную боль  
вам (и другим) в будущем при отладке.**

**Да, и поздравляю — вы успешно добрались до самого конца этого слож-  
ного раздела и многое узнали о декораторах в Python. Отличная работа!**

**Ключевые выводы**

* **Декораторы определяют структурные блоки многократного исполь-  
  зования, которые можно применять к вызываемому объекту с целью  
  модификации его поведения без необратимого изменения самого вы-  
  зываемого объекта.**
* **Синтаксис @ является всего-навсего сокращенной записью для вызова  
  декоратора с входной функцией. Многочисленные декораторы, разме-  
  щенные над одной-единственной функцией, применяются снизу-вверх  
  (стековая укладка декораторов).**
* **В качестве оптимального практического приема отладки исполь-  
  зуйте в своих собственных декораторах вспомогательный декоратор  
  functools.wraps, чтобы переносить метаданные из недекорированного  
  вызываемого объекта в декорированный.**
* **Точно так же, как и любой другой инструмент в комплекте инструмен-  
  тов разработки программного обеспечения, декораторы не являются  
  панацеей, и ими не стоит злоупотреблять. Важно уравновесить необ-  
  ходимость «довести дело до конца» с целевой установкой «не увязнуть  
  в ужасной и неудобной в обслуживании мешанине кодовой базы».**

**3.4 . Веселье с \*args и \*\*kwargs**

**Однажды я программировал в паре с суровым питонистом, который то  
и дело восклицал «арррг!» и «кварг!» всякий раз, когда набирал опре-  
деление функции с необязательными или именованными параметрами.  
Во всем остальном мы отлично поладили. М-да, вот что в итоге делает  
с людьми программирование в академии...**

**Нужно сказать, что сколько бы ни потешались над параметрами** \*args  
**и** \*\*kwargs**, тем не менее они являются очень полезным функциональным  
средством языка Python. И понимание их потенциала сделает из вас более  
эффективного разработчика.**

**Итак, для чего же используются параметры** \*args **и** \*\*kwargs**? Они по-  
зволяют функции принимать необязательные аргументы, благодаря чему  
вы можете создавать гибкие API в модулях и классах:**

**def foo(required, \*args, \*\*kwargs):  
print(required)  
if args:**

**print(args)  
if kwargs:**

**print(kwargs)**

**Приведенная выше функция требует по крайней мере одного аргумента  
под названием «**required**», то есть обязательный, но она также может при-  
нимать дополнительные позиционные и именованные аргументы.**

**Если мы вызовем функцию с дополнительными аргументами, то** args  
**соберет дополнительные позиционные аргументы в кортеж, потому что  
имя параметра имеет префикс \*.**

**Аналогичным образом,** kwargs **соберет дополнительные именованные  
аргументы в словарь, потому что имя параметра имеет префикс \*\*.**

**Как** args**, так и** kwargs **могут быть пустыми, если никакие дополнительные  
аргументы в функцию не переданы.**

**Когда мы вызываем функцию с различными комбинациями аргументов,  
вы видите, как Python собирает их в параметрах** args **и** kwargs **в соответ-**ствии с тем, являются они позиционными или именованными аргумен-  
тами:

**>>> foo()**

***TypeError:***

***"foo() missing 1 required positional arg: 'required'"***

**>>> foo('привет')  
привет**

**>>> foo('привет', 1, 2, 3)**

**привет**

**(1, 2, 3)**

**>>> foo('привет', 1, 2, 3, кеу1='значение', key2=999)**

**привет**

**(1, 2, 3)**

**{'key1': 'значение', 'key2': 999}**

**Сразу хочу прояснить. Название параметров args и kwargs принято по  
договоренности, как согласованное правило именования. Приведенный  
выше пример будет работать точно так же, если вы назовете их \*parms  
и \*\*argv. Фактическим синтаксисом является, соответственно, просто  
звездочка (\*) или двойная звездочка (\*\*).**

**Однако чтобы избежать недоразумений, я рекомендую придерживаться  
общеринятого согласованного правила именования. (И иметь возмож-  
ность время от времени рычать «арррг!» и «кваррррг!».)**

**Переадресация необязательных  
или именованных аргументов**

**Существует возможность передавать необязательные или именованные  
параметры из одной функции в другую. Это можно делать при помощи  
операторов распаковки аргументов \* и \*\* во время вызова функции, в ко-  
торую вы хотите переадресовать аргументы[[20]](#footnote-20).**

**Это также дает вам возможность модифицировать аргументы перед тем,  
как вы передадите их дальше. Вот пример:**

**def foo(x, \*args, \*\*kwargs):  
kwargs['имя'] = 'Алиса'  
new\_args = args + ('дополнительный', )  
bar(x, \*new\_args, \*\*kwargs)**

**Данный прием может быть полезен для создания производных классов  
и написания оберточных функций. Например, он может применяться  
для расширения поведения родительского класса без необходимости  
повторять полную сигнатуру его конструктора в дочернем классе. Это  
может быть довольно удобно, если вы работаете с API, который может  
измениться за пределами вашего контроля:**class Car:

**def init (self, color, mileage):**

**self.color = color  
self.mileage = mileage  
class AlwaysBlueCar(Car):**

**\*args,  
\_init (\*args**

**\*\*kwargs):  
\*\*kwargs)**

**def init (self,**

**super().  
self.color = 'синий'**

**>>> AlwaysBlueCar('зеленый', 48392).color  
'синий'**

**Конструктор класса AlwaysBlueCar просто передает все аргументы в свой  
родительский класс и затем переопределяет внутренний атрибут. Это оз-  
начает, что если конструктор родительского класса изменится, то велика  
вероятность того, что AlwaysBlueCar будет по-прежнему функционировать  
как было задумано.**

**Оборотной стороной здесь является то, что конструктор AlwaysBlueCar  
теперь имеет довольно бесполезную сигнатуру, — мы не узнаем, какие  
аргументы он ожидает, не заглянув в родительский класс.**

**Как правило, вы не будете использовать этот прием со своими собствен-  
ными иерархиями классов. Более вероятный сценарий будет такой, что вы  
захотите изменить или переопределить поведение в некотором внешнем  
классе, которым не управляете.**

**Но это всегда опасная территория, поэтому лучше соблюдать осторож-  
ность (иначе вскоре у вас, возможно, появится еще одна причина вос-  
кликнуть «аррррг!»).**

**Еще один сценарий, где этот прием будет потенциально полезен, — на-  
писание оберточных функций, таких как декораторы. Там вы также захо-  
тите принимать произвольные аргументы, которые будут переправляться  
в обернутую функцию.**

**И если мы можем сделать это без необходимости копипастить сигнатуру  
оригинальной функции, то, возможно, сопровождение станет удобнее:**

**def trace(f):**

**@functools.wraps(f)**

**def decorated\_function(\*args, \*\*kwargs):  
print(f, args, kwargs)  
result = f(\*args, \*\*kwargs)  
print(result)  
return decorated\_function  
@trace**

**def greet(greeting, name):**

**return '{}, {}!'.format(greeting, name)**

**>>> greet('Привет', 'Боб')**

**<function greet at 0x1031c9158> ('Привет', 'Боб') {}**

***'Привет, Боб!'***

**Такого рода приемами иногда трудно уравновесить идею сделать ваш про-  
граммный код достаточно четким и при этом остаться в рамках принципа  
DRY[[21]](#footnote-21). Это всегда будет нелегким выбором. Если у вас есть возможность  
получить другое мнение от коллеги, то я призываю вас — спрашивайте.**

**Ключевые выводы**

**□ В Python переменные \*args и \*\*kwargs позволяют писать функции  
с неизвестным количеством аргументов.**

* **Переменная \*args собирает дополнительные позиционные аргументы  
  в кортеж. Переменная \*\*kwargs собирает дополнительные именован-  
  ные аргументы в словарь.**
* **Фактическим синтаксисом является \* и \*\*. Названия args и kwargs —  
  это просто договоренность (которой следует придерживаться).**

**3.5 . Распаковка аргументов функции**

**Действительно крутым, но немного загадочным функциональным сред-  
ством языка является способность «распаковывать» аргументы функции  
из последовательностей и словарей при помощи операторов \* и \*\*.**

**Давайте определим простую функцию для работы в качестве примера:**

**def print\_vector(x, y, z):**

**print('<%s, %s, %s>' % (x, y, z))**

**Как вы видите, эта функция принимает три аргумента (x, y и z) и печатает  
их в приятно отформатированном виде. Мы можем применить эту функ-  
цию в нашей программе для структурной распечатки трехмерных векторов:**

**>>> print\_vector(0, 1, 0)**

**<0, 1, 0>**

**Нужно сказать, что в зависимости от того, какую структуру данных мы  
выбираем для представления трехмерных векторов, их распечатка нашей  
функцией print\_vector может восприниматься немного неуклюжей. На-  
пример, если наши векторы представлены в виде кортежей или списков,  
то во время их распечатки мы должны явным образом задавать индекс  
каждого компонента:**

**>>> tuple\_vec = (1, 0, 1)**

**>>> list\_vec = [1, 0, 1]**

**>>> print\_vector(tuple\_vec[0],  
tuple\_vec[1],  
tuple\_vec[2])**

**<1, 0, 1>**

**Обычный вызов функции с отдельными аргументами кажется излиш-  
не многословным и громоздким. Не лучше ли будет просто развернуть  
векторный объект на три его компонента и передать их все одним разом  
в функцию** print\_vector**?**

**(Разумеется, вы могли бы просто переопределить функцию** print\_vector  
**так, чтобы она принимала один-единственный параметр, представляющий  
векторный объект, но ради того, чтобы иметь простой пример, мы этот  
вариант пока проигнорируем.)**

**К счастью, в Python имеется более подходящий способ справиться с этой  
ситуацией — при помощи распаковки аргументов функции с использова-  
нием оператора \*:**

**>>> print\_vector(\*tuple\_vec)**

**<1, 0, 1>**

**>>> print\_vector(\*list\_vec)**

**<1, 0, 1>**

**Размещение звездочки \* перед итерируемым объектом в вызове функции  
его распакует и передаст его элементы как отдельные позиционные аргу-  
менты в вызванную функцию.**

**Этот прием работает для любого итерируемого объекта, включая выраже-  
ния-генераторы. В результате использования оператора \* с генератором  
все поступающие из генератора элементы будут использованы и переданы  
в функцию:**

**>>> genexpr = (x \* x for x in range(3))**

**>>> print\_vector(\*genexpr)**

**Помимо оператора \* для распаковки последовательностей, в частности  
кортежей, списков и генераторов, в позиционные аргументы, также име-  
ется оператор \*\* для распаковки именованных аргументов, поступающих  
из словарей. Предположим, что наш вектор был представлен в виде сле-  
дующего объекта** diet**:**

**>>> dict\_vec = {'y': 0, 'z': 1, 'x': 1}**

**Этот объект-словарь можно передать в функцию** print\_vector **практиче-  
ски таким же образом, использовав оператор \*\* для распаковки:  
>>> print\_vector(\*\*dict\_vec)**

**<1, 0, 1>**

**Поскольку словари не упорядочены, этот оператор соотносит значения  
словаря и аргументы функции на основе ключей словаря: аргумент x  
получает значение, связанное в словаре с 'x'.**

**Если для распаковки словаря использовать оператор одинарной звездоч-  
ки (\*), то вместо этого ключи будут переданы в функцию в произвольном  
порядке:**

**>>> print\_vector(\*dict\_vec)**

**<y, x, z>**

**Функциональное средство языка Python, связанное с распаковкой ар-  
гументов функции, предоставляет вам большую гибкость за бесплатно.  
Зачастую это означает, что вам не придется реализовывать класс для не-  
обходимого вашей программе типа данных. При этом вполне достаточно  
обойтись простыми встроенными структурами данных, подобными кор-  
тежам или спискам, и, как результат, это поможет уменьшить сложность  
вашего программного кода.**

**Ключевые выводы**

* **Операторы \* и \*\* могут использоваться для «распаковки» аргументов  
  функции, поступающих из последовательностей и словарей.**
* **Эффективное применение распаковки аргументов будет способство-  
  вать написанию более гибких интерфейсов для ваших модулей и функ-  
  ций.**

1. **. Здесь нечего возвращать**

**В конец любой функции Python добавляет неявную инструкцию return  
None. По этой причине, если в функции не указано возвращаемое значе-  
ние, по умолчанию она возвращает None.**

**Это означает, что инструкции** return None **можно заменять на пустые  
инструкции** return **или даже пропускать их полностью и по-прежнему  
получать тот же самый результат:**

**def fool(value):  
if value:**

**return value**

**else:**

**return None  
def foo2(value):**

***"""Пустая инструкция return подразумевает 'return None'"""*if value:**

**return value  
else:**

**return  
def foo3(value):**

***"""Пропущенная инструкция return подразумевает 'return None'"""*if value:**

**return value**

**Все три функции правильно возвращают** None**, если передать им в качестве  
единственного аргумента фиктивное значение:**

**>>> type(fooi(0))**

**<class 'NoneType'>**

**>>> type(foo2(0))**

**<class 'NoneType'>**

**>>> type(foo3(0))**

**<class 'NoneType'>**

**Итак, когда же лучше всего использовать это функциональное средство  
языка Python в своем собственном программном коде?**

**Мое эмпирическое правило заключается в следующем: если функция не  
имеет возвращаемого значения (в других языках такая функция называ-  
ется процедурой), то я исключаю инструкцию** return**. Добавлять эту ин-  
струкцию было бы лишним и вносило бы путаницу. Примером процедуры  
является встроенная в Python функция печати** print**, которая вызывается**

**только ради ее побочных эффектов (распечатки текста) и никогда — ради  
ее возвращаемого значения.**

**Давайте возьмем функцию, например встроенную в Python функцию**sum**. Она, безусловно, имеет логическое возвращаемое значение, и, как  
правило, функция** sum **не вызывается только ради ее побочных эффектов.  
Ее цель состоит в том, чтобы подсчитать сумму последовательности чи-  
сел и затем представить результат. Итак, если с логической точки зрения  
функция действительно имеет возвращаемое значение, то необходимо  
решить, использовать неявную инструкцию** return **или нет.**

**С одной стороны, вы можете утверждать, что исключение явной инструк-  
ции** return None **делает программный код более сжатым и, следовательно,  
более легким для чтения и понимания. Субъективно вы отметили бы, что  
это делает программный код «симпатичнее».**

**С другой стороны, некоторые программисты могут удивиться, что Python  
ведет себя таким образом. В том, что касается написания чистого и удоб-  
ного в сопровождении программного кода, такое непредсказуемое пове-  
дение редко является хорошим признаком.**

**Например, я использовал «неявную инструкцию возврата» в одном из  
примеров исходного кода в более ранней версии этой книги. Я не говорил  
о том, что делал, — мне просто нужен был безупречный короткий образец  
кода для объяснения какого-то другого функционального средства языка  
Python.**

**В итоге я начал получать непрекращающийся поток электронных писем,  
указывающих мне на «пропущенную» в том примере кода инструкцию  
возврата. Неявное поведение инструкции** return **в Python отнюдь не  
было очевидным для всех и в данном случае отвлекало от сути. Я добавил  
примечание, чтобы прояснить, что происходит, и электронные письма  
прекратились.**

**Не поймите меня превратно: я люблю писать чистый и «красивый» про-  
граммный код так же, как и любой другой разработчик. И раньше я тоже  
был твердо убежден, что программисты должны знать всю подноготную  
языка, с которым они работают.**

**Но когда вы рассматриваете влияние даже такого простого недоразумения  
на сопровождение кода, возможно, имеет смысл склониться к написанию  
более явного и четкого программного кода. В конце концов, программный  
код — это общение.**

**Ключевые выводы**

* **Если в функции не указано возвращаемое значение, то она возвращает  
  None. Возвращать None явным образом или неявным, решается стили-  
  стически.**
* **Это ключевое функциональное средство языка Python, однако ваш  
  программный код может передавать свое намерение четче при помощи  
  явной инструкции return None.**

1. **.1 . Сравнения объектов: is против ==**

**Когда я был мальчишкой, у наших соседей жили кошки-близняшки.  
Внешне они были очень похожи — одинаковая темно-серая шерсть и оди-  
наковый пронизывающий взгляд зеленых глаз. Отбросив некоторые инди-  
видуальные особенности, на глаз вы бы их не различили. Но, конечно, они  
были двумя разными кошками, двумя отдельными существами, несмотря  
на то что выглядели одинаково.**

**Это подводит меня к разнице в смысле между понятиями «равенство»  
и «тождество». И эта разница крайне важна для понимания того, как  
ведут себя операторы сравнения Python** is **и** ==**.**

**Оператор** == **выполняет сравнение путем проверки на равенство: если бы  
эти кошки были объектами Python и мы сравнивали их оператором** ==**, то  
в качестве ответа мы получили бы, что «обе кошки равны».**

**Однако оператор** is **сравнивает идентичности: если бы мы сравнивали  
наших кошек оператором** is**, то в качестве ответа мы получили бы, что  
«это две разные кошки».**

**Но прежде чем я запутаюсь в этом кошачьем клубке, давайте взглянем на  
небольшой реальный код Python.**

**Прежде всего, мы создадим новый объект-список и назовем его** а**, а затем  
определим еще одну переменную (**b**), которая указывает на тот же самый  
объект-список:**

**>>> a = [1, 2, 3]**

**>>> b = a**

**Давайте изучим эти две переменные. Мы видим, что они указывают на  
внешне идентичные списки:**

|  |  |  |
| --- | --- | --- |
| **>>>** | **a** |  |
| **[1,** | **2,** | **3]** |
| **>>>** | **b** |  |
| **[1,** | **2,** | **3]** |

**Когда мы сравним эти два объекта-списка на равенство при помощи опе-  
ратора ==, мы получим ожидаемый результат, поскольку эти два объекта-  
списка выглядят одинаково:**

**>>> a == b  
True**

**Однако этот результат не говорит о том, указывают ли a и b в действитель-  
ности на тот же самый объект. Конечно, мы знаем, что это так, потому что  
мы определили их ранее, но предположим, что мы не знаем, — тогда как  
можно было бы это узнать?**

**Ответ на этот вопрос следует искать в сравнении обеих переменных опе-  
ратором is. Это сравнение подтверждает, что обе переменные в действи-  
тельности указывают на один объект-список:**

**>>> a is b**

**True**

**Давайте посмотрим, что происходит, когда мы создаем идентичную копию  
нашего объекта-списка. Это можно сделать, вызвав list() с существую-  
щим списком в качестве аргумента, чтобы создать копию, которую мы  
назовем с:**

**>>> с = list(a)**

**И снова вы увидите, что только что созданный нами новый список вы-  
глядит идентичным объекту-списку, на который указывают a и b:**

**>>> с  
[1, 2, 3]**

**А вот теперь начинается самое интересное. Давайте сравним нашу копию  
списка c с первоначальным списком а, использовав для этого оператор ==.  
Какой ответ вы ожидаете увидеть?**

**>>> а == c  
True**

**О’кей. Надеюсь, что вы как раз этого и ожидали. Данный результат гово-  
рит следующее: c и а имеют одинаковое содержимое. Python их считает  
равными. Но вот вопрос: указывают ли они в действительности на один  
и тот же объект? Давайте это выясним при помощи оператора is:**

**>>> а is c  
False**

**О-па! И вот тут мы получаем другой результат. Python говорит, что c  
и а указывают на два разных объекта, несмотря на то что их содержимое  
может быть одинаковым.**

**Итак, чтобы подытожить, давайте попробуем разложить разницу между  
is и == на два коротких определения:**

* **Выражение is дает True, если две переменные указывают на тот же  
  самый (идентичный) объект.**
* **Выражение == дает True, если объекты, на которые ссылаются пере-  
  менные, равны (имеют одинаковое содержимое).**

**Всякий раз, когда вам придется решать, применять оператор is или опе-  
ратор ==, просто вспомните про кошек-близняшек (в принципе, сойдут  
и собаки). Если вы это будете делать, то у вас все будет в порядке.**

1. **. Преобразование строк  
   (каждому классу по repr )**

**Когда вы определяете собственный класс в Python и затем пытаетесь на-  
печатать один из его экземпляров в консоли (или проверить его в сеансе  
интерпретатора), вы получаете относительно неудовлетворительный**результат. Принятое по умолчанию поведение с преобразованием в стро-  
ковое значение в стиле «to-string» является примитивным и испытывает  
недостаток в подробностях:

**class Car:**

**def init (self, color, mileage):**

**self.color = color  
self.mileage = mileage**

**>>> my\_car = Саг('красный', 37281)**

**>>> print(my\_car)**

**< console .Car object at 0x109b73da0>**

**>>> my\_car**

**<\_\_console\_\_.Car object at 0x109b73da0>**

**По умолчанию вы получаете лишь строковое значение, содержащее имя  
класса и идентификатор экземпляра объекта (который в Python являет-  
ся адресом объекта в оперативной памяти). Это лучше, чем ничего, но не  
очень-то полезно.**

**Вы можете попытаться найти обходной путь, непосредственно распе-  
чатав атрибуты класса или даже добавив в классы собственный метод**to\_stringO**:**

**>>> print(my\_car.color, my\_car.mileage)  
красный 37281**

**Общая идея совершенно верная, но она игнорирует договоренности  
об именовании и встроенные механизмы, которые Python использует  
для обработки того, как объекты представляются в виде строк.**

**Вместо того чтобы строить свой собственный механизм преобразования  
строк, будет гораздо лучше, если вы добавите в свой класс дандер-ме-**

**тоды** str **и** repr **. Они представляют собой питоновский способ**

**управления тем, как объекты преобразовываются в строковые значения  
в различных ситуациях1.**

**См. документацию Python «Модель данных Python»:** [**https://docs**](https://docs) **. python, org/3/  
reference/datamodel . html**

**Давайте взглянем, как эти методы работают на практике. Для начала мы  
добавим метод** str **в класс** Car**, который мы определили ранее:**

**class Car:**

**def init (self, color, mileage):**

**self.color = color  
self.mileage = mileage**

**def str (self):**

**return f'{self.color} автомобиль'**

**Если сейчас попробовать напечатать или проинспектировать экземпляр**Car**, то вы получите другой, слегка улучшенный результат:**

**>>> my\_car = Car('красный', 37281)**

**>>> print(my\_car)**

***'красный автомобиль'***

**>>> my\_car**

**< console .Car object at 0x109ca24e0>**

**Инспектирование объекта** Car **в консоли по-прежнему дает предыдущий  
результат, содержащий идентификатор объекта. Однако распечатка объ-  
екта показала строку, возвращенную добавленным нами методом** str **.**

**Метод** str **является одним из дандер-методов Python (с двойным**

**подчеркиванием), и он вызывается, когда вы пытаетесь преобразовать  
объект в строковое значение посредством различных доступных способов:**

**>>> print(my\_car)  
красный автомобиль  
>>> str(my\_car)**

***'красный автомобиль'***

**>>> '{}'.format(my\_car)**

***'красный автомобиль'***

**При надлежащей реализации** str **вам не придется переживать по**

**поводу печати атрибутов объектов непосредственно или написания от-  
дельной функции** to\_stringO**. Это питоновский способ управлять пре-  
образованием строк.**

**Между прочим, некоторые разработчики предпочитают называть дан-  
дер-методы Python «магическими». Но эти методы никоим образом  
магическими не являются. То, что имена этих методов начинаются и окан-  
чиваются символами двойного подчеркивания, является всего-навсего  
согласованным правилом именования, которое выделяет их как ключевые  
функциональные средства языка Python. Он также помогает избежать  
конфликтов из-за совпадения имен с вашими собственными методами**

**и атрибутами. Конструктор объектов** init **соблюдает то же самое**

**правило, и в этом нет ничего волшебного или загадочного.**

**Не бойтесь использовать дандер-методы Python — они призваны вам  
помогать.**

**Метод str против repr**

**Нужно сказать, что наша история преобразования строк на этом не за-  
канчивается. Вы заметили, что осмотр объекта my\_car в сеансе интер-  
претатора по-прежнему дает этот странный результат <Car object at  
0x109ca24e0>?**

**Это произошло, потому что фактически имеется два дандер-метода, кото-  
рые управляют тем, как объекты преобразовываются в строковые значе-  
ния в Python 3. Первый, str , и вы только что с ним познакомились.**

**Второй, repr , и характер его работы аналогичен методу str , но**

**он используется в других ситуациях. (В Python 2.x также имеется метод  
 Unicode , которого я коснусь чуть позже.)**

**Ниже приведен простой эксперимент для «обкатки» ситуации, когда**

**используется метод str или repr . Давайте переопределим наш**

**автомобильный класс таким образом, чтобы он содержал оба дандер-ме-  
тода для преобразования в строковое значение с результатами, которые  
легко различить:**

**class Car:**

**def init (self, color, mileage):**

**self.color = color  
self.mileage = mileage**

**def repr (self):**

**return ' repr для объекта Car'**

**def str (self):**

**return ' str для объекта Car'**

**Теперь, когда вы поэкспериментируете с приведенными выше примерами,  
вы поймете, какой метод управляет результатом преобразования строк  
в каждом случае:**

**>>> my\_car = Саг('красный', 37281)**

**>>> print(my\_car)**

**str для объекта Car**

**>>> '{}'.format(my\_car)**

***'* *str* *для объекта Car'***

**>>> my\_car**

**repr для объекта Car**

**Этот эксперимент подтверждает, что в результате инспектирования объ-  
екта в сеансе интерпретатора Python просто печатается результат выпол-  
нения метода repr объекта.**

**Интересно отметить, что в контейнерах, таких как списки и словари,  
для представления содержащихся в них объектов всегда используется  
результат метода repr , даже если вызвать функцию str с самим кон-**

**тейнером:**

**str([my\_car])**

***'[* *repr* *для объекта Car]'***

**Что касается ручного выбора между обоими методами преобразования  
строк, например, чтобы яснее выразить замысел вашего программного  
кода, то лучше всего использовать встроенные функции str() и repr().**

**Их применение предпочтительнее прямых вызовов метода str или**

**repr объекта, поскольку они воспринимаются лучше и дают тот же**

**самый результат:**

**>>> str(my\_car)**

***'* *str* *для объекта Car'***

**>>> repr(my\_car)**

***'* *repr* *для объекта Car'***

**Даже по завершении этого исследования вам, возможно, будет любопытно**

**узнать, какова «реальная» разница между методами str и repr .**

**На вид они оба служат одной и той же цели, поэтому может быть не ясно,  
когда использовать каждый из них.**

**В случае таких вопросов обычно неплохо взглянуть на то, что делает  
стандартная библиотека Python. Самое время поставить еще один экс-  
перимент. Мы создадим объект** datetime.date **и выясним, каким образом  
в нем используются методы** repr **и** str **для управления преоб-**

**разованием строк:**

**>>> import datetime**

**>>> today = datetime.date.today()**

**Результат метода** str **объекта даты должен быть прежде всего удо-**

**бочитаемым. Он призван возвращать легко воспринимаемое человеком  
сжатое текстовое представление — то, что вы спокойно можете показать  
пользователю. По этой причине, когда мы вызываем функцию** str() **с объ-  
ектом даты, мы получаем нечто похожее на формат даты по ISO:**

**>>> str(today)**

***'2017-02-02'***

**В случае с методом** repr **идея состоит в том, что его результат должен**

**быть прежде всего однозначным. Результирующее строковое значение  
больше предназначено для разработчиков как средство отладки. И в  
связи с этим он должен максимально четко выражать то, чем этот объект  
является. Именно поэтому при вызове функции** repr() **с объектом вы  
получите более подробный результат. Он даже будет содержать полное  
имя модуля и класса:**

**>>> repr(today)**

***'datetime.date(2017j 2, 2)'***

**Возвращаемое методом** repr **строковое значение можно скопировать**

**и вставить в консоль интерпретатора и исполнить его как допустимый  
фрагмент кода Python, чтобы воссоздать оригинальный объект даты. Этот  
изящный подход и хороший целевой ориентир следует иметь в виду при  
написании своих собственных функций** repr**.**

**С другой стороны, я полагаю, что довольно-таки трудно найти приме-  
нение такой функции на практике. Она не будет стоить затраченных на  
нее усилий и просто создаст для вас дополнительную работу. Мое эмпи-  
рическое правило заключается в том, чтобы делать свои строки** repr   
однозначными и полезными для разработчиков, но я не рассчитываю,  
что они смогут восстанавливать полное состояние объекта.

**Почему каждый класс нуждается в repr**

**Если опустить метод str , то Python в поисках str отыграет назад**

**к результату repr . По этой причине я рекомендую добавлять в свои**

**классы всегда, по крайней мере, метод repr . Это обеспечит полезный**

**результат преобразования строк почти во всех случаях при минимуме  
работы по его реализации.**

**Ниже показано, как можно быстро и эффективно добавить в свои классы  
элементарную поддержку преобразования строк. Для нашего класса Car  
мы могли бы начать с приведенного ниже метода repr :**

**def repr (self):**

**return f'Car({self.color!r}, {self.mileage!r})'**

**Обратите внимание на то, что я использую флаг преобразования ! г,  
тем самым гарантируя, что в выводимом строковом значении вместо  
str(self.color) и str(self.mileage) будут использованы repr(self.  
color) и repr(self.mileage).**

**Это работает безупречно, но оборотной стороной является то, что мы по-  
вторили имя класса в форматной строке. Для того чтобы избежать этого  
повторения, здесь можно применить трюк с использованием атрибута**

**calss . name объекта. Данный атрибут всегда будет зеркально**

**отображать имя класса в виде строки.**

**Преимущество состоит в том, что вам не придется модифицировать реа-  
лизацию метода repr , когда имя класса изменится. Это позволяет**

**беспрепятственно придерживаться принципа DRY, то есть «не повто-  
ряйся»:**

**def repr (self):**

**return (f'{self. class . name }('**

**f'{self.color!r}, {self.mileage!r})')**

**Оборотной стороной этой реализации является то, что форматная строка  
довольно длинная и громоздкая. Но при условии выверенного формати-  
рования вы сможете сохранить свой исходный код аккуратным и соот-  
ветствующим правилам PEP 8.**

**Во время инспектирования объекта или непосредственного вызова функ-  
ции repr() с объектом при наличии приведенной выше реализации метода  
 repr мы получаем полезный результат:**

**>>> repr(my\_car)**

***'Car(red, 37281)'***

**Печать объекта или вызов функции str() с этим объектом возвращают  
то же самое строковое значение, потому что заданная по умолчанию реа-  
лизация str просто вызывает метод repr :**

**>>> print(my\_car)**

***'Car(red, 37281)'***

**>>> str(my\_car)**

***'Car(red, 37281)'***

**Убежден, что этот подход обеспечивает наибольшую эффективность  
и скромный объем работы по его реализации. Более того, этот подход  
опирается на типовой шаблон в стиле формы для печенья, который можно  
применять, не особо задумываясь. По этой причине я всегда стремлюсь  
добавлять в свои классы элементарную реализацию метода \_\_repr\_\_.**

**Ниже показан законченный пример для Python 3 с дополнительной реа-  
лизацией метода str :**class Car:

**def init (self, color, mileage):**

**self.color = color  
self.mileage = mileage**

**def repr (self):**

**return (f'{self. class . name }('**

**f'{self.color!r}, {self.mileage!r})')**

**def str (self):**

**return f'{self.color} автомобиль'**

**Отличия Python 2 . x: Unicode**

**В Python 3 имеется один тип данных на все случаи жизни для представ-  
ления текста: str. Он содержит символы Юникода и может представлять  
большинство систем письменности в мире.**

**В Python 2.x для строковых данных используется другая модель дан-  
ных[[22]](#footnote-22). Для представления текста служат два типа: str, который ограничен  
набором символов ASCII, и Unicode, который эквивалентен типу str  
Python 3.**

**Вследствие этой разницы в Python 2 существует еще один дандер-ме-**

**тод в составе методов управления преобразованием строк: Unicode .**

**В Python 2 str возвращает байты, тогда как Unicode возвращает**

***символы.***

**По своим замыслу и целям метод Unicode является более новым**

**и предпочтительным методом управления преобразованием строк. Кроме  
того, имеется сопровождающая его встроенная функция unicode(). Она  
вызывает соответствующий дандер-метод подобно тому, как работают  
функции str() и repr().**

**Чем дальше, тем лучше. Но все станет намного причудливее, когда вы  
посмотрите на правила вызова методов str и Unicode в Python 2.**

**Инструкция print и функция str() вызывают метод str . Встро-  
енная в Python 2 функция Unicode() вызывает метод Unicode , если**

**он существует; в противном случае отыгрывает назад к методу str**

**и декодирует результат в системную кодировку текста.**

**По сравнению с Python 3 эти особые случаи несколько усложняют пра-  
вила преобразования текста. Но есть способ все снова упростить в прак-  
тическом плане. Юникод является предпочтительным и перспективным  
способом работы с текстом в программах Python.**

**Поэтому в Python 2.x я в целом рекомендовал бы размещать весь свой  
код форматирования строк внутрь метода Unicode , а затем создавать**

**реализацию заглушки** str **, которая возвращает представление в виде**

**Юникода в кодировке UTF-8:**

**def str (self):**

**return unicode(self).encode('utf-8')**

**Заглушка** str **будет одинаковой для большинства классов, ее вы про-**

**сто можете копипастить повсюду, где это необходимо (либо разместить ее  
в базовом классе, где это имеет смысл). Тогда весь ваш код преобразова-  
ния строк, который предназначен для использования не разработчиками,  
будет лежать в методе** Unicode **.**

**Приведем законченный пример для Python 2.x:**

**class Car(object):**

**def init (self, color, mileage):**

**self.color = color  
self.mileage = mileage**

**def repr (self):**

**return '{}({!r}, {!r})'.format(**

**self. class . name ,**

**self.color, self.mileage)  
def \_\_Unicode\_\_(self):**

**return u'{self.color} автомобиль'.format(  
self=self)**

**def \_\_str\_\_(self):**

**return unicode(self).encode('utf-8')**

**Ключевые выводы**

* **Управлять преобразованием строк в своих собственных классах можно,**

**используя дандер-методы str и repr .**

* **Результат метода str должен быть удобочитаемым. Результат мето-  
  да repr должен быть однозначным.**
* **В свои классы всегда следует добавлять метод repr . По умолчанию**

**реализация метода str просто вызывает метод repr .**

* **В Python 2 вместо метода str следует использовать метод uni-**

**code .**

1. **. Определение своих собственных  
   классов-исключений**

**Когда я начал использовать Python, то не решался в своем программном  
коде писать собственные классы-исключения. Вместе с тем определение  
собственных типов ошибок может быть очень ценным. Вы четко выделите  
потенциальные случаи ошибок, и, как результат, ваши функции и модули  
станут более удобными в сопровождении. Вы также сможете использовать  
собственные типы ошибок, которые обеспечат дополнительную отладоч-  
ную информацию.**

**Все это улучшит ваш программный код и облегчит его понимание. Он  
станет легче для отладки и удобнее в сопровождении. Задача определения  
ваших собственных классов-исключений не будет такой сложной, когда  
вы разобьете ее на несколько простых примеров. В этой главе я проведу  
вас по основным пунктам, которые необходимо помнить.**

**Допустим, что вы хотели бы выполнить валидацию входного строкового  
значения, которое в вашем приложении представляет имя человека. Игру-  
шечный пример функции валидации имени может выглядеть следующим  
образом:**

**def validate(name):**

**if len(name) < 10:  
raise VaiueErrar**

**Если валидация терпит неудачу, она вызывает исключение ValueError.  
Это кажется вполне уместным и выглядит по-питоновски. Пока что все  
идет неплохо.**

**Вместе с тем в использовании универсального класса-исключения «вы-  
сокого уровня» типа ValueError есть оборотная сторона. Предположим,  
что один из ваших коллег вызывает эту функцию как составную часть  
библиотеки и не очень разбирается в ее внутреннем устройстве. Когда  
не удается выполнить валидацию имени, отчет об обратной трассировке  
будет выглядеть примерно так:**

**>>> validate('джо')**

**Traceback (most recent call last):**

**File "<input>", line 1, in <module>  
validate('джо')**

**File "<input>", line 3, in validate  
raise VaiueErrar  
VaiueErrar**

**Этот отчет не очень-то и полезен. Несомненно, мы знаем, что что-то по-  
шло не так и что проблема имела отношение к «неправильному значе-  
нию» или типа того, но чтобы быть в состоянии исправить эту проблему,  
ваш коллега почти наверняка должен свериться с реализацией функции  
validate(). Однако чтение исходного кода стоит времени. И это время  
будет быстро накапливаться.**

**К счастью, мы можем сделать и кое-что получше. Введем собственный  
тип исключений, который будет представлять неудавшуюся валидацию  
имени. Мы построим наш новый класс-исключение на основе встроенно-  
го в Python класса ValueError, но заставим его говорить за себя, дав ему  
более конкретное имя:**

**class NameTooShortError( VaiueError):  
pass**

**def validate(name):**

**if len(name) < 10:**

**raise NameTooShortError(name)**

**Теперь у нас есть «самодокументирующий» тип исключений  
NameTooShortError («Имя слишком короткое»), который расширяет  
встроенный класс ValueError. Обычно вы будете делать свои собственные  
исключения производными от корневого класса Exception либо от других  
встроенных в Python исключений наподобие ValueError или TypeError —  
в зависимости от того, что кажется целесообразным.**

**Кроме того, обратите внимание на то, как мы теперь передаем перемен-  
ную name в конструктор нашего собственного класса-исключения во время  
создания его экземпляра внутри validate. Новая реализация приводит  
к тому, что ваш коллега получит намного более приятный отчет об обрат-  
ной трассировке:**

**>>> validate('джейн')**

**Traceback (most recent call last):**

**File "<input>", line 1, in <module>  
validate('джейн')**

**File "<input>", line 3, in validate  
raise NameTooShortError(name)  
NameTooShortError: джейн**

**Опять-таки, попытайтесь встать на место своего коллеги по команде.  
Собственные классы исключений существенно помогают понять, что  
именно происходит, когда дела идут не так, как надо (а рано или поздно  
это обязательно случится).**

**То же самое верно, даже если вы работаете над кодовой базой в полном  
одиночестве. Несколько недель или месяцев спустя вам будет намного  
проще выполнять сопроводительную работу, если ваш исходный код будет  
хорошо структурирован.**

**Потратив всего 30 секунд на определение простого класса-исключения,  
вы уже получили намного больший коммуникативный фрагмент кода.  
Но давайте пойдем дальше. Еще много чего нужно обследовать.**

**Всякий раз, когда вы выпускаете пакет Python в публичное пространство  
или создаете модуль многократного использования для своей компании,  
образцовая практика предусматривает создание для такого модуля соб-  
ственного базового класса-исключения и затем создание производных от  
него всех других ваших исключений.**

**Ниже показано, как создать собственную иерархию исключений для всех  
исключений в модуле или пакете. Первый шаг состоит в объявлении ба-  
зового класса, от которого наследуют все конкретные ошибки:**

**class BaseValidationError(^alueError):  
pass**

**Далее, все наши «реальные» классы ошибок могут быть сделаны произ-  
водными от базового класса ошибок. В результате мы получаем хорошую  
и чистую иерархию исключений, приложив лишь незначительные допол-  
нительные усилия:**

**class NameTooShortError(BaseValidationError):  
pass**

**class NameTooLongError(BaseValidationError):  
pass**

**class NameTooCuteError(BaseValidationError):  
pass**

**Например, это позволяет пользователям вашего пакета писать инструк-  
ции try-except, которые могут обработать все ошибки, возникающие в ре-  
зультате работы этого пакета, без необходимости отлавливать их вручную:**

**try:**

**validate(name)**

**except BaseValidationError as err:  
handle\_validation\_error(err)**

**Люди по-прежнему могут отлавливать более конкретные виды исключений  
этим способом, но если они этого не хотят, то, по крайней мере им не при-  
дется прибегать к захватыванию всех исключений при помощи всеобъем-  
лющей инструкции except. Обычно такой подход считается антишаблоном  
проектирования — он может негласно поглотить и скрыть разрозненные  
ошибки и сделать ваши программы намного труднее для отладки.**

**Разумеется, вы можете развить эту идею и логически сгруппировать ис-  
ключения в подробнейшие субиерархии. Но будьте осторожны — можно  
очень легко внести ненужную сложность, переборщив с этой работой.**

**Подводя итоги, следует отметить, что определение собственных классов-  
исключений облегчает принятие вашими пользователями стиля про-  
граммирования «Легче попросить прощения, чем разрешения» (EAFP),  
который считается более питоновским.**

**Ключевые выводы**

* **Определение ваших собственных типов исключений позволяет яснее  
  сформулировать замысел вашего программного кода и облегчить его  
  отладку.**
* **Следует делать свои собственные исключения производными от встро-  
  енного в Python класса Exception или от более конкретных классов-  
  исключений, таких как ValueError или KeyError.**

**□ Для определения логически сгруппированных иерархий исключений  
можно использовать наследование.**

1. **. Клонирование объектов  
   для дела и веселья**

**В Python инструкции присваивания не создают копии объектов, они лишь  
привязывают имена к объекту. Для неизменяемых объектов этот факт  
обычно не имеет значения.**

**Но для работы с изменяемыми объектами или коллекциями изменяемых  
объектов вам, возможно, стоит найти способ создания «реальных копий»,  
или «клонов», этих объектов.**

**По существу, вам иногда будут требоваться копии, которые можно моди-  
фицировать без автоматической модификации оригинала. В этом разделе  
я кратко представлю то, как копировать, или «клонировать», объекты  
в Python, и покажу связанные с этим подводные камни.**

**Начнем с того, что обратимся к копированию встроенных в Python кол-  
лекций. Встроенные в Python изменяемые коллекции, такие как списки,  
словари и множества, могут быть скопированы путем вызова своих фа-  
бричных функций с существующей коллекцией в качестве аргумента:**

**new\_list = list(original\_list)  
new\_dict = dict(original\_dict)  
new\_set = set(original\_set)**

**Однако этот метод не будет работать с собственными объектами и, вдо-  
бавок ко всему, он создает только мелкие копии. Для составных объектов,  
таких как списки, словари и множества, между мелким и глубоким копи-  
рованием имеется важное различие.**

**Мелкая копия (shallow copy) означает конструирование нового объекта-  
коллекции и затем его заполнение ссылками на дочерние объекты, най-  
денные в оригинале. В сущности, мелкая копия имеет всего один уровень  
в глубину. Процесс копирования выполняется нерекурсивно и поэтому не  
создает копий самих дочерних объектов.**

**Глубокая копия (deep copy) выполняет процесс копирования рекурсивно.  
Это означает конструирование сначала нового объекта коллекции, а за-  
тем рекурсивное его заполнение копиями дочерних объектов, найденных  
в оригинале. При копировании объекта таким способом выполняется об-  
ход всего дерева объектов целиком, и создается полностью независимый  
клон исходного объекта и всех его потомков.**

**Понимаю, что это была довольно заумная тирада. Поэтому обратимся  
к нескольким примерам, которые доведут до сознания разницу между  
глубокими и мелкими копиями.**

**Создание мелких копий**

**В приведенном ниже примере мы создадим новый вложенный список  
и затем мелко его скопируем при помощи фабричной функции** list()**:**

**>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**>>> ys = list(xs) *# Сделать мелкую копию***

**Это означает, что список** ys **теперь будет новым и независимым объектом  
с тем же самым содержимым, что и список** xs**. Это можно проверить, про-  
инспектировав оба объекта:**

**>>> xs**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**>>> ys**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**Чтобы подтвердить, что список** ys **действительно независим от ориги-  
нала, давайте разработаем маленький эксперимент. Можно попробовать  
добавить новый подсписок в оригинал (**xs**) и затем убедиться, что эта  
модификация не затронула копию (**ys**):  
>>> xs.append(['новый подсписок'])**

**>>> xs**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['новый подсписок']]  
>>> ys**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**Как видите, эффект был ожидаем. С изменением скопированного списка  
на «поверхностном» уровне никаких проблем не возникло.**

**Однако поскольку мы создали лишь мелкую копию оригинального спи-  
ска, список ys по-прежнему содержит ссылки на оригинальные дочерние  
объекты, хранящиеся в xs.**

**Эти дочерние элементы не были скопированы. Все свелось к тому, что  
в скопированном списке на них снова содержатся ссылки.**

**Поэтому, когда вы модифицируете один из дочерних объектов в спи-  
ске xs, эта модификация также будет отражена в списке ys — таким  
образом, оба списка совместно используют одинаковые дочерние объек-  
ты. Эта копия представляет собой всего лишь мелкую копию с одним  
уровнем в глубину:**

**>>> xs[1][0] = 'X'**

**>>> xs**

**[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['новый подсписок']]**

**>>> ys**

**[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]**

**В примере выше мы (казалось бы) изменили только список xs. Но ока-  
зывается, что в индексе 1 списков xs и ys были изменены оба подсписка.  
Опять-таки, это произошло, потому что мы создали всего-навсего мелкую  
копию оригинального списка.**

**Если бы на первом шаге мы создали глубокую копию списка xs, то оба объ-  
екта были бы полностью независимы. В этом и заключается практическая  
разница между мелкими и глубокими копиями объектов.**

**Теперь вы знаете, как создавать мелкие копии некоторых встроенных  
классов коллекций, и знаете разницу между мелким и глубоким копиро-  
ванием. Вопросы, ответы на которые мы по-прежнему хотим получить,  
следующие:**

* **Как создавать глубокие копии встроенных коллекций?**
* **Как создавать копии (мелкие и глубокие) произвольных объектов,  
  включая собственные классы?**

**Ответы на эти вопросы лежат в модуле** copy **стандартной библиотеки  
Python. Этот модуль обеспечивает простой интерфейс для создания мел-  
ких и глубоких копий произвольных объектов Python.**

**Создание глубоких копий**

**Давайте повторим предыдущий пример с копированием списка, но с од-  
ним важным различием. В этот раз мы собираемся создать глубокую  
копию, используя вместо встроенной фабричной функции функцию**deepcopy()**, определенную в модуле** copy**:**

**>>> import copy**

**>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**>>> zs = copy.deepcopy(xs)**

**Когда вы проинспектируете список** xs **и его клон** zs**, созданный нами  
с помощью** copy.deepcopy()**, вы увидите, что они оба снова выглядят  
идентичными— точно так же, как и в предыдущем примере:**

**>>> xs**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**>>> zs**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**Однако если вы внесете модификацию в один из дочерних объектов в ори-  
гинальном объекте (**xs**), то вы увидите, что эта модификация не затронет  
глубокую копию (**zs**).**

**Оба объекта, оригинал и копия, на этот раз полностью независимы. Спи-  
сок** xs **был клонирован рекурсивно, включая все его дочерние объекты:**

**>>> xs[1][0] = 'X'**

**>>> xs**

**[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]**

**>>> zs**

**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**Возможно, вам стоит сделать паузу, чтобы обратиться к интерпретатору  
Python и прямо сейчас выполнить все эти примеры. Усвоить копирование**объектов легче, когда вы имеете возможность набраться опыта и поэкспе-  
риментировать с примерами из первых рук.

**Между прочим, при помощи функции в модуле copy вы также можете  
создавать мелкие копии. Функция copy.copy() создает мелкие копии  
объектов.**

**Это полезно, если вам нужно четко сообщить, что где-то в своем про-  
граммном коде вы создаете мелкую копию. Использование copy.copy()  
позволяет указывать на этот факт. Однако что касается встроенных  
коллекций, то для создания их мелких копий более питоновским стилем  
будет считаться использование фабричных функций list, dict и set.**

**Копирование произвольных объектов**

**Вопрос, на который мы по-прежнему должны ответить, состоит в том, как  
создавать копии (мелкие и глубокие) произвольных объектов, включая  
собственные классы. Теперь давайте обратимся к этому вопросу.**

**И снова на выручку приходит модуль copy. Его функции copy.copy()  
и copy.deepcopy() могут использоваться для создания дубликата любого  
объекта.**

**И снова наилучший способ понять, как их использовать, — поставить  
простой эксперимент. Я собираюсь взять за основу предыдущий пример  
с копированием списка. Давайте начнем с определения простого класса  
двумерной точки:**

**class Point:**

**def init (self, x, y):**

**self.x = x  
self.y = y**

**def repr (self):**

**return f'Point({self.x!r}, {self.y!r})'**

**Надеюсь, вы согласитесь, что это было довольно прямолинейно. Я добавил  
реализацию \_\_repr\_\_(), с тем чтобы мы могли легко проинспектировать  
создаваемые на основе этого класса объекты в интерпретаторе Python.**

**Далее мы создадим экземпляр Point, а затем его (мелко) скопируем, ис-  
пользовав модуль copy:**

**>>> a = Point(23, 42)**

**>>> b = copy.copy(a)**

**Если проинспектировать содержимое оригинального объекта Point и его  
(мелкого) клона, то мы увидим то, что и ожидали:**

**>>> a**

**Point(23, 42)**

**>>> b**

**Point(23, 42)**

**>>> a is b  
False**

**Следует иметь в виду кое-что еще. Поскольку наш объект-точка для сво-  
их координат использует примитивные типы (целые числа), то в данном  
случае нет никакой разницы между мелкой и глубокой копией. Но я рас-  
ширю пример секунду спустя.**

**Теперь перейдем к более сложному примеру. Я собираюсь определить еще  
один класс, который будет представлять двумерные прямоугольники.  
Я сделаю это таким образом, который позволяет создавать более сложную  
иерархию объектов, — мои прямоугольники будут использовать объекты  
Point, представляющие их координаты:**

**class Rectangle:**

**def init (self, topleft, bottomright):**

**self.topleft = topleft  
self.bottomright = bottomright**

**def repr (self):**

**return (f'Rectangle({self.topleft!r},'  
f'{self.bottomright!r})')**

**Сначала мы попытаемся создать мелкую копию экземпляра Rectangle:**

**rect = Rectangle(Point(0, 1), Point(5, 6))  
srect = copy.copy(rect)**

**Если вы проинспектируете оригинальный прямоугольник и его копию,**

**то увидите, что переопределение метода repr () прекрасно сработало**

**и процесс мелкого копирования был выполнен, как мы и ждали:**

**>>> rect**

**Rectangle(Point(0, 1), Point(5, 6))**

**>>> srect**

**Rectangle(Point(0, 1), Point(5, 6))**

**>>> rect is srect  
False**

**Помните, как в предыдущем примере со списком иллюстрировалась  
разница между глубокими и мелкими копиями? Здесь я собираюсь при-  
менить тот же самый подход. Я изменю объект, находящийся глубоко  
в иерархии объектов, и затем вы вновь увидите, как это изменение будет  
отражено в (мелкой) копии:**

**>>> rect.topleft.x = 999  
>>> rect**

**Rectangle(Point(999, 1), Point(5, 6))**

**>>> srect**

**Rectangle(Point(999, 1), Point(5, 6))**

**Надеюсь, что этот пример показал то, что вы ожидали. Далее, я создам  
глубокую копию оригинального прямоугольника. Затем внесу в нее одно  
изменение, и вы увидите, какие объекты были затронуты:**

**>>> drect = copy.deepcopy(srect)**

**>>> drect.topleft.x = 222  
>>> drect**

**Rectangle(Point(222, 1), Point(5, 6))**

**>>> rect**

**Rectangle(Point(999, 1), Point(5, 6))**

**>>> srect**

**Rectangle(Point(999, 1), Point(5, 6))**

**Вуаля! На этот раз глубокая копия (drect) полностью независима от  
оригинала (rect) и мелкой копии (srect).**

**В этом разделе мы рассмотрели многие вопросы, и при этом остались еще  
некоторые тонкости, связанные с копированием объектов.**

**Эта тема стоит того, чтобы в нее углубиться (еще бы!), поэтому, возмож-  
но, вам стоит плотнее заняться документацией модуля** copy**1. Например,  
объекты могут управлять тем, как они копируются, если в них определить  
специальные методы** copy () **и** deepcopy ()**. Приятного времяпре-**

**провождения!**

**Ключевые выводы**

* **В результате создания мелкой копии объекта дочерние объекты не  
  клонируются. По этой причине результирующая копия не является  
  полностью независимой от оригинала.**
* **В процессе глубокого копирования объекта дочерние объекты клони-  
  руются рекурсивно. Клон полностью независим от оригинала, но на  
  создание глубокой копии уходит больше времени.**
* **При помощи модуля** copy **вы можете копировать произвольные объ-  
  екты (включая собственные классы).**

1. **. Абстрактные базовые классы держат  
   наследование под контролем**

**Абстрактные классы (АК), иногда также называемые абстрактными ба-  
зовыми классами, гарантируют, что производные классы реализуют те или  
иные методы базового класса. В этом разделе вы узнаете о преимуществах  
абстрактных классов и о том, как их определять при помощи встроенного  
в Python модуля** abc**.**

**Итак, в чем же прелесть абстрактных классов? Не так давно у меня на ра-  
боте был спор о том, какой шаблон использовать для реализации удобной  
в сопровождении иерархии классов в Python. Точнее говоря, цель состо-  
яла в том, чтобы определить простую иерархию классов для сервисного  
бэкенда самым благоприятным для программиста и удобным в сопровож-  
дении способом. [[23]](#footnote-23)**

**У нас был класс BaseService, который определял общий интерфейс и не-  
сколько конкретных реализаций. Конкретные реализации делают разные  
вещи, но все они обеспечивают тот же самый интерфейс (MockService,  
RealService и т. д.). Чтобы более четко проявить взаимосвязи, все кон-  
кретные реализации были производными от класса BaseService.**

**Чтобы сделать этот программный код максимально удобным в обслужи-  
вании и благоприятным для программиста, мы хотели удостовериться, что**

* **создание экземпляров базового класса невозможно,**
* **упущение из виду реализации методов интерфейса в одном из под-  
  классов вызывает ошибку на ранней стадии.**

**Итак, почему же может возникнуть потребность в использовании моду-  
ля Python abc для решения этой задачи? Названная выше конструкция  
довольно распространена в более сложных системах. Чтобы обеспечить  
реализацию ряда методов базового класса производным классом, как  
правило, используется примерно такая идиома Python:**

**class Base:**

**def foo(self):**

**raise *NotImpLementedError()***

**def bar(self):**

**raise *NotImpLementedError()***

**class Concrete(Base):  
def foo(self):**

**return 'вызвана foo()'**

* ***О нет, мы забыли переопределить bar()...***
* ***def bar(self):***
* ***return "вызвана bar()"***

**Итак, что же мы получаем из этой первой попытки решения задачи?  
Вызов методов экземпляра Base правильно вызывает исключения  
NotImplementedError:**>>> b = Base()

**>>> b.foo()  
*NotImpLementedError***

**Более того, и создание экземпляра, и использование Concrete работают  
так, как ожидалось. И если вызвать не реализованный в нем метод, такой  
как bar(), то в результате тоже будет вызвано исключение:**

**>>> c = ConcreteQ  
>>> c.foo()**

***'вызвана foo()'***

**>>> c.bar()**

***NotImpLementedError***

**Эта первая реализация выглядит неплохо, но пока не идеально. Ее обо-  
ротными сторонами является то, что мы по-прежнему можем**

* **легко создавать экземпляры Base, не получая ошибку, а также**
* **обеспечивать неполные подклассы — создание экземпляра Concrete не  
  будет вызывать ошибку до тех пор, пока мы не вызовем отсутствующий  
  метод bar().**

**При помощи модуля Python abc, который был добавлен в Python 2.6[[24]](#footnote-24), мы  
можем добиться большего успеха и решить эти оставшиеся проблемы. Вот  
обновленная реализация с использованием абстрактного класса, опреде-  
ленного в модуле abc:**

**from abc import ABCMeta, abstractmethod**

**class Base(metaclass=ABCMeta):**

**@abstractmethod  
def foo(self):  
pass**

**@abstractmethod  
def bar(self):  
pass**

**class Concrete(Base):  
def foo(self):  
pass**

***# Мы снова забыли объявить bar()...***

**Этот фрагмент кода по-прежнему ведет себя так, как нужно, и создает  
правильную иерархию классов:**

**assert issubclass(Concrete, Base)**

**С другой стороны, мы здесь получаем еще одно преимущество. Подклас-  
сы** Base **вызывают исключение** TypeError **во время создания экземпляра  
всякий раз, когда мы забываем реализовать какие-либо абстрактные  
методы. Вызванное исключение говорит о том, какой метод или методы  
отсутствуют:**

**>>> c = Concrete()**

**TypeError:**

***"Can't instantiate abstract class Concrete with abstract methods bar"***

**Без модуля** abc **мы получали бы только исключение** NotImplementedError  
**в случае фактического вызова отсутствующего метода. Возможность  
получать уведомления об отсутствующих методах во время создания эк-  
земпляра является большим преимуществом. В результате написание не-  
допустимых подклассов в значительной степени блокируется. Возможно,  
этот факт не сыграет какой-то особой роли, если вы пишете новый код, но  
обещаю, что спустя несколько недель или месяцев он станет полезным.**

**Безусловно, этот шаблон не является полной заменой проверки типов во  
время компиляции. Однако я обнаружил, что он часто делает иерархии  
классов более прочными и более удобными в сопровождении. Использо-  
вание абстрактных классов позволяет программисту четче формулировать  
свой замысел и таким образом делает код более коммуникативным. Я ре-  
комендую вам почитать документацию по модулю** abc **и присмотреться  
к ситуациям, где применение этого шаблона имеет смысл.**

**Ключевые выводы**

* **Абстрактные классы (АК) следят за тем, чтобы производные классы  
  реализовывали те или иные методы базового класса во время создания  
  экземпляра.**
* **Применение АК помогает избежать ошибок и сделать иерархии клас-  
  сов более легкими в сопровождении.**

1. **. Чем полезны именованные кортежи**

**Python поставляется со специализированным контейнерным типом  
namedtuple, то есть именованным кортежем. И этот тип, по всей видимо-  
сти, не привлекает того внимания, которое он заслуживает. Именованный  
кортеж представляет собой одно из тех удивительных функциональных  
средств языка Python, которое спрятано у всех на виду.**

**Именованные кортежи могут быть отличной альтернативой определению  
класса вручную, и у них есть некоторые другие интересные свойства,  
с которыми я хочу вас познакомить в этом разделе.**

**Итак, что же такое именованный кортеж и в чем проявляется его ис-  
ключительность? Именованные кортежи лучше всего представить как  
расширение встроенного типа данных tuple.**

**Кортежи Python — это простая структура данных, предназначенная для  
группирования произвольных объектов. Кроме того, кортежи не могут  
изменяться — после их создания их нельзя изменять. Ниже приведен  
короткий пример:**

**>>> tup = ('привет', object(), 42)**

**>>> tup**

**('привет', <object object at 0x105e76b70>, 42)**

**>>> tup[2]**

**42**

**>>> tup[2] = 23  
TypeError:**

***"'tuple' object does not support item assignment"***

**Оборотной стороной простых кортежей является то, что данные, которые  
вы в них храните, могут быть извлечены только адресацией посредством  
целочисленных индексов. Вы не можете назначать имена отдельным свой-  
ствам, хранящимся в кортеже. А это может повлиять на удобочитаемость  
программного кода.**

**Кроме того, кортеж всегда является вспомогательной структурой. Трудно  
гарантировать, что у двух кортежей будет одно и то же количество полей  
и одинаковые хранящиеся в них свойства. В результате появляется воз-  
можность беспрепятственно вносить ошибки «по недоразумению», просто  
перепутав порядок следования полей.**

**Именованные кортежи спешат на помощь**

**Именованные кортежи призваны решать эти две проблемы.**

**В первую очередь именованные кортежи являются неизменяемыми  
контейнерами, точно так же, как обычные кортежи. Поместив данные  
в атрибут верхнего уровня в именованном кортеже, вы не сможете его  
модифицировать путем обновления этого атрибута. Все атрибуты объекта**namedtuple **подчиняются принципу «однократная запись, многократное  
чтение».**

**Помимо этого, контейнеры типа** namedtuple **являются, скажем так, име-  
нованными кортежами (named tuples). Доступ к каждому хранящемуся  
в них объекту можно получить через уникальный (человекочитаемый)  
идентификатор. Это свойство освобождает вас от необходимости запоми-  
нать целочисленные индексы или обращаться к искусственным приемам,  
таким как определение целочисленных констант в качестве мнемокодов  
ваших индексов.**

**Вот пример того, как выглядит именованный кортеж:**

**>>> from collections import namedtuple  
>>> Car = namedtuple('ABTo' , 'цвет пробег')**

**Именованные кортежи были добавлены в стандартную библиотеку  
Python версии 2.6. Чтобы ими воспользоваться, необходимо импорти-  
ровать модуль** collections**. В приведенном выше примере я определил  
простой тип данных** Car **с двумя полями:** цвет **и** пробег**.**

**Вам, возможно, любопытно, почему в этом примере я передаю фабрич-  
ной функции** namedtuple **строковое значение** 'Авто' **в качестве первого  
аргумента.**

**В документации Python этот параметр упоминается как «имя типа». Он  
является именем нового класса, который создается в результате вызова  
функции** namedtuple**.**

**Поскольку функция** namedtuple **не может знать о том, каким является имя  
переменной, которой мы назначаем результирующий класс, мы должны  
сообщить ему явным образом, какое имя класса мы хотим использовать.**

**Имя класса используется в строке документации docstring и в реализа-  
ции метода repr , которые функция namedtuple генерирует для нас**

**автоматически.**

**В этом примере есть и другая синтаксическая диковинка — почему мы  
передаем поля в виде строки, в которой их имена закодированы как 'цвет  
пробег'?**

**Ответ заключается в том, что фабричная функция namedtuple вызывает  
функцию split() со строковым значением, содержащим имена полей,  
которая преобразовывает его в список имен полей. Так что в действитель-  
ности это просто сокращенная запись для приведенных ниже двух шагов:**

**>>> 'цвет npo6er'.splitQ  
['цвет', 'пробег']**

**>>> Car = namedtuple('ABTo', ['цвет', 'пробег'])**

**Разумеется, вы также можете непосредственно передать список со стро-  
ковыми именами полей, если вы предпочитаете, чтобы это выглядело  
именно так. Преимущество от использования списка как такового состоит  
в том, что этот код легче переформатировать, если есть необходимость  
разбить его на несколько строк кода:**

**>>> Car = namedtuple('Автo', [**

**... 'цвет',**

**... 'пробег',**

**... ])**

**Что бы вы ни решили, теперь при помощи фабричной функции Car вы  
можете создавать новые объекты «car». Поведение будет таким же, как  
если бы вы создали класс Car вручную и определили в нем конструктор,  
принимающий значения «цвет» и «пробег»:**

**>>> my\_car = Car('красный', 3812.4)**

**>>> my\_car.цвeт  
'красный'**

**>>> my\_car.прoбeг**

**3812.4**

**Помимо получения доступа к значениям, хранящимся в именованном  
кортеже, по их идентификаторам, вы по-прежнему можете получать к ним  
доступ по их индексу. Благодаря этому именованные кортежи могут ис-  
пользоваться в качестве прямой замены обычным кортежам:**

**>>> my\_car[0]**

***'red'***

**>>> tuple(my\_car)**

**('красный', 3812.4)**

**Распаковка кортежа и оператор \* для распаковки аргументов функции  
по-прежнему работают как надо:**

**>>> color, mileage = my\_car  
>>> print(color, mileage)  
красный 3812.4  
>>> print(\*my\_car)  
красный 3812.4**

**К тому же в качестве бесплатного приложения вы получите приличное  
строковое представление своего объекта namedtuple, что позволит наби-  
рать чуть меньше текста и сэкономит на многословности:**

**пробег=3812.4)**

**>>> my\_car  
Авто(цвет='красный'**

**Подобно кортежам, именованные кортежи не изменяются. Когда вы  
попытаетесь переписать одно из их полей, вы получите исключение  
AttributeError:**

**>>> ту\_саг.цвет = 'синий'  
AttributeError: "can't set attribute"**

**На внутреннем уровне объекты namedtuple реализованы как обычные  
классы Python. В том, что касается использования оперативной памяти,  
они тоже «лучше» обычных классов и так же эффективны с точки зрения  
потребления оперативной памяти, как и обычные кортежи.**

В Python именованные кортежи неплохо рассматривать как ***эффективную  
с точки зрения потребления оперативной памяти краткую форму для  
определения неизменяющегося класса вручную.***

**Создание производных от Namedtuple подклассов**

**Поскольку объекты** namedtuple **строятся поверх обычных классов Python,  
вы даже можете добавлять в них методы. Например, подобно любому  
другому классу, вы можете расширить класс** namedtuple **и таким образом  
добавить в него методы и новые свойства. Приведем пример:**

**Car = namedtuple('ABTo', 'цвет пробег')  
class MyCarWithMethods(Car):  
def hexcolor(self):**

**if self.цвет == 'красный':  
return '#ff0000'  
else:**

**return '#000000'**

**Теперь можно создавать объекты** MyCarWithMethods **и, следовательно, вы-  
зывать их метод** hexcolor()**:**

**>>> c = MyCarWithMethods('красный', 1234)**

**>>> c.hexcolor()**

**'#ff0000'**

**Вместе с тем выглядеть это может слегка неуклюжим. По-видимому,  
такая возможность пригодится, если вам нужен класс с неизменяемыми  
свойствами, но здесь легко и в ногу себе выстрелить.**

**Например, в добавлении нового неизменяемого поля (immutable field)  
есть свои сложности из-за внутренней структуры именованных кортежей.  
Самый легкий способ создать иерархии именованных кортежей — ис-  
пользовать свойства** \_fields **базового кортежа:**

**>>> Car = namedtuple('Авто', 'цвет пробег')**

**>>> ElectricCar = namedtuple(**

**... 'ЭлектрическоеАвто', Car.\_fields + ('заряд',))**

**Это дает желаемый результат:  
>>> ElectricCar('красный', 1234, 45.0)**

**ЭлектрическоеАвто(цвет='красный', пробег=1234, заряд=45.0)**

**Встроенные вспомогательные методы**

**Помимо свойства** \_fields**, каждый экземпляр именованного кортежа так-  
же предлагает еще несколько вспомогательных методов, которые могли  
бы быть вам полезны. Все их имена начинаются с одинарного символа  
подчеркивания (**\_**), который обычно сигнализирует о том, что метод или  
свойство являются «приватными» и не являются частью стабильного  
публичного интерфейса класса или модуля.**

**Правда, в случае с именованными кортежами согласованное правило  
именования с символом подчеркивания несет в себе другой смысл. Эти  
вспомогательные методы и свойства являются составной частью публич-  
ного интерфейса класса** namedtuple**. Вспомогательные методы получают  
такие имена, чтобы избежать конфликтов имен с определяемыми пользо-  
вателями полями кортежей. Так что можете спокойно ими пользоваться,  
если они вам нужны!**

**Хочу показать вам несколько сценариев, где могли бы пригодиться вспо-  
могательные методы именованного кортежа. Давайте начнем со вспомо-  
гательного метода** \_asdict()**. Он возвращает содержимое именованного  
кортежа в виде словаря:**

**>>> my\_car.\_asdictQ**

**OrderedDict([('цвет', 'красный'), ('пробег', 3812.4)])**

**Этот метод очень полезен для предотвращения опечаток в именах полей  
во время генерирования результата в формате JSON, например:**

**>>> json.dumps(my\_car.\_asdictQ, ensure\_ascii=False)**

**# False для кириллицы**

***'{"цвет": "красный", "пробег": 3812.4}'***

**Метод** \_replace() **— это еще один полезный вспомогательный метод. Он  
создает (мелкую) копию кортежа и позволяет вам выборочно заменять  
некоторые его поля:**

**>>> my\_car.\_replace(цвет='синий')**

**Авто(цвет='синий', пробег=3812.4)**

**Наконец, метод класса** \_make() **может использоваться для создания новых  
экземпляров класса** namedtuple **из (итерируемой) последовательности:**

**>>> Саг.\_таке(['красный', 999])  
Авто(со1ог='красный', пробег=999)**

**Когда использовать именованные кортежи**

**Именованные кортежи могут оказаться простым средством для приве-  
дения в порядок исходного кода, и они могут сделать код более удобо-  
читаемым, обеспечив вашим данным наиболее совершенную структуру.**

**Например, на моем опыте переход от ситуативных типов данных, таких  
как словари с фиксированным форматом, к именованным кортежам по-  
могает яснее выражать свои замыслы. Нередко, когда я предпринимаю эту  
рефакторизацию, я каким-то невообразимым образом прихожу к более  
совершенному решению проблемы, с которой я сталкиваюсь.**

**Использование именованных кортежей поверх неструктурированных,  
а также использование словарей может облегчить жизнь моих коллег,  
потому что именованные кортежи позволяют раздавать данные в «само-  
документированном» виде (в известной степени).**

**С другой стороны, я стараюсь не использовать именованные кортежи  
ради них самих, если они не помогают мне писать «более чистый» и бо-  
лее удобный в сопровождении исходный код. Как и в отношении многих  
других методов, приводимых в настоящей книге, иногда может оказаться  
слишком много хорошего (что, как известно, тоже плохо).**

**Тем не менее если именованные кортежи использовать с осторожностью,  
они, несомненно, могут сделать ваш программный код Python лучше  
и выразительнее.**

**Ключевые выводы**

* **В языке Python** collection. namedtuple **является эффективной с точки  
  зрения потребляемой оперативной памяти краткой формой для опре-  
  деления неизменяющегося класса вручную.**
* **Именованные кортежи помогут почистить ваш исходный код, обес-  
  печив вашим данным более доступную для понимания структуру.**

**□ Именованные кортежи обеспечивают несколько полезных вспомо-  
гательных методов, которые начинаются с одинарного символа под-  
черкивания, но при этом являются составной частью публичного  
интерфейса. Вполне нормально их использовать.**

1. **Переменные класса против переменных  
   экземпляра: подводные камни**

**Помимо проведения различия между методами класса и методами экзем-  
пляра, объектная модель Python также проводит различие между пере-  
менными класса и переменными экземпляра.**

**Это различие имеет большое значение. Мне, как начинающему раз-  
работчику на Python, оно также доставляло немало хлопот. В течение  
длительного времени я не мог найти время, чтобы разобраться в этих по-  
нятиях с самых азов. И поэтому мои первые эксперименты с ООП были  
пронизаны удивительными линиями поведения и странными ошибками.  
В этом разделе мы устраним путаницу относительно этой темы при по-  
мощи нескольких практических примеров.**

**Как я уже сказал, в объектах Python объявляются два вида атрибутов  
данных: переменные класса (class variables) и переменные экземпляра  
(instance variables).**

Переменные класса **объявляются внутри определения класса (но  
за пределами любых методов экземпляра). Они не привязаны ни к од-  
ному конкретному экземпляру класса. Вместо этого переменные клас-  
са хранят свое содержимое в самом классе, и все объекты, созданные  
на основе того или иного класса, предоставляют общий доступ к одина-  
ковому набору переменных класса. Например, это означает, что моди-  
фикация переменной класса одновременно затрагивает все экземпляры  
объекта.**

Переменные экземпляра **всегда привязаны к конкретному экземпляру  
объекта. Их содержимое хранится не в классе, а в каждом отдельном  
объекте, созданном на основе класса. По этой причине содержимое пере-  
менной экземпляра абсолютно независимо от одного экземпляра объекта**к другому. И поэтому модификация переменной экземпляра одновремен-  
но затрагивает только один экземпляр объекта.

**Ладно, все это было довольно абстрактно — самое время рассмотреть  
немного исходного кода! Давайте потренируемся на собачках^ В обу-  
чающих пособиях, посвященных ООП, для иллюстрации этого тезиса  
всегда используются автомобили или домашние животные, и мне сложно  
отказаться от этой традиции.**

**Что собаке для счастья нужно? Правильно! Четыре лапы да имя:  
class Dog:**

**num\_legs = 4 *# <- Переменная класса***

**def init (self, name):**

**self.name = name *# <- Переменная экземпляра***

**О’кей. У нас есть изящное объектно-ориентированное представление  
ситуации с собакой, которую я только что описал. Создание новых экзем-  
пляров Dog работает, как и ожидалось, и каждый из них получает пере-  
менную экземпляра с именем name:**

**>>> jack = Dog('Джек')**

**>>> jill = Dog('Джилл')**

**>>> jack.name, jill.name  
('Джек', 'Джилл')**

**Во всем, что касается переменных класса, всегда есть чуть больше гибко-  
сти. Доступ к переменной класса num\_legs можно получить либо непо-  
средственно в каждом экземпляре Dog, либо в самом классе:**

**>>> jack.num\_legs, jill.num\_legs  
(4, 4)**

**>>> Dog.num\_legs  
4**

**Однако попытка получить доступ к переменной экземпляра через класс  
потерпит неудачу с исключением AttributeError. Переменные экземпля-  
ра характерны для каждого экземпляра объекта и создаются, когда выпол-  
няется конструктор init — они даже не существуют в самом классе.**

**В этом заключается ключевое различие между переменными класса  
и переменными экземпляра:**

**>>> Dog.name  
*AttributeError:***

**"type object 'Dog' has no attribute 'name'"**

**Ладно, пока все идет неплохо.**

**Допустим, в один прекрасный день пес по кличке Джек поедал свой ужин,  
расположившись слишком близко от микроволновки, в результате у него  
выросла дополнительная пара лап. Как бы вы представили этот факт в не-  
большой «песочнице» с исходным кодом, которая у нас сейчас есть?**

**Первая идея**Dog**:**

**просто модифицировать переменную num\_legs в классе**

**>>> Dog.num\_legs = 6**

**Но помните, мы не хотим, чтобы все собаки стали носиться вокруг о шести  
лапах. Итак, сейчас мы только что превратили каждую собаку в нашей ми-  
кровселенной в сверхсобаку, потому что мы модифицировали переменную  
класса. Это затрагивает всех собак, даже тех, которые были созданы ранее:**

**>>> jack.num\_legs, jill.num\_legs  
(6, 6)**

**Этот вариант не сработал. А не сработал он потому, что модификация  
переменной класса в пространстве имен класса затрагивает все экзем-  
пляры класса. Давайте отыграем это изменение в переменной класса  
назад и вместо этого попробуем дать дополнительную пару лап только  
конкретному псу Джеку:**

**>>> Dog.num\_legs = 4  
>>> jack.num\_legs = 6**

**Так, и что за чудовище мы получили? Сейчас выясним:**

**>>> jack.num\_legs, jill.num\_legs, Dog.num\_legs  
(6, 4, 4)**

**Ладно. Выглядит «довольно неплохо» (ну, кроме того, конечно, что мы  
прямо сейчас дали бедному псу несколько лишних лап). Но как это из-  
менение на самом деле повлияло на наши объекты** Dog**?**

**А проблема, как выясняется, здесь в следующем: несмотря на то что мы  
получили желаемый результат (лишние лапы для Джека), мы внесли  
переменную экземпляра** num\_legs **в экземпляр с псом по кличке Джек.  
И теперь новая переменная экземпляра** num\_legs **«оттеняет» переменную  
класса с тем же самым именем, переопределяя и скрывая ее, когда мы об-  
ращаемся к области действия экземпляра:**

**>>> jack.num\_legs, jack. class .num\_legs**

**(б, 4)**

**Как вы видите, переменные класса, казалось бы, стали несогласованными.  
Это произошло потому, что внесение изменения в** jack.num\_legs **созда-  
ло переменную экземпляра с тем же самым именем, что и у переменной  
класса.**

**Это не всегда плохо, но важно понимать, что именно здесь произошло.  
Прежде чем я наконец-то разобрался в области действия уровня класса  
и уровня экземпляра в Python, это было широкими воротами, через кото-  
рые в мои программы то и дело закрадывались ошибки.**

**Сказать по правде, попытка модифицировать переменную класса через  
экземпляр объекта, который затем непредумышленно создает переменную  
экземпляра с тем же именем, затеняя оригинальную переменную класса,  
является в Python чем-то вроде подводного камня ООП.**

**Пример без собак**

**Хотя в процессе написания этого раздела книги ни одна собака не по-  
страдала (это все шутки и игры до тех пор, пока кто-то не вырастит себе  
лишнюю пару лап), я хочу дать вам еще один практический пример  
полезных штук, которые вы можете сделать с переменными класса. То,  
что будет немного ближе к реальным приложениям с переменными  
класса.**

**Итак, вот этот пример. Приведенный ниже класс** CountedObject **отсле-  
живает, сколько раз он использовался для создания экземпляров на про-  
тяжении жизни программы (что на деле может обеспечить интересный  
метрический показатель производительности):**

**class CountedObject:  
num\_instances = 0**

**def init (self):**

**self. class .num\_instances += 1**

**Класс** CountedObject **содержит переменную класса** num\_instances**, которая  
служит в качестве общего счетчика. Когда класс объявлен, он инициали-  
зирует счетчик нулем, а затем оставляет его в покое.**

**Всякий раз, когда вы создаете новый экземпляр этого класса, он увели-  
чивает общий счетчик на единицу во время выполнения конструктора** init **:**

**>>> CountedObject.num\_instances  
0**

**>>> CountedObjectQ.num\_instances  
1**

**>>> CountedObjectQ.num\_instances  
2**

**>>> CountedObjectQ.num\_instances  
3**

**>>> CountedObject.num\_instances  
3**

**Обратите внимание, как этот фрагмент кода должен проскакивать через  
небольшой обруч, чтобы обеспечить увеличение переменной счетчика  
в классе. Легко можно было бы сделать ошибку, если бы я написал кон-  
структор следующим образом:  
# ПРЕДУПРЕЖДЕНИЕ: Эта реализация содержит ошибку**

**class BuggyCountedObject:  
num\_instances = 0**

**def \_\_init\_\_(self):**

**self.num instances += 1 # !!!**

**Как вы увидите, эта (плохая) реализация никогда не будет увеличивать  
общую переменную счетчика:**

**>>> BuggyCountedObject.num\_instances  
0**

**>>> BuggyCountedObject().num\_instances  
1**

**>>> BuggyCountedObject().num\_instances  
1**

**>>> BuggyCountedObject().num\_instances  
1**

**>>> BuggyCountedObject.num\_instances  
0**

**Уверен, что вы увидели, где я допустил промах. Эта (ошибочная) реализа-  
ция не увеличивает общий счетчик, потому что я сделал ошибку, которую  
объяснил в предыдущем примере с псом Джеком. Эта реализация не будет  
работать, потому что я непредумышленно «затенил» переменную класса**num\_instance**, создав в конструкторе переменную экземпляра с тем же  
именем.**

**Она правильно вычисляет новое значение счетчика (перейдя от 0 к 1),  
но затем сохраняет результат в переменной экземпляра, а это означает,  
что другие экземпляры класса никогда не увидят обновленное значение  
счетчика.**

**Как вы видите, допустить эту ошибку очень легко. Во время работы с раз-  
деляемым состоянием в классе следует быть осторожным и перепроверять  
области действия. Автоматизированные тесты и контроль качества работы  
со стороны коллег существенно помогают в этом.**

**Однако надеюсь, что вы видите, почему и как переменные класса (несмо-  
тря на их подводные камни) могут оказаться полезными инструментами  
на практике. Удачи!**

**Ключевые выводы**

**□ Переменные класса предназначены для данных, совместно используе-  
мых всеми экземплярами класса. Они принадлежат именно классу, а не**

**конкретному экземпляру и являются общими для всех экземпляров  
класса.**

* **Переменные экземпляра предназначены для данных, которые уникаль-  
  ны для каждого экземпляра. Они принадлежат отдельным экземплярам  
  объекта и не являются общими для других экземпляров класса. Каждая  
  переменная экземпляра получает уникальное резервное хранилище,  
  характерное для данного экземпляра.**
* **Поскольку переменные класса могут быть «затенены» переменными  
  экземпляра, имеющими одинаковое имя, можно легко (непреднаме-  
  ренно) переопределить переменные класса, в результате чего будут  
  внесены ошибки и создано странное поведение.**

**4.8 . Срыв покровов с методов экземпляра,  
методов класса и статических методов**

**В этой главе вы увидите, что именно в Python стоит за методами класса  
(class methods), статическими методами (static methods) и обычными  
методами экземпляра (instance methods).**

**Если вы разовьете интуитивное понимание их различий, то сможете писать  
объектно-ориентированный программный код Python, который яснее со-  
общает свой замысел и в конечном счете будет удобнее в сопровождении.**

**Давайте начнем с написания класса (Python 3), который содержит про-  
стые примеры всех трех типов методов:**

**class MyClass:**

**def method(self):**

**return 'вызван метод экземпляра', self**

**@classmethod**

**def classmethod(cls):**

**return 'вызван метод класса', cls**

**@staticmethod  
def staticmethod():**

**return 'вызван статический метод'**

**Примечание для пользователей Python 2: декораторы** @staticmethod **и** @  
classmethod **доступны, начиная с Python 2.4, и поэтому данный пример  
будет работать как есть. Вместо того чтобы использовать простое объявле-  
ние** class MyClass**, вы можете объявить класс в новом стиле, с наследова-  
нием от** object **с помощью синтаксической конструкции** MyClass(object)**.  
Но в остальном все в шоколаде!**

**Методы экземпляра**

**Первый метод в** MyClass **с именем** method **является обычным методом  
экземпляра. Это базовый, без наворотов, тип метода, который вы будете  
использовать большую часть времени. Вы видите, что этот метод прини-  
мает один параметр,** self**, который указывает на экземпляр класса** MyClass  
**во время вызова этого метода. Но, разумеется, методы экземпляра могут  
принимать более одного параметра.**

**Через параметр** self **методы экземпляра могут свободно получать доступ  
к атрибутам и другим методам в том же самом объекте. Это придает им  
большую мощь в том, что касается модификации состояния объекта.**

**Методы экземпляра могут не только модифицировать состояние объекта,**

**но и получать доступ к самому классу через атрибут** self. class **. Это**

**означает, что методы экземпляра также могут модифицировать состояние  
класса.**

**Методы класса**

**Давайте сравним это со вторым методом,** MyClass.classmethod**. Я пометил  
этот метод декоратором** ^classmethod**1, чтобы обозначить его как метод  
класса.**

**Вместо параметра** self **методы класса принимают параметр** cls**, который  
указывает на класс, а не на экземпляр объекта во время вызова этого ме-  
тода. [[25]](#footnote-25)**

**Поскольку метод класса имеет доступ только к этому аргументу** cls**, он  
не может менять состояние экземпляра объекта. Для этого потребовался  
бы доступ к параметру** self**. Однако методы класса по-прежнему могут  
модифицировать состояние класса, которое применимо во всех экзем-  
плярах класса.**

**Статические методы**

**Третий метод,** MyClass.staticmethod**, был помечен декоратором** @sta-  
ticmethod**1, чтобы обозначить его как статический метод.**

**Этот тип метода не принимает ни параметр** self**, ни параметр** cls**, хотя,  
конечно же, он может быть сделан так, чтобы принимать произвольное  
количество других параметров.**

**Как результат, статический метод не может модифицировать состояние  
объекта или состояние класса. Статические методы ограничены теми  
данными, к которым они могут получить доступ, — они, прежде всего,  
являются средством организации пространства имен ваших методов.**

**Посмотрим на них в действии!**

**Я знаю, что пояснения были весьма теоретизированными до этого места.  
Более того, полагаю, что важно на практике развить интуитивное пони-  
мание того, как эти типы методов различаются. Именно поэтому теперь  
мы пробежимся по нескольким примерам.**

**Взглянем на то, как эти методы себя ведут в действии, когда мы их вы-  
зываем. Начнем с создания экземпляра класса, а затем вызовем три опре-  
деленных в нем разных метода.**

**Класс** MyClass **был создан так, чтобы реализация каждого метода воз-  
вращала кортеж, содержащий информацию, которую мы можем исполь-  
зовать, чтобы проследить, что происходит и к каким частям класса или  
объекта этот метод может получить доступ. [[26]](#footnote-26)**

**Вот что происходит, когда мы вызываем** метод экземпляра**:**

**>>> obj = MyClass()**

**>>> obj.method()**

**('вызван метод экземпляра', <MyClass instance at 0x11a2>)**

**Этот результат подтверждает, что в данном случае метод экземпляра  
с именем** method **имеет доступ к экземпляру объекта (напечатанному как**<MyClass instance>**) через аргумент** self**.**

**Во время вызова этого метода Python заменяет аргумент** self **на объект  
экземпляра,** obj**. Чтобы получить тот же самый результат, мы можем про-  
игнорировать синтаксический сахар, предоставляемый синтаксической  
конструкцией вызова с точкой,** obj.method()**, и передать объект экзем-  
пляра вручную:**

**>>> MyClass.method(obj)**

**('вызван метод экземпляра', <MyClass instance at 0x11a2>)**

**Между прочим, методы экземпляра могут также получать доступ непо-  
средственно к самому классу через атрибут** self. class **. Это делает**

**методы экземпляра мощными с точки зрения ограничений доступа — они  
могут свободно модифицировать состояние в экземпляре объекта и в са-  
мом классе.**

**Теперь давайте испытаем** метод класса**:**

**>>> obj.classmethod()**

**('вызван метод класса', <class MyClass at 0x11a2>)**

**Вызов метода** classmethod() **показал, что у него нет доступа к объекту**<MyClass instance>**, а есть только к объекту** <class MyClass>**, представ-  
ляющему сам класс (в Python все является объектом, даже сами классы).**

**Обратите внимание на то, как Python автоматически передает класс в ка-  
честве первого аргумента в функцию, когда мы вызываем метод** MyClass.  
classmethod()**. В Python такое поведение запускается вызовом метода  
через точечный синтаксис (dot syntax). Параметр** self **в методах экземп-  
ляра работает таким же образом.**

**Также обратите внимание на то, что обозначение этих параметров как self  
и cls является всего-навсего согласованным правилом именования. С тем  
же успехом вы можете назвать их the\_object и the\_class и получить тот  
же самый результат. Важно лишь то, что в списке параметров для этого  
конкретного метода они располагаются первыми.**

**Теперь самое время вызвать статический метод:**

**>>> obj.staticmethod()**

***'вызван статический метод'***

**Заметили, как мы вызвали метод staticmethod() объекта и смогли сделать  
это успешно? Некоторые разработчики удивляются, когда узнают, что  
статический метод можно вызывать на экземпляре объекта.**

**За кадром, когда статический метод вызывается с использованием то-  
чечного синтаксиса, Python просто накладывает ограничения доступа, не  
передавая аргумент self или cls.**

**Этим подтверждается, что статические методы не могут получить доступ  
ни к состоянию экземпляра объекта, ни к состоянию класса. Они работают  
как обычные функции, но при этом они принадлежат пространству имен  
класса (и каждого экземпляра).**

**Теперь давайте посмотрим, что произойдет при попытке вызвать эти ме-  
тоды на самом классе, не создавая экземпляр объекта заранее:**

**>>> MyClass.classmethod()**

**('вызван метод класса', <class MyClass at 0x11a2>)**

**>>> MyClass.staticmethod()**

***'вызван статический метод'***

**>>> MyClass.method()**

**TypeError: unbound method method() must**

**be called with MyClass instance as first  
argument (got nothing instead)"""**

**Мы нормально смогли вызвать classmethod() и staticmethod(), а вот  
попытка вызвать метод экземпляра method() не удалась с исключением  
TypeError.**

**Такого результата следовало ожидать. На этот раз мы не создали экземп-  
ляр объекта и попытались вызвать функцию экземпляра непосредственно  
на самом шаблоне класса. Иными словами, в Python нет способа запол-  
нить аргумент self, и поэтому данный вызов терпит неудачу с исключе-  
нием TypeError.**

**Это должно сделать различие между этими тремя типами методов чуть  
яснее. Но не переживайте, я не собираюсь останавливаться на этом. В сле-  
дующих двух разделах я пробегусь по двум немного более реалистичным  
примерам, которые покажут, когда использовать эти конкретные типы  
методов.**

**В своих примерах я буду исходить из этого элементарного класса Pizza:  
class Pizza:**

**def init (self, ingredients):**

**self.ingredients = ingredients**

**def repr (self):**

**return f'Pizza({self.ingredients!r})'**

**>>> Pizza(['сыр', 'помидоры'])**

**Pizza(['сыр', 'помидоры'])**

**Фабрики аппетитной пиццы с @classmethod**

**Если вы сталкивались с пиццей в реальном мире, то вы знаете, что суще-  
ствует много видов аппетитной пиццы:**

**Pizza(['моцарелла', 'помидоры'])**

**Pizza(['моцарелла', 'помидоры', 'ветчина', 'грибы'])**

**Pizza(['моцарелла'] \* 4)**

**Итальянцы придумали свою классификацию пицц несколько веков назад,  
и поэтому все эти типы восхитительных пицц имеют свои собственные  
имена. Будет хорошо, если мы этим воспользуемся и дадим пользовате-  
лям нашего класса Pizza более оптимальный интерфейс для создания  
объектов-пицц, которые они хотят.**

**Хороший и очевидный способ это сделать — использовать методы класса  
в качестве фабричных функций^ для различных видов пицц, которые мы  
можем создать:**

**class Pizza:**

**def init (self, ingredients):**

**self.ingredients = ingredients**

**def repr (self):**

**return f'Pizza({self.ingredients!r})'**

**@classmethod  
def margherita(cls):**

**return cls(['моцарелла', 'помидоры'])**

**@classmethod  
def prosciutto(cls):**

**return cls(['моцарелла', 'помидоры', 'ветчина'])**

**Обратите внимание на то, как я использую аргумент** cls **в фабричных  
методах** margherita **и** prosciutto **вместо вызова конструктора** Pizza **не-  
посредственно.**

**Вы можете использовать эту идиому, чтобы следовать принципу «Не по-  
вторяйся» (DRY). Если в какой-то момент мы решим этот класс переиме-  
новать, нам не нужно будет помнить об обновлении имени конструктора  
во всех фабричных функциях.**

**Итак, что же мы можем сделать с этими фабричными методами? Давайте  
их испытаем:**

**>>> Pizza.margherita()**

**Pizza(['моцарелла', 'помидоры'])**

**>>> Pizza.prosciutto()**

**Pizza(['моцарелла', 'помидоры', 'ветчина']) [[27]](#footnote-27)**

**Как видите, фабричные функции можно использовать для создания новых  
объектов Pizza, которые сконфигурированы именно так, как мы хотим.**

**Внутри они все используют одинаковый конструктор init и просто**

**обеспечивают краткую форму для запоминания самых разнообразных  
ингредиентов.**

**Еще один способ взглянуть на это использование методов класса — по-  
нять, что они позволяют определять для своих классов альтернативные  
конструкторы.**

**Python допускает всего один метод init в классе. Использование ме-**

**тодов класса позволяет добавлять столько альтернативных конструкторов,  
сколько потребуется. Это может сделать интерфейс ваших классов (до  
известной степени) самодокументирующим и упростит их использование.**

**Когда использовать статические методы**

**Здесь уже сложнее найти хороший пример, и знаете что? Я просто про-  
должу растягивать аналогию пиццы, делая ее все тоньше и тоньше^ (ам!)**

**И вот что я придумал:**

**import math**

**class Pizza:**

**def init (self, radius, ingredients):**

**self.radius = radius  
self.ingredients = ingredients**

**def repr (self):**

**return (f'Pizza({self.radius!r},'  
f'{self.ingredients!r})')**

**def area(self):**

**return self.circle\_area(self.radius)**

**@staticmethod  
def circle\_area(r):**

**return r \*\* 2 \* math.pi**

**Итак, что же я тут поменял? Прежде всего, я изменил конструктор и ме-  
тод repr , и теперь они принимают дополнительный аргумент radius.**

**Я также добавил метод экземпляра area(), который вычисляет и воз-  
вращает площадь пиццы. Это также будет подходящей кандидатурой для  
^property... но постойте, это же просто игрушечный пример.**

**Вместо того чтобы вычислять площадь непосредственно внутри метода  
area() при помощи общеизвестной формулы площади круга, я вынес это  
вычисление в отдельный статический метод circle\_area().**

**Давайте его испытаем!**

**>>> p = Pizza(4, ['mozzarella', 'tomatoes'])**

**>>> p**

**Pizza(4, {self.ingredients})**

**>>> p.area()**

**50.26548245743669**

**>>> Pizza.circle\_area(4)**

**50.26548245743669**

**Несомненно, этот пример по-прежнему довольно упрощенный, но он  
поможет объяснить некоторые преимущества, предоставляемые стати-  
ческими методами.**

**Как мы узнали, статические методы не могут получать доступ к состо-  
янию класса или экземпляра, потому что они не принимают аргумент  
cls или self. Этот факт является большим ограничением — но он также  
является замечательным сигналом, который обозначает, что тот или иной  
метод независим от всего остального вокруг него.**

**Из примера выше совершенно ясно, что circle\_area() никак не может  
модифицировать класс или экземпляр класса. (Разумеется, это ограни-  
чение всегда можно обойти при помощи глобальной переменной, но это  
уже к делу не относится.)**

**Итак, почему же это полезно?**

**Обозначение метода как статического не просто подсказка, что этот метод  
не сможет модифицировать состояние экземпляра или класса. Но, как вы  
убедились, это ограничение также подкрепляется во время выполнения  
программы Python.**

**Такие приемы дают четкое представление о составных частях вашей архи-  
тектуры классов для того, чтобы процесс новой разработки естественным  
образом направлялся в пределах этих границ. Безусловно, эти ограниче-  
ния достаточно легко нарушить. Но на практике они нередко помогают  
избежать непреднамеренных модификаций, которые идут вразрез с перво-  
начальным проектом.**

**Другими словами, использование статических методов и методов класса  
способствует передаче замысла разработчика, при этом достаточно под-  
крепляя этот замысел, чтобы избежать большинства ошибок «по недора-  
зумению» и ошибок, которые разрушили бы проект.**

**При экономном применении и только в тех случаях, когда это имеет  
смысл, написание части своих методов таким вот образом может предо-  
ставить преимущества в сопровождении и уменьшит вероятность того,  
что другие разработчики будут использовать ваши классы неправильно.**

**Статические методы также обладают преимуществами в том, что касается  
написания тестового программного кода. Поскольку метод circle\_area()  
абсолютно независим от остальной части класса, его намного легче про-  
тестировать.**

**Нам не придется переживать по поводу настройки полного экземпляра  
класса перед тем, как мы сможем протестировать этот метод в модуль-  
ном тесте. Мы просто можем действовать подобно тому, как мы дей-  
ствовали бы при тестировании обычной функции. И опять-таки, это  
облегчает сопровождение кода в будущем и обеспечивает связь между  
объектно-ориентированным и процедурным стилями программиро-  
вания.**

**Ключевые выводы**

**□ Методы экземпляра нуждаются в экземпляре класса и могут получать  
доступ к экземпляру через параметр self.**

* **Методы класса не нуждаются в экземпляре класса. Они не могут полу-  
  чать доступ к экземпляру (self), но у них есть доступ непосредственно  
  к самому классу через cls.**
* **Статические методы не имеют доступа ни к cls, ни к self. Они работа-  
  ют как обычные функции, но принадлежат пространству имен класса.**
* **Статические методы и методы класса сообщают и (до известной сте-  
  пени) подкрепляют замысел разработчика в отношении конструкции  
  класса. Это может обладать определенными преимуществами в сопро-  
  вождении кода.**

**Общие структуры  
данных Python**

**Что должен применять на практике и что должен твердо знать каждый  
разработчик на Python?**

**Структуры данных. Они являются основополагающими конструкциями,  
вокруг которых строятся программы. Каждая структура данных обеспечи-  
вает отдельно взятый способ организации данных с целью эффективного  
к ним доступа в зависимости от вашего варианта использования.**

**Убежден, что возвращение к основам для программиста всегда окупается,  
независимо от его уровня квалификации или опыта.**

**Нужно сказать, что я не сторонник того, что необходимо сосредоточи-  
ваться на расширении знаний об одних только структурах данных —  
проблема такого подхода заключается в том, что тогда мы застреваем  
в «стране грез» и не даем реальных результатов, пригодных для поставки  
клиентам^**

**Но я обнаружил, что небольшое время, потраченное на приведение  
в порядок своих знаний о структурах данных (и алгоритмах), всегда  
окупается.**

**Делаете ли вы это в течение нескольких дней в виде четко сформули-  
рованного «спринта» либо в виде затянувшегося проекта урывками тут  
и там, не имеет никакого значения. Так или иначе, обещаю, что время  
будет потрачено не напрасно.**

**Ладно, значит, структуры данных в Python, так? У нас есть списки, сло-  
вари, множества^ м-м-м. Стеки? Разве у нас есть стеки?**

**Видите ли, проблема в том, что Python поставляется с обширным набо-  
ром структур данных, которые находятся в его стандартной библиотеке.  
Однако их обозначение иногда немного «уводит в сторону».**

**Зачастую неясно, как именно общеизвестные «абстрактные типы данных»,  
такие как стек, соответствуют конкретной реализации на Python. Другие  
языки, например Java, больше придерживаются принципов «computer  
sciencе» и явной схемы именования: в Java список не просто «список» — это  
либо связный список** LinkedList**, либо динамический массив** ArrayList**.**

**Это позволяет легче распознать ожидаемое поведение и вычислительную  
сложность этих типов. В Python отдается предпочтение более простой  
и более «человеческой» схеме обозначения, и она мне нравится. Отчасти  
именно поэтому программировать на Python так интересно.**

**Но обратная сторона в том, что даже для опытных разработчиков на  
Python может быть неясно, как реализован встроенный тип** list**: как свя-  
занный список либо как динамический массив. И в один прекрасный день  
отсутствие этого знания приведет к бесконечным часам разочарования  
или неудачному собеседованию при приеме на работу.**

**В этой части книги я проведу вас по фундаментальным структурам  
данных и реализациям абстрактных типов данных (АТД), встроенным  
в Python и его стандартную библиотеку.**

**Здесь моя цель состоит в том, чтобы разъяснить, как наиболее распро-  
страненные абстрактные типы данных соотносятся с принятой в Python  
схемой обозначения, и предоставить краткое описание каждого из них.  
Эта информация также поможет вам засиять во всей красе на собеседо-  
ваниях по программированию на Python.**

**Если вы ищете хорошую книгу, которая приведет в порядок ваши общие  
познания относительно структур данных, то я настоятельно рекомендую  
книгу Стивена С. Скиены «Алгоритмы: построение и анализ» (Steven  
S. Skiena’s, The Algorithm Design Manual).**

**В ней выдерживается прекрасный баланс между обучением фундамен-  
тальным (и более продвинутым) структурам данных и демонстрацией**того, как применять их на практике в различных алгоритмах. Книга Стива  
послужила мне большим подспорьем при написании этих разделов.

1. **. 1 . Словари, ассоциативные массивы  
   и хеш-таблицы**

**В Python словари — центральная структура данных. В словарях хранится  
произвольное количество объектов, каждый из которых идентифициру-  
ется уникальным ключом словаря.**

**Словари также нередко называют ассоциативными массивами (associative  
arrays), ассоциативными хеш-таблицами (hashmaps), поисковыми та-  
блицами (lookup tables) или таблицами преобразования. Они допускают  
эффективный поиск, вставку и удаление любого объекта, связанного  
с заданным ключом.**

**Что это означает на практике? Оказывается, что телефонные книги пред-  
ставляют собой достойный аналог объектов-словарей из реальной жизни:**

**Телефонные книги позволяют быстро получать информацию (номер  
телефона), связанную с заданным ключом (именем человека) . Поэтому  
вместо того, чтобы читать телефонную книгу от корки до корки в поис-  
ках чьего-то номера, можно почти напрямую перескочить к имени и по-  
смотреть связанную с ним информацию**

**Эта аналогия несколько рушится, когда дело доходит до того, каким об-  
разом информация организована, чтобы допускать выполнение быстрых  
операций поиска. Но фундаментальные характеристики производитель-  
ности остаются прежними: словари позволяют быстро находить инфор-  
мацию, связанную с заданным ключом.**

**Резюмируя, словари — это одна из наиболее часто используемых и самых  
важных структур данных в информатике.**

**Итак, каким же образом Python обращается со словарями?**

**Давайте отправимся на экскурсию по реализациям словаря, имеющимся  
в ядре Python и стандартной библиотеке Python.**

**diet — ваш дежурный словарь**

**Из-за своей важности Python содержит надежную реализацию словаря,  
которая встроена непосредственно в ядро языка: тип данных diet1.**

**Для работы со словарями в своих программах Python также предостав-  
ляет немного полезного «синтаксического сахара». Например, синтаксис  
выражения с фигурными скобками для словаря и конструкция включения  
в словарь позволяют удобно определять новые объекты-словари:**

**Phonebook = {**

**'боб': 7387,**

**'элис': 3719,**

**'джек': 7052,**

**}**

**squares = {x: x \* x for x in range(6)}**

**>>> phonebook['элис']**

**3719**

**>>> squares**

**{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}**

**Есть некоторые ограничения относительно того, какие объекты могут  
использоваться в качестве допустимых ключей.**

**Словари Python индексируются ключами, у которых может быть любой  
хешируемый тип2: хешируемый объект имеет хеш-значение, которое**

**никогда не меняется в течение его жизни (см. hash ), и его можно**

**сравнивать с другими объектами (см. eq ). Кроме того, эквивалентные**

**друг другу хешируемые объекты должны иметь одинаковое хеш-значение.**

**Неизменяемые типы, такие как строковые значение и числа, являются  
хешируемыми объектами и хорошо работают в качестве ключей словаря.  
В качестве ключей словаря также можно использовать объекты-корте-  
жи — при условии, что они сами содержат только хешируемые типы. [[28]](#footnote-28) [[29]](#footnote-29)**

**Для большинства вариантов использования встроенная в Python реали-  
зация словаря делает все, что вам нужно. Словари хорошо оптимизирова-  
ны и лежат в основе многих частей языка: например, и атрибуты класса,  
и переменные в стековом фрейме во внутреннем представлении хранятся  
в словарях.**

**Словари Python основаны на хорошо протестированной и тонко на-  
строенной реализации хеш-таблицы, которая обеспечивает ожидаемые  
характеристики производительности с временной сложностью 0(1) для  
операций поиска, вставки, обновления и удаления в среднем случае.**

**Нет особых причин не использовать стандартную реализацию diet, вклю-  
ченную в Python. Тем не менее существуют специализированные сторон-  
ние реализации словаря, например списки с пропусками или словари на  
основе B-деревьев.**

**Помимо «обыкновенных» объектов diet, стандартная библиотека Python  
также содержит ряд реализаций специализированных словарей. Все эти  
специализированные словари опираются на встроенный класс словаря  
(и обладают его характеристиками производительности), но помимо этого  
еще добавляют некоторые удобные свойства.**

**Давайте их рассмотрим.**

**collections . OrderedDict — помнят порядок вставки ключей**

**В Python включен специализированный подкласс diet, который за-  
поминает порядок вставки добавляемых в него ключей: collections.  
OrderedDict1.**

**Хотя в Python 3.6 и выше стандартные экземпляры diet сохраняют по-  
рядок вставки ключей, такое поведение является всего лишь побочным  
эффектом реализации в Python и не определяется спецификацией языка2.  
Поэтому, если для работы вашего алгоритма порядок следования ключей [[30]](#footnote-30) [[31]](#footnote-31)**

**имеет значение, лучше всего четко донести эту идею, задействовав класс  
OrderDict явным образом.**

**Между прочим, OrderedDict не является встроенной составной частью  
базового языка и должен быть импортирован из модуля collections, на-  
ходящегося в стандартной библиотеке.**

**>>> import collections**

**>>> d = collections.OrderedDict(one=1, two=2, three=3)**

**>>> d**

**OrderedDict([('один', 1), ('два', 2), ('три', 3)])**

**>>> d['четыре'] = 4  
>>> d**

**OrderedDict([('один', 1), ('два', 2),**

**('три', 3), ('четыре', 4)])**

**>>> d.keys()**

**odict\_keys(['один', 'два', 'три', 'четыре'])**

**ccNections . defaultdict — возвращает значения,  
заданные по умолчанию для отсутствующих ключей**

**Класс defaultdict — это еще один подкласс словаря, который в своем  
конструкторе принимает вызываемый объект, возвращаемое значение  
которого будет использовано, если требуемый ключ нельзя найти1.**

**Это свойство может сэкономить на наборе кода и сделать замысел про-  
граммиста яснее в сравнении с использованием методов get() или от-  
лавливанием исключения KeyError в обычных словарях.**

**>>> from collections import defaultdict**

**>>> dd = defaultdict(list)**

* ***Попытка доступа к отсутствующему ключу его создает и***
* ***инициализирует, используя принятую по умолчанию фабрику,***
* ***то есть в данном примере ListQ:***

**>>> dd['собаки'].append('Руфус')**

**>>> dd['собаки'].append('Кэтрин')  
>>> dd['собаки'].append('Сниф')**

**>>> dd['собаки']**

**['Руфус', 'Кэтрин', 'Сниф']**

**collections . ChainMap — производит поиск в многочисленных  
словарях как в одной таблице соответствия**

**Структура данных collections.ChainMap группирует многочисленные  
словари в одну таблицу соответствия[[32]](#footnote-32). Поиск проводится по очереди во  
всех базовых ассоциативных объектах до тех пор, пока ключ не будет  
найден. Операции вставки, обновления и удаления затрагивают только  
первую таблицу соответствия, добавленную в цепочку.**

**>>> from collections import ChainMap  
>>> dict1 = {'один': 1, 'два': 2}**

**>>> dict2 = {'три': 3, 'четыре': 4}**

**>>> chain = ChainMap(dict1, dict2)**

**>>> chain**

**ChainMap({'один': 1, 'два': 2}, {'три': 3, 'четыре': 4})**

* ***ChainMap выполняет поиск в каждой коллекции в цепочке***
* ***слева направо, пока не найдет ключ (или не потерпит неудачу):***

**>>> chain['три']**

**3**

**>>> chain['один']**

**1**

**>>> chain['отсутствует']**

**КеуВггаг: 'отсутствует'**

**types . MappingProxyType — обертка для создания словарей  
только для чтения**

**MappingProxyType — это обертка стандартного словаря, которая предо-  
ставляет доступ только для чтения данных обернутого словаря [[33]](#footnote-33). Этот**

**класс был добавлен в Python 3.3 и может использоваться для создания  
неизменяемых версий словарей.**

**Например, он может быть полезен, если требуется вернуть словарь,  
передающий внутреннее состояние из класса или модуля, при этом  
препятствуя доступу к этому объекту для записи. Использование  
MappingProxyType позволяет вводить эти ограничения без необходимости  
сначала создавать полную копию словаря.**

**>>> from types import MappingProxyType**

**>>> writable = {'один': 1, 'два': 2} # доступный для обновления**

**>>> read\_only = MappingProxyType(writable)**

* ***Этот представитель/прокси с доступом только для чтения:***

**>>> read\_only['один']**

**1**

**>>> read\_only['один'] = 23  
TypeError:**

***"'mappingproxy' object does not support item assignment"***

* ***Обновления в оригинале отражаются в прокси:***

**>>> writable['один'] = 42**

**>>> read\_only**

**mappingproxy({'один': 42, 'один': 2})**

**Словари в Python: заключение**

**Все перечисленные в этом разделе питоновские реализации словаря  
являются действующими, они встроены в стандартную библиотеку  
Python.**

**Если вы ищете общую рекомендацию по поводу того, какой ассоциатив-  
ный тип использовать в ваших программах, я указал бы на встроенный  
тип данных dict. Он представляет собой универсальную и оптимизиро-  
ванную реализацию хеш-таблицы, которая встроена непосредственно  
в ядро языка.**

**Я порекомендовал бы использовать один из прочих перечисленных здесь  
типов данных, только если у вас есть особые требования, которые не могут  
быть обеспечены типом dict.**

**Да, я по-прежнему убежден, что все эти варианты допустимы, но, как  
правило, в большинстве случаев ваш исходный код будет яснее и легче  
в сопровождении для других разработчиков, если он будет опираться на  
стандартные словари Python.**

**Ключевые выводы**

* **Словари — это единственная центральная структура данных в Python.**
* **Встроенный тип diet будет «вполне приемлем» в большинстве случаев.**
* **Специализированные реализации, такие как словари с доступом толь-  
  ко для чтения или упорядоченные словари, имеются в стандартной  
  библиотеке Python.**

1. **. Массивоподобные структуры данных**

**Массив (array) — это фундаментальная структура данных, имеющаяся  
в большинстве языков программирования, и он имеет широкий спектр  
применений в самых разных алгоритмах.**

**В этом разделе мы рассмотрим реализации массива в Python, в кото-  
рых используются только базовые функциональные средства языка  
или функциональность, которая включена в стандартную библиотеку  
Python.**

**Вы увидите достоинства и недостатки каждого подхода, благодаря чему  
сможете решить, какая реализация подходит для вашего варианта исполь-  
зования. Но прежде чем начать, рассмотрим некоторые основы.**

**Как работают массивы и для чего они применяются?**

**Массивы состоят из записей данных, при этом записи имеют фиксиро-  
ванный размер, что позволяет эффективно размещать каждый элемент  
на основе его индекса.**

**Поскольку массивы хранят информацию в смежных блоках памяти, их  
рассматривают как непрерывные (нефрагментированные) структуры**

**данных (в противоположность связным структурам данных, таким как  
связные списки, например).**

**Аналогией из реального мира, соответствующей этой структуре данных,  
является автостоянка:**

**Автостоянку можно рассматривать как единое целое и как отдельный  
объект, но внутри автостоянки есть места для парковки, индексируемые  
по уникальному числу. Места для парковки являются контейнерами для  
транспортных средств — каждое место для парковки может либо быть  
пустым, либо содержать автомобиль, мотоцикл или другое транспортное  
средство, припаркованное там**

**Но не все автостоянки одинаковые:**

**Некоторые автостоянки могут быть ограничены только одним типом  
транспортного средства . Например, на кемпинговой автостоянке не раз-  
решено парковать велосипеды . «Ограниченная» автостоянка соответ-  
ствует структуре данных для «типизированного массива», которая допу-  
скает только те элементы, которые имеют одинаковый тип хранящихся  
в них данных.**

**С точки зрения производительности поиск элемента, содержащегося  
в массиве, выполняется очень быстро при условии, что указан индекс эле-  
мента. Для данного случая надлежащая реализация массива гарантирует  
постоянное 0(1) время доступа.**

**В своей стандартной библиотеке Python содержит несколько массивопо-  
добных структур данных, каждая из которых обладает слегка отличающи-  
мися характеристиками. Давайте их рассмотрим.**

**list — изменяемые динамические массивы**

**Списки (lists) являются составной частью ядра языка Python[[34]](#footnote-34). Несмотря  
на свое имя, списки Python реализованы как динамические массивы. Это  
означает, что список допускает добавление и удаление элементов и авто-**

**матически корректирует резервное хранилище, в котором эти элементы  
содержатся, путем выделения или высвобождения оперативной памяти.**

**Списки Python могут содержать произвольные элементы — в Python  
абсолютно «всё» является объектом, включая и функции. Поэтому вы  
можете сочетать и комбинировать разные типы данных и хранить их все  
в одном списке.**

**Такая возможность может быть очень мощной, но у нее есть и обратная  
сторона: поддержка многочисленных типов данных одновременно озна-  
чает, что данные, как правило, упакованы менее плотно. И в результате  
вся структура занимает больше места.**

**>>> arr = ['один', 'два', 'три']**

**>>> arr[0]**

***'один'***

* ***Списки имеют хороший метод repr:***

**>>> arr**

**['один', 'два', 'три']**

* ***Списки могут изменяться:***

**>>> arr[1] = 'привет'**

**>>> arr**

**['один', 'привет', 'три']**

**>>> del arr[1]**

**>>> arr**

**['один', 'три']**

* ***Списки могут содержать произвольные типы данных:***

**>>> arr.append(23)**

**>>> arr**

**['один', 'три', 23]**

**tuple — неизменяемые контейнеры**

**Аналогично спискам, кортежи тоже являются составной частью ядра  
языка Python1. Однако в отличие от списков, в Python объекты-кортежи**

**не изменяются. Это означает, что элементы не могут динамически добав-  
ляться или удаляться — все элементы в кортеже должны быть определены  
во время создания.**

**Точно так же, как и списки, кортежи могут содержать элементы произ-  
вольных типов данных. В этой гибкости много мощности, но, опять-таки,  
это также означает, что данные упакованы менее плотно, чем это было бы  
в типизированном массиве.**

**>>> arr = 'один', 'два', 'три'**

**>>> arr[0]**

***'one'***

* ***Кортежи имеют хороший метод repr:***

**>>> arr ('один', 'два', 'три')**

* ***Кортежи не могут изменяться:***

**>>> arr[1] = 'привет'**

***TypeError:***

***"'tuple' object does not support item assignment"***

**>>> del arr[1]**

**TypeError:**

***"'tuple' object doesn't support item deletion"***

* ***Кортежи могут содержать произвольные типы данных:***
* ***(При добавлении элементов создается копия кортежа)***

**>>> arr + (23,)**

**('один', 'два', 'три', 23)**

**array, array — элементарные типизированные массивы**

**Модуль Python array обеспечивает пространственно-эффективное хра-  
нение элементарных типов данных в стиле языка C, таких как байты,  
32-разрядные целые числа, числа с плавающей точкой и т. д.**

**Массивы, создаваемые на основе класса array.array, могут изменяться  
и ведут себя аналогично спискам, за исключением одного важного раз-  
личия — они являются «типизированными массивами», ограниченными  
единственным типом данных1.**

**Из-за этого ограничения объекты array.array со многими элементами  
более пространственно эффективны, чем списки и кортежи. Хранящиеся  
в них элементы плотно упакованы, и это может быть полезно, если вам  
нужно хранить много элементов одного и того же типа.**

**Кроме того, массивы поддерживают многие из тех же методов, что и у  
обычных списков, и вы можете их использовать в качестве «прямой за-  
мены» без необходимости вносить в свой код другие изменения.**

**>>> import array**

**>>> arr = array.array('f', (1.0, 1.5, 2.0, 2.5))**

**>>> arr[1]**

**1.5**

* ***Массивы имеют хороший метод repr:***

**>>> arr**

**array('f', [1.0, 1.5, 2.0, 2.5])**

* ***Массивы могут изменяться:***

**>>> arr[1] = 23.0**

**>>> arr**

**array('f', [1.0, 23.0, 2.0, 2.5])**

**>>> del arr[1]**

**>>> arr**

**array('f', [1.0, 2.0, 2.5])**

**>>> arr.append(42.0)**

**>>> arr**

**array('f', [1.0, 2.0, 2.5, 42.0])**

* ***Массивы - это "типизированные" структуры данных:***

**>>> arr[1] = 'привет'**

**TypeError: "must be real number, not str"**

**str — неизменяемые массивы символов Юникода**

**В Python 3.x объекты строкового типа str используются для хранения  
текстовых данных в виде неизменяемых последовательностей символов  
Юникода[[35]](#footnote-35). В сущности, это означает, что тип str представляет собой  
неизменяемый массив символов. Как это ни странно, но тип str также  
является рекурсивной структурой данных: каждый символ в строке сам  
является объектом str длиной, равной 1.**

**Строковые объекты пространственно эффективны, потому что они плотно  
упакованы и специализируются на одном-единственном типе данных.  
Если вы храните текст в кодировке Юникод, то лучше использовать этот  
тип данных. Поскольку строки в Python не могут изменяться, модифи-  
кация строкового значения требует создания модифицированной копии.  
Самым близким эквивалентом «изменяющейся последовательности  
символов» будет список, в котором символы хранятся по отдельности.**

**>>> arr = 'abcd'**

**>>> arr[1]**

***'b'***

**>>> arr  
'abed'**

* ***Строки неизменяемы:***

**>>> arr[1] = 'e'**

***TypeError:***

**"'str' object does not support item assignment"**

**>>> del arr[1]**

***TypeError:***

**"'str' object doesn't support item deletion"**

* ***Строки могут быть распакованы в список, в результате чего***
* ***они получают изменяемое представление:***

**>>> list('abcd')**

**['a', 'b', 'c', 'd']**

**>>> ''.join(list('abcd'))**

***'abed'***

***# Строки — это рекурсивные структуры данных:*>>> type('abc')**

**"<class 'str'>"**

**>>> type('abc'[0])**

***"<eLass 'str'>"***

**bytes — неизменяемые массивы одиночных байтов**

**Объекты bytes представляют собой неизменяемые последовательности  
одиночных байтов (целых чисел в диапазоне 0 < x < 255)1. В концептуаль-  
ном плане они подобны объектам str и их также можно представить как  
неизменяемые массивы байтов.**

**Аналогично строковому типу, тип bytes имеет свой собственный лите-  
ральный синтаксис, предназначенный для создания объектов, и объекты  
этого типа пространственно эффективны. Объекты bytes не могут изме-  
няться, но, в отличие от строковых объектов, для «изменяемых массивов  
байтов» есть специальный тип данных, который называется bytearray,  
или байтовый массив, в который они могут быть распакованы. Вы узнаете**

1. **нем подробнее в следующем подразделе.**

**>>> arr = bytes((0, 1, 2, 3))**

**>>> arr[1]**

**1**

* ***Байтовые литералы имеют свой собственный синтаксис:***

**>>> arr**

**b'x00x01x02x03'**

**>>> arr = b'x00x01x02x03'**

* ***Разрешены только допустимые "байты":***

**>>> bytes((0, 300))**

**ѴаіиеВггаг: "bytes must be in range(0, 256)"**

* ***Байты неизменяем^!:***

**>>> arr[1] = 23  
TypeError:**

**"'bytes' object does not support item assignment"**

**>>> del arr[1]**

***TypeError:***

***"'bytes' object doesn't support item deletion"***

**bytearray — изменяемые массивы одиночных байтов**

**Тип bytearray представляет собой изменяемую последовательность целых  
чисел в диапазоне 0 < x < 255[[36]](#footnote-36). Они тесно связаны с объектами bytes, при  
этом главное их отличие в том, что объекты bytearray можно свободно  
изменять — вы можете переписывать элементы, удалять существующие  
элементы или добавлять новые. Объект bytearray будет соответствующим  
образом расти и сжиматься.**

**Объекты bytearray могут быть преобразованы обратно в неизменяе-  
мые объекты bytes, но это влечет за собой копирование абсолютно всех  
хранящихся в них данных — весьма медленная операция, занимающая  
O(n) времени.**

**>>> arr = bytearray((0, 1, 2, 3))**

**>>> arr[1]**

**1**

* ***Метод repr для bytearray:***

**>>> arr bytearray(b'x00x01x02x03')**

* ***Байтовые массивы bytearray изменяемы:***

**>>> arr[1] = 23**

**>>> arr**

**bytearray(b'x00x17x02x03')**

**>>> arr[1]**

**23**

* ***Байтовые массивы bytearray могут расти и сжиматься в размере:***

**>>> del arr[1]**

**>>> arr**

**bytearray(b'x00x02x03')**

**>>> arr.append(42)**

**>>> arr**

**bytearray(b'x00x02x03\*')**

* ***Байтовые массивы bytearray могут содержать только "байты"***
* ***(целые числа в диапазоне 0 <= x <= 255)***

**>>> arr[1] = 'привет'**

**TypeError: "an integer is required"**

**>>> arr[1] = 300**

**VaiueErrar: "byte must be in range(0, 256)"**

* ***Bytearrays может быть преобразован в байтовые объекты:***
* ***(Это скопирует данные)***

**>>> bytes(arr)  
b'x00x02x03\*'**

**Ключевые выводы**

**В том, что касается реализации массивов в Python, вы можете выбирать  
из широкого круга встроенных структур данных. В этом разделе мы сосре-  
доточились на ключевых функциональных средствах языка и структурах  
данных, включенных только в стандартную библиотеку.**

**Если вы готовы выйти за пределы стандартной библиотеки Python, то  
сторонние пакеты, такие как NumPy^, предлагают широкий спектр мас-  
сивоподобных реализаций с большим быстродействием для научных  
вычислений и науки о данных.**

**Если ограничиваться массивоподобными структурами данных, включен-  
ными в Python, то наш выбор сводится к следующему.**

**Вам нужно хранить произвольные объекты, которые потенциально  
могут иметь смешанные типы данных? Используйте список или кортеж  
в зависимости от того, хотите вы иметь неизменяемую структуру данных  
или нет.**

**У вас есть числовые (целочисленные или с плавающей точкой) данные  
и для вас важны плотная упаковка и производительность? Попробуйте  
array.array и посмотрите, способен ли этот тип делать все, что вам нуж-  
но. Кроме того, рассмотрите выход за пределы стандартной библиотеки  
и попробуйте такие пакеты, как NumPy или Pandas[[37]](#footnote-37) [[38]](#footnote-38).**

**У вас есть текстовые данные, представленные символами Юникода?**

**Используйте встроенный в Python тип str. Если вам нужна «изменяемая  
последовательность символов», то используйте list как список символов.**

**Вы хотите хранить нефрагментированный блок байтов? Используйте  
неизменяемый тип bytes, либо bytearray, если вам нужна изменяемая  
структура данных.**

**В большинстве случаев мне нравится начинать с простого списка list.  
И только потом я конкретизирую используемый тип, если производитель-  
ность или занимаемое пространство оперативной памяти становятся про-  
блемой. В большинстве случаев использование массивоподобной струк-  
туры данных общего назначения, такой как список list, обеспечивает  
наибольшую скорость разработки и удобство во время программирования.**

**Для себя я понял, что в самом начале это обычно намного важнее, чем  
пытаться выжимать последнюю каплю производительности.**

1. **. Записи, структуры и объекты  
   переноса данных**

**Записи, как и структуры данных, по сравнению с массивами обеспечивают  
фиксированное количество полей, у каждого из которых может быть имя,  
а также другой тип.**

**В этом разделе вы увидите, как реализовывать в Python записи, структуры  
и «старые добрые объекты данных» с использованием всего лишь встро-  
енных типов данных и классов из стандартной библиотеки.**

**Кстати, здесь я использую определение понятия «запись» в широком  
смысле. Например, я также собираюсь обсудить такие типы, как встроен-  
ный в Python тип tuple, который может как считаться, так и не считаться  
записью в строгом смысле этого слова, потому что кортежи не обеспечи-  
вают именованные поля.**

**Python предлагает несколько типов данных, которые можно использовать  
для реализации записей, структур и объектов переноса данных. В этом  
разделе вы кратко рассмотрите каждую реализацию и ее уникальные ха-  
рактеристики. В конце раздела вы найдете резюме и руководство для при-  
нятия решений, которое поможет вам сделать свой собственный выбор.**

**Ладно, давайте начнем!**

**dict — простые объекты данных**

**Словари Python хранят произвольное количество объектов, при этом  
каждый идентифицируется уникальным ключом[[39]](#footnote-39). Словари также нередко  
называются ассоциативными массивами или таблицами соответствий  
и позволяют производить эффективный поиск, вставку и удаление любого  
объекта, связанного с заданным ключом.**

**В Python использование словарей в качестве типа данных запись или  
объекта данных вполне возможно. Словари в Python легко создаются,  
поскольку они имеют свой собственный синтаксический сахар, который  
встроен в язык в форме литералов словаря. Синтаксис словаря краток  
и довольно удобен для набора на клавиатуре.**

**Объекты данных, создаваемые с использованием словарей, могут изме-  
няться, и при этом практически отсутствует защита от опечаток в именах  
полей, поскольку поля могут свободно добавляться и удаляться в любое  
время. Оба этих свойства способны добавить поразительные ошибки,  
и всегда существует компромисс между удобством и устойчивостью  
к ошибкам, которого нужно достигать.**

**carl = {**

**'цвет': 'красный',**

**'пробег': 3812.4,**

**'автомат': True,**

**}**

**car2 = {**

**'цвет': 'синий',**

**'пробег': 40231,**

**'автомат': False,**

**}**

***# Словари имеют хороший метод repr:*>>> car2**

**{'цвет': 'синий', 'автомат': False, 'пробег': 40231}**

* ***Получить пробег:***

**>>> car2['пробег']**

**40231**

* ***Словари изменяемы:***

**>>> car2['пробег'] = 12**

**>>> car2['лобовое стекло'] = 'треснутое'**

**>>> car2**

**{'лобовое стекло': 'треснутое', 'цвет': 'синий',  
'автомат': False, 'пробег': 12}**

* ***Отсутствует защита от неправильных имен полей***
* ***или отсутствующих/лишних полей:*cans = {**

**'цвет': 'зеленый',**

**'автомат': False,**

**'лобовое стекло': 'треснутое',**

**}**

**tuple — неизменяемые группы объектов**

**Кортежи в Python представляют собой простые структуры данных, пред-  
назначенные для группирования произвольных объектов[[40]](#footnote-40). Кортежи не-  
изменяемы — после их создания их нельзя исправить.**

**С точки зрения производительности кортежи занимают чуть меньше  
оперативной памяти, чем списки в Python[[41]](#footnote-41), и к тому же быстрее созда-  
ются.**

**Как вы видите в приведенном ниже результате дизассемблирования  
байткода, конструирование кортежной константы занимает всего один  
код операции LOAD\_CONST, в то время как конструирование объекта-списка  
с одинаковым содержимым требует еще нескольких операций:**

**>>> import dis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **>>> dis.dis(compile("(23,** | **'a',** | **'b', '** | **'c')", '',** | **'eval'))** |
| **0 LOAD\_CONST** | **4** | **((23,** | **'a', 'b',** | **'c'))** |
| **3 RETURN\_VALUE** |  |  |  |  |
| **>>> dis.dis(compile("[23,** | **'a',** | **'b', '** | **'c']", '',** | **'eval'))** |
| **0 LOAD\_CONST** | **0** | **(23)** |  |  |
| **3 LOAD\_CONS** | **1** | **('a')** |  |  |
| **6 LOAD\_CONS** | **2** | **('b')** |  |  |
| **9 LOAD\_CONST** | **3** | **('c')** |  |  |
| **12 BUILD LIS** | **4** |  |  |  |

**15 RETURN\_VALUE**

**Однако вам не стоит особенно налегать на эти различия. На практике  
разница в производительности часто будет незначительной, и попытка  
выжать из программы больше эффективности, переключаясь со списков  
на кортежи, вероятно, будет нерациональной.**

**Возможным недостатком простых кортежей является то, что данные,  
которые вы в них храните, извлекаются только путем доступа к корте-  
жу через целочисленные индексы. У вас не получится назначить имена  
отдельным хранящимся в кортеже свойствам. А это может сказаться на  
удобочитаемости исходного кода.**

**Кроме того, кортеж всегда является ситуативной структурой: трудно  
гарантировать, что у двух кортежей будет одинаковое количество полей  
и одинаковые хранящиеся в них свойства.**

**А это — раздолье для ошибок «по недоразумению», например для раз-  
ночтений порядка следования полей. Поэтому я рекомендую держать  
минимальное количество полей в кортеже.**

***# Поля: цвет, пробег, автомат***

**>>> carl = ('красный', 3812.4, True)**

**>>> car2 = ('синий', 40231.0, False)**

***# Экземпляры кортежа имеют хороший метод repr:  
>>>* carl**

**('красный', 3812.4, True)**

**>>> car2**

**('синий', 40231.0, False)**

* ***Получить пробег:***

**>>> car2[1]**

**40231.0**

* ***Кортежи неизменяем^!:***

**>>> car2[1] = 12  
TypeError:**

***"'tuple' object does not support item assignment"***

* ***Нет защиты от неверных имен полей***
* ***или отсутствующих/лишних полей:***

**>>> car3 = (3431.5, 'зеленый', True, 'серебряный')**

**Написание собственного класса — больше работы,  
больше контроля**

**Классы позволяют определять «шаблоны» многократного использования  
для объектов данных, причем эти шаблоны гарантируют, что каждый объ-  
ект предоставляет одинаковый набор полей.**

**Использование обычных классов Python в качестве типов данных запись  
вполне возможно, но это также влечет за собой ручную работу, связанную  
с получением удобных функциональных возможностей у других реали-  
заций. Например, добавление новых полей в конструктор init будет**

**многословным и займет время.**

**Кроме того, принятое по умолчанию строковое представление объек-  
тов-экземпляров, создаваемых на основе собственных классов, не очень  
полезно. Чтобы это исправить, вам, вероятно, придется добавить свой  
собственный метод repr ^, который, как правило, довольно многосло-**

**вен и подлежит обновлению всякий раз, когда вы добавляете новое поле.**

**Хранящиеся в классах поля могут изменяться, и новые поля могут до-  
бавляться свободно, нравится вам это или нет. С помощью декоратора  
^property можно обеспечить себе большее управление и создавать поля  
с доступом только для чтения[[42]](#footnote-42) [[43]](#footnote-43), но это требует написания большего коли-  
чества связующего кода.**

**Написание собственного класса — отличная возможность, когда в объек-  
ты-записи требуется добавить бизнес-логику и поведение с использовани-  
ем методов. Однако это означает, что такие объекты технически больше  
не являются простыми объектами данных.**

**class Car:**

**def init (self, color, mileage, automatic):**

**self.color = color  
self.mileage = mileage  
self.automatic = automatic**

**>>> carl = Саг('красный', 3812.4, True)**

**>>> car2 = Саг('синий', 40231.0, False)**

***# Получить пробег:***

**>>> car2.mileage  
40231.0**

* ***Классы изменяемы:***

**>>> car2.mileage = 12**

**>>> car2.windshield = 'треснутое'**

* ***Строковое представление не очень полезно***

***repr* *):***

* ***(приходится добавлять написанный вручную метод*>>> car1**

**<Car object at 0x1081e69e8>**

**collections . namedtuple — удобные объекты данных**

**Класс namedtuple, доступный в Python 2.6+, предоставляет расширение  
встроенного типа данных tuple1. Аналогично определению собственного  
класса, применение именованного кортежа namedtuple позволяет опре-  
делять «шаблоны» многократного использования для своих записей,  
гарантирующие использование правильных имен полей.**

**Именованные кортежи неизменяемы, как и обычные кортежи. Это означа-  
ет, что вы не можете добавлять новые поля или изменять существующие  
поля после того, как экземпляр namedtuple был создан. [[44]](#footnote-44)**

**Помимо этого, именованные кортежи являются, скажем так, именован-  
ными кортежами (named tuples). Доступ к каждому хранящемуся в них  
объекту можно получить по уникальному идентификатору. Это освобож-  
дает от необходимости запоминать целочисленные индексы или идти  
обходными методами, например определять индексы целочисленных  
констант в качестве мнемокодов.**

**На внутреннем уровне объекты namedtuple реализованы как обычные  
классы Python. В том, что касается использования оперативной памяти,  
они тоже «лучше» обычных классов и столь же эффективны с точки зре-  
ния потребляемой оперативной памяти, что и обычные кортежи:**

**>>> from collections import namedtuple  
>>> from sys import getsizeof**

**>>> p1 = namedtuple('Point', 'x y z')(1, 2, 3)**

**>>> p2 = (1, 2, 3)**

**>>> getsizeof(p1)**

**72**

**>>> getsizeof(p2)**

**72**

**Именованные кортежи могут довольно просто привести в порядок ис-  
ходный код и сделать его более удобочитаемым, обеспечив вашим данным  
более совершенную структуру.**

**По моему опыту, переход от ситуативных типов данных, таких как словари  
с фиксированным форматом, к именованным кортежам помогает яснее  
выражать свои намерения. Нередко, когда я применяю эту рефакториза-  
цию, каким-то невообразимым образом я прихожу к более совершенному  
решению проблемы, с которой сталкиваюсь.**

**Использование именованных кортежей вместо неструктурированных  
кортежей и словарей может облегчить жизнь и моим коллегам, потому  
что именованные кортежи позволяют раздавать данные в «самодокумен-  
тированном» виде (в известной степени).**

**>>> from collections import namedtuple**

**>>> Car = namedtuple('ABTo' , 'цвет пробег автомат')  
>>> carl = Саг('красный', 3812.4, True)**

* ***Экземпляры имеют хороший метод repr:***

**>>> carl Авто(цвет='красный', пробег=3812.4, автомат=Тгие)**

* ***Доступ к полям:***

**>>> сагі.пробег**

**3812.4**

* ***Поля неизменяемы:***

**>>> сагі.пробег = 12  
AttributeErrar: "can't set attribute"**

**>>> car1.лобовое\_стекло = 'треснутое'**

***AttributeError:***

***"'Car' object has no attribute 'лобовое\_стекло'"***

**typing . NamedTuple — усовершенствованные  
именованные кортежи**

**Этот класс был добавлен в Python 3.6 и является младшим братом класса  
namedtuple в модуле collections1. Он очень похож на namedtuple, и его**

**главное отличие состоит в том, что у него есть обновленный синтаксис для  
определения новых типов записей и добавленная поддержка подсказок  
при вводе исходного кода.**

**Кроме того, обратите внимание, что сигнатуры типов не поддерживаются  
без отдельного инструмента проверки типов, такого как mypy2. Но даже  
без инструментальной поддержки они могут предоставлять полезные под-  
сказки для других программистов (или могут быть ужасно запутанными,  
если подсказки в отношении типов становятся устаревшими).**

**>>> from typing import NamedTuple  
class Car(NamedTuple):  
цвет: str  
пробег: float  
автомат: bool [[45]](#footnote-45) [[46]](#footnote-46)**

**>>> carl = Саг('красный', 3812.4, True)**

* ***Экземпляры имеют хороший метод repr:***

**>>> carl Саг(цвет='красный', пробег=3812.4, автомат=Тгие)**

* ***Доступ к полям:***

**>>> сагі.пробег 3812.4**

* ***Поля неизменяем^!:***

**>>> сагі.пробег = 12  
AttributeErrar: "can't set attribute"**

**>>> car1.лобовое\_стекло = 'треснутое'**

***AttributeError:***

***"'Car' object has no attribute 'лобовое\_стекло'"***

* ***Аннотации типа не поддерживаются без отдельного***
* ***инструмента проверки типов, такого как mypy:***

**>>> Car('красный', 'НЕВЕЩЕСТВЕННЫЙ', 99)  
Car(цвет='красный', пробег='НЕВЕЩЕСТВЕННЫЙ', автомат=99)**

**struct . Struct — сериализованные С-структуры**

**Класс struct.Struct1 выполняет преобразование между значениями  
Python и структурами C, сериализованными в форму объектов Python  
bytes. Например, он может использоваться для обработки двоичных  
данных, хранящихся в файлах или поступающих из сетевых соединений.**

**Структуры Struct определяются с использованием форматного строко-  
подобного мини-языка, который позволяет определять расположение  
различных типов данных C, таких как char, int и long, а также их без-  
знаковых вариантов.**

**Сериализованные структуры редко используются для представления объ-  
ектов данных, предназначенных для обработки исключительно внутри  
кода Python. Они нужны в первую очередь в качестве формата обмена  
данными, а не как способ их хранения в оперативной памяти, применяе-  
мый только программным кодом Python.**

**В некоторых случаях упаковка примитивных данных в структуры по-  
зволяет уменьшить объем потребляемой оперативной памяти, чем их [[47]](#footnote-47)**

**хранение в других типах данных. Однако чаще всего такая работа будет  
довольно продвинутой (и, вероятно, ненужной) оптимизацией.**

**>>> from struct import Struct  
>>> MyStruct = Struct('i?f')**

**>>> data = MyStruct.pack(23, False, 42.0)**

* ***Вы получаете двоичный объект данных (blob):***

**>>> data**

**b'x17x00x00x00x00x00x00x00x00x00(B'**

* ***BLOB-объекты можно снова распаковать:***

**>>> MyStruct.unpack(data)**

**(23, False, 42.0)**

**types . SimpleNamespace — причудливый  
атрибутивный доступ**

**А вот еще один «эзотерический» вариант реализации объектов данных  
в Python: types.SimpleNamespace1. Этот класс был добавлен в Python 3.3,  
и он обеспечивает атрибутивный доступ к своему пространству имен.**

**Это означает, что экземпляры SimpleNamespace показывают все свои клю-  
чи как атрибуты класса. А значит, вы можете использовать «точечный»  
атрибутивный доступ объект.ключ вместо синтаксиса с индексацией  
в квадратных скобках объект['ключ'], который применяется обычными  
словарями. Все экземпляры также по умолчанию включают содержатель-  
ный метод repr .**

**Как видно из его названия, тип SimpleNamespace прост в использовании!  
Это, в сущности, прославленный словарь, который предоставляет доступ  
по атрибуту и выдает приличную распечатку. Атрибуты могут свободно  
добавляться, изменяться и удаляться.**

**>>> from types import SimpleNamespace  
>>> car1 = SimpleNamespace(цвет='красный', [[48]](#footnote-48)**

**... пробег=3812.4,**

**... автомат=Тгие)**

* ***Метод repr по умолчанию:***

**>>> carl**

**namespace(aBTOMaT=True, пробег=3812.4, цвет='красный')**

* ***Экземпляры поддерживают атрибутивный доступ и могут изменяться:  
  >>>* сагі.пробег = 12**

**>>> саг1.лобовое\_стекло = 'треснутое'**

**>>> del сагі.автомат  
>>> carl**

**namespace(лобовое\_стекло='треснутое', пробег=12, цвет='красный')**

**Ключевые выводы**

**Итак, какой же тип следует использовать для объектов данных в Python?  
Как вы убедились, есть целый ряд различных вариантов для реализации  
записей или объектов данных. Как правило, ваше решение будет зависеть  
от вашего сценария использования:**

**У вас есть всего несколько (2-3) полей: использование обыкновенного  
объекта-кортежа может подойти, если порядок следования полей легко  
запоминается или имена полей излишни. Например, представьте точку  
(x, у, z) в трехмерном пространстве.**

**Вам нужны неизменяемые поля: в данном случае обыкновенные кортежи,  
collections.namedtuple и typing.NamedTuple, дадут неплохие возмож-  
ности для реализации этого типа объекта данных.**

**Вам нужно устранить имена полей, чтобы избежать опечаток: вашими  
друзьями здесь будут collections.namedtuple и typing.NamedTuple.**

**Вы не хотите усложнять: обыкновенный объект-словарь может быть  
хорошим вариантом из-за удобного синтаксиса, который сильно напо-  
минает JSON.**

**Вам нужен полный контроль над вашей структурой данных: самое время  
написать собственный класс с методами-модификаторами (сеттерами)  
и методами-получателями (геттерами) @property.**

**Вам нужно добавить в объект поведение (методы): вам следует на-  
писать собственный класс с нуля либо путем расширения collections.  
namedtuple или typing.NamedTuple.**

**Вам нужно плотно упаковать данные, чтобы сериализовать их для  
записи на жесткий диск или отправить их по Сети: самое время навести  
справки по поводу struct.Struct, потому что этот объект представляет  
собой превосходный вариант использования.**

**Если вы ищете безопасный вариант, который можно использовать по  
умолчанию, то моя общая рекомендация в отношении реализации про-  
стой записи, структуры или объекта данных в Python будет следующей:  
использовать collections.namedtuple в Python 2.x и его младшего брата,  
typing.NamedTuple, в Python 3.**

1. **. Множества и мультимножества**

**В этом разделе вы увидите, как в Python реализуются такие структуры  
данных, как изменяемое и неизменяемое множество и мультимножество  
(или тип bag, то есть мешок), с использованием встроенных типов данных  
и классов стандартной библиотеки. Однако сначала давайте составим  
краткое резюме по поводу того, что такое множество.**

**Множество представляет собой неупорядоченную коллекцию объектов,  
которая не допускает повторяющихся элементов. Как правило, множе-  
ства используются для быстрой проверки принадлежности значения  
множеству, вставки новых значений в множество, удаления значений из  
множества и вычисления на множествах операций, таких как объединение  
или пересечение двух множеств.**

**Предполагается, что в «надлежащей» реализации множества операции  
проверки на принадлежность будут выполняться за быстрое 0(1) время.  
Операции объединения, пересечения, разности и взятия подмножеств  
должны в среднем занимать 0(n) времени. В реализациях множества,  
включенных в стандартную библиотеку Python, данные характеристики  
производительности соблюдаются[[49]](#footnote-49).**

**Точно так же, как и словари, множества в Python обрабатываются особым  
образом и имеют свой синтаксический сахар, упрощающий их создание.  
Например, синтаксис выражения с фигурными скобками для множеств  
и конструкция включения в множество позволяют удобно определять  
новые экземпляры множеств:**

**vowels = {'a', 'e', 'i', 'o', 'u'}  
squares = {x \* x for x in range(10)}**

**Тем не менее следует быть осторожными: для того чтобы создать пу-  
стое множество, вам нужно вызвать конструктор set(). Использование  
фигурных скобок {} неоднозначно и вместо этого создаст пустой сло-  
варь.**

**Python и его стандартная библиотека предоставляют несколько реализа-  
ций множества. Давайте их рассмотрим.**

**set — ваше дежурное множество**

**Это встроенная в Python реализация множества1. Тип set изменяемый  
и допускает динамическую вставку и удаление элементов.**

**>>> vowels = {'а', 'о', 'э',**

**>>> 'э' in vowels**

**True**

**и**

**у**

**ы**

**е**

**е**

**ю**

**'я'}**

**Множества Python set подкрепляются типом данных diet и обладают  
одинаковыми характеристиками производительности. Любой хешируе-  
мый объект может храниться в множестве set2.**

**>>> letters = set('алиса')**

**>>> letters.interseetion(vowels)  
{'а', 'и'}**

**>>> vowels.add('х') [[50]](#footnote-50) [[51]](#footnote-51)**

**>>> vowels**

**{'х', 'о', 'э', 'у', 'и', 'ы', 'е', 'е', 'ю', 'а', 'я'}**

**>>> len(vowels)  
6**

**frozenset — неизменяемые множества**

**Класс frozenset реализует неизменяемую версию множества set. Такое  
множество не может быть изменено после того, как оно было сконстру-  
ировано1. Множества frozenset статичны и допускают только операции  
с запросами в отношении своих элементов (никаких вставок или удале-  
ний). Поскольку множества frozenset статичны и хешируемы, они могут  
использоваться в качестве ключей словаря или в качестве элементов  
другого множества, а это то, что невозможно с обычными (изменяемыми)  
объектами-множествами set.**

**>>> vowels = frozenset({'a', 'о', 'э', 'и', 'у', 'ы', 'е', 'е', 'ю',**

**'я'}) >>> vowels.add('р')**

***AttributeError:***

**"'frozenset' object has no attribute 'add'"**

* ***Множества frozenset хешируемы и могут***
* ***использоваться в качестве ключей словаря:***

**>>> d = { frozenset({1, 2, 3}): 'привет' }**

**>>> d[frozenset({1, 2, 3})]**

***'привет'***

**collections . Counter — мультимножества**

**Класс collections.Counter стандартной библиотеки Python реализует  
тип «мультимножество» (или «мешок»), который допускает неоднократ-  
ное появление элемента в множестве [[52]](#footnote-52) [[53]](#footnote-53)**

**Это бывает полезно, если вам нужно вести учет не только того, при-  
надлежит ли элемент множеству, но и того, сколько раз он был включен  
в множество:**

**>>> from collections import Counter  
>>> inventory = Counter()**

**>>> loot = {'клинок': 1, 'хлеб': 3}**

**>>> inventory.update(loot)**

**>>> inventory**

**Counter({'клинок': 1, 'хлеб': 3})**

**>>> more\_loot = {'клинок': 1, 'яблоко': 1}**

**>>> inventory.update(more\_loot)**

**>>> inventory**

**Counter({'клинок': 2, 'хлеб': 3, 'яблоко': 1})**

**Приведу одно предостережение относительно класса Counter: следует со-  
блюдать осторожность во время подсчета количества элементов в объекте  
Counter. В результате вызова функции len() возвращается количество  
уникальных элементов в мультимножестве, тогда как общее количество  
элементов может быть получено с использованием функции sum:**

**>>> len(inventory)**

1. ***# Количество уникальных элементов*>>> sum(inventory.values())**

**6 *# Общее количество элементов***

**Ключевые выводы**

* **Множество является еще одной полезной и широко используемой  
  структурой данных, включенной в Python и ее стандартную библио-  
  теку.**
* **Используйте встроенный тип set, когда вы хотите получить изменя-  
  емое множество.**
* **Объекты frozenset хешируемы и могут использоваться в качестве  
  словаря или ключей множества.**
* **Класс collections.Counter реализует структуры данных «мультимно-  
  жество», или «мешок».**

1. **. Стеки (с дисциплиной доступа LIFO)**

**Стек представляет собой коллекцию объектов, которая поддерживает бы-  
струю семантику доступа «последним пришел — первым ушел» (LIFO — last  
in, first out) для вставок и удалений. В отличие от списков или множеств,  
стеки, как правило, не допускают произвольного доступа к объектам,  
которые они содержат. Операции вставки и удаления также нередко на-  
зываются вталкиванием (push) и выталкиванием (pop).**

**Полезной аналогией для стековой структуры данных из реального мира  
является стопка тарелок:**

**Новые тарелки добавляются на вершину стопки . И поскольку тарелки  
дорогие и тяжелые, можно взять только самую верхнюю тарелку (метод  
«последним пришел — первым ушел») . Чтобы добраться до тарелок,  
которые находятся внизу стопки, необходимо поочередно удалить все  
тарелки, которые находятся выше**

**Стеки и очереди похожи. Обе эти структуры данных являются линей-  
ными коллекциями элементов, и разница между ними состоит в порядке  
доступа к элементам.**

**В случае с очередью вы удаляете элемент, который был добавлен в нее  
раньше всех (метод «первым пришел — первым ушел», или FIFO); однако  
в случае со стеком вы удаляете элемент, который был добавлен в него  
позже всех (метод «последним пришел — первым ушел», или LIFO).**

**С точки зрения производительности предполагается, что надлежащая  
реализация стека будет занимать O(1) времени на операции вставки  
и удаления.**

**Стеки находят широкое применение в алгоритмах, например в синтакси-  
ческом анализе языка и управлении рабочей памятью времени исполне-  
ния («стек вызовов»). Короткий и красивый алгоритм с использованием  
стека представлен поиском в глубину (DFS) на древовидной или графо-  
вой структуре данных.**

**Python поставляется с несколькими реализациями стека, каждая из  
которых имеет слегка отличающиеся характеристики. Сейчас мы их рас-  
смотрим и сравним их характеристики.**

**list — простые встроенные стеки**

**Встроенный в Python тип list создает нормальную стековую структуру  
данных, поскольку он поддерживает операции вталкивания и выталкива-  
ния за амортизируемое 0(1) время1.**

**На внутреннем уровне списки Python реализованы как динамические  
массивы, а значит, при добавлении или удалении элементов им время от  
времени нужно изменять пространство оперативной памяти для храня-  
щихся в них элементов. Список выделяет избыточную резервную память,  
с тем чтобы не каждая операция вталкивания и выталкивания требовала  
изменения размера памяти, и, как результат, для этих операций вы полу-  
чаете амортизируемую временную сложность 0(1).**

**Недостаток же состоит в том, что это делает показатели их производитель-  
ности менее надежными, чем стабильные вставки и удаления с временной  
сложностью 0(1), которые обеспечиваются реализацией на основе связ-  
ного списка (такого, как collections.deque, см. ниже). С другой стороны,  
списки реально обеспечивают быстрый (со временем 0(1)) произвольный  
доступ к элементам в стеке, и это может быть дополнительным преиму-  
ществом.**

**Используя списки в качестве стеков, необходимо учитывать одно важное  
предостережение относительно производительности.**

**Чтобы получить производительность с амортизируемым временем 0(1)  
для вставок и удалений, новые элементы должны добавляться в конец  
списка методом append() и снова удалятся из конца методом pop(). Для  
оптимальной производительности стеки на основе списков Python долж-  
ны расти по направлению к более высоким индексам и сжиматься к более  
низким.**

**Добавление и удаление элементов в начале списка намного медленнее  
и занимает 0(n) времени, поскольку существующие элементы должны  
сдвигаться, чтобы создать место для нового элемента. Такого антишаблона  
производительности следует избегать.**

**>>> s = []**

**>>> s.append('ecTb')**

**>>> s.append('cnaTb')**

**>>> s.append('программировать')**

**>>> s  
['есть'**

**'спать', 'программировать']**

**>>> s.pop()**

***'программировать'***

**>>> s.pop()**

***'спать'***

**>>> s.pop()**

***'есть'***

**>>> s.pop()**

**IndexErrar: "pop from empty list"**

**collections . deque — быстрые и надежные стеки**

**Класс deque реализует очередь с двусторонним доступом, которая под-  
держивает добавление и удаление элементов с любого конца за 0(1)  
(неамортизируемое) время. Поскольку двусторонние очереди одинаково  
хорошо поддерживают добавление и удаление элементов с любого конца,  
они могут служить и в качестве очередей, и в качестве стеков[[54]](#footnote-54).**

**Объекты Python deque реализованы как двунаправленные связные спи-  
ски, что дает им стабильную производительность для операций вставки  
и удаления элементов, но при этом плохую 0(n) производительность для  
произвольного доступа к элементам в середине очереди[[55]](#footnote-55).**

**В целом двусторонняя очередь collections.deque - отличный выбор,  
если вы ищете стековую структуру данных в стандартной библиотеке  
Python, которая обладает характеристиками производительности, ана-  
логичными реализации на основе связного списка.**

**>>> from collections import deque  
>>> s = deque()**

**>>> s.append('ecTb')**

**>>> s.append('cnaTb')**

**>>> s.append('программировать')**

**>>> s**

**deque(['ecTb', 'спать', 'программировать'])**

**>>> s.pop()**

***'программировать'***

**>>> s.pop()**

***'спать'***

**>>> s.pop()**

***'есть'***

**>>> s.pop()**

**IndexErrar: "pop from an empty deque"**

**deque . LifoQueue — семантика блокирования  
для параллельных вычислений**

**Данная реализация стека в стандартной библиотеке Python синхрони-  
зирована и обеспечивает семантику блокирования с целью поддержки  
многочисленных параллельных производителей и потребителей1.**

**Помимо LifoQueue, модуль queue содержит несколько других классов,  
которые реализуют очереди с мультипроизводителями/мультипотреби-  
телями, широко используемые в параллельных вычислениях.**

**В зависимости от вашего варианта использования семантика блокирова-  
ния может оказаться полезной, а может накладывать ненужные издержки.  
В этом случае в качестве стека общего назначения лучше всего использо-  
вать список list или двустороннюю очередь deque.**

**>>> from queue import LifoQueue  
>>> s = LifoQueue()**

**>>> s.put('ecTb')**

**>>> s.put('cnaTb')**

**>>> s.put('программировать')**

**>>> s**

**<queue.LifoQueue object at 0x108298dd8>**

**>>> s.get()**

***'программировать'***

***>>> s.get()***

***'спать'***

***>>> s.get()***

***'есть'***

**>>> s.get\_nowait()  
queue.Empty**

**>>> s.get()**

***# Блокирует / ожидает бесконечно...***

**Сравнение реализаций стека в Python**

**Как вы убедились, Python поставляется с несколькими реализациями  
стековой структуры данных. Все они обладают слегка различающимися  
характеристиками, а также компромиссным соотношением производи-  
тельности и применения.**

**Если вам не нужна поддержка параллельной обработки (или вы не хотите  
обрабатывать блокировку и снятие блокировки вручную), то ваш выбор  
сводится к встроенному типу list или collections.deque. Разница лежит  
в используемой за кадром структуре данных и общей простоте использо-  
вания:**

**□ Список list поддерживается динамическим массивом, который делает  
его отличным выбором для быстрого произвольного доступа, но при  
этом требует нерегулярного изменения размеров во время добавления  
или удаления элементов. Список выделяет излишнюю резервную  
память, чтобы не каждая операция вталкивания и выталкивания тре-  
бовала изменения размеров, и для этих операций вы получаете амор-**

**тизируемую временную сложность 0(1). Однако вам следует быть  
внимательными и стараться выполнять вставку и удаление элементов  
«с правильной стороны», используя методы append() и pop(). В про-  
тивном случае производительность замедлится до 0(n).**

* **Двусторонняя очередь collections.deque поддерживается двунаправ-  
  ленным связным списком, который оптимизирует добавления и уда-  
  ления с обоих концов и обеспечивает для этих операций стабильную  
  производительность 0(1). Производительность класса deque не только  
  стабильнее, но его также легче использовать, потому что вам не при-  
  ходится переживать по поводу добавления или удаления элементов  
  «не с того конца».**

**Резюмируя, я полагаю, что двусторонняя очередь collections.deque**

**представляет собой отличный вариант для реализации стека (очереди**

**LIF0) на Python.**

**Ключевые выводы**

* **Python поставляется с несколькими реализациями стека, которые  
  обладают слегка различающимися характеристиками производитель-  
  ности и особенностями использования.**
* **Двусторонняя очередь collections.deque обеспечивает безопасную  
  и быструю реализацию стека общего пользования.**
* **Встроенный тип list может применяться в качестве стека, но следует  
  соблюдать осторожность и добавлять и удалять элементы только при  
  помощи методов append() и pop(), чтобы избежать замедления произ-  
  водительности.**

1. **. Очереди (с дисциплиной доступа FIFO)**

**В этом разделе вы увидите, как реализовывать очередь, то есть структуру  
данных с дисциплиной доступа FIFO, используя только встроенные типы  
данных и классы из стандартной библиотеки Python. Но сначала давайте  
вкратце повторим, что такое очередь.**

**Очередь представляет собой коллекцию объектов, которая поддерживает  
быструю семантику доступа «первым пришел — первым ушел» (FIFO — first  
in, first out) для вставок и удалений. Операции вставки и удаления иногда  
называются поставить в очередь (enqueue) и убрать из очереди (dequeue).  
В отличие от списков или множеств, очереди, как правило, не допускают  
произвольного доступа к объектам, которые они содержат.**

**Ниже приведена аналогия для очереди с дисциплиной доступа «первым  
пришел — первым ушел» из реального мира:**

**Представьте очередь разработчиков-питонистов, ожидающих получе-  
ния значка участника конференции в день регистрации на PyCon . По  
мере прибытия новых участников к месту проведения конференции они  
выстраиваются в очередь, «становясь в ее конец», чтобы получить свои  
значки . Удаление (обслуживание) происходит в начале очереди, когда  
разработчики получают свои значки и пакет с материалами и подарками  
конференции и покидают очередь**

**Еще один способ запомнить особенности структуры данных очередь со-  
стоит в том, чтобы представить ее как конвейер:**

**Новые элементы (молекулы воды, теннисные мячи, \_) вставляются  
в одном конце и проходят в другой, где вы или кто-то другой их все  
время удаляете . Когда элементы находятся в очереди (в твердой ме-  
таллической трубе), вы не можете до них добраться . Единственный спо-  
соб взаимодействия с элементами из очереди заключается в том, чтобы  
добавлять новые элементы в конец (ставить в очередь) или удалять  
элементы из начала конвейера (убирать из очереди) .**

**Очереди похожи на стеки, и разница между ними в том, как удаляются  
элементы.**

**В случае с очередью вы удаляете элемент, который был добавлен в нее  
раньше всех (принцип «первым пришел — первым ушел», или FIFO); одна-  
ко в случае со стеком вы удаляете элемент, который был добавлен в него  
позже всех (принцип «последним пришел — первым ушел», или LIFO).**

**С точки зрения производительности предполагается, что надлежащая  
реализация очереди будет занимать 0(1) времени на операции вставки  
и удаления. Эти две выполняемые с очередью операции являются глав-**

**ными, и при правильной реализации они обеспечивают высокое быстро-  
действие.**

**Очереди находят широкое применение в алгоритмах и нередко помогают  
решать задачи планирования и параллельного программирования. Корот-  
кий и красивый алгоритм с использованием очереди представлен поис-  
ком в ширину (breadth-first search, BFS) на древовидной или графовой  
структуре данных.**

**В алгоритмах планирования выполнения задач во внутреннем представ-  
лении нередко используются очереди с приоритетом. Они представля-  
ют собой специализированные очереди: вместо получения следующего  
элемента по времени вставки очередь с приоритетом получает элемент  
с самым высоким приоритетом. Приоритет отдельных элементов опреде-  
ляется очередью, основанной на примененном к их ключам упорядочении.  
В следующем разделе мы обратимся к очередям с приоритетом и рассмо-  
трим ближе, как они реализуются в Python.**

**Однако в обычной очереди содержащиеся в ней элементы не переупоря-  
дочиваются. Точно так же, как и в примере с конвейером, «вы получите  
только то, что вы вставили», и именно в таком порядке.**

**Python поставляется с несколькими реализациями очереди, каждая из ко-  
торых обладает несколько различающимися характеристиками. Давайте  
их рассмотрим.**

**list — ужасно меееедленная очередь**

**В качестве очереди можно использовать обычный список, но с точки  
зрения производительности такое решение не идеально1. Списки для  
этой цели довольно медленные, потому что вставка в начало очереди или  
удаление элемента влекут за собой сдвиг всех других элементов на одну  
позицию, требуя O(n) времени.**

**Поэтому я не рекомендую использовать список в качестве импровизиро-  
ванной очереди в Python (если только вы не имеете дело с небольшим  
количеством элементов).**

**>>> q = []**

**>>> q.append('ecTb')**

**>>> q.append('cnaTb')**

**>>> q.append('программировать')**

**>>> q**

**['есть', 'спать', 'программировать']**

***# Осторожно: это очень медленная операция!*>>> q.pop(0)**

***'есть'***

**collections . deque — быстрые и надежные очереди**

**Класс deque реализует очередь с двусторонним доступом, которая под-  
держивает добавление и удаление элементов с любого конца за 0(1)  
(неамортизируемое) время. Поскольку двусторонние очереди одинаково  
хорошо поддерживают добавление и удаление элементов с любого конца,  
они могут служить в качестве очередей и в качестве стеков[[56]](#footnote-56).**

**Объекты Python deque реализованы как двунаправленные связные списки  
(doubly-linked lists)[[57]](#footnote-57). Это придает им превосходную и стабильную произ-  
водительность для операций вставки и удаления элементов, но при этом  
плохую 0(n) производительность для произвольного доступа к элементам  
в середине очереди.**

**Как результат, двусторонняя очередь collections.deque будет хорошим  
выбором, если вы ищете структуру данных очередь в стандартной библио-  
теке Python.**

**>>> from collections import deque  
>>> q = deque()**

**>>> q.append('ecTb')**

**>>> q.append('cnaTb')**

**>>> q.append('программировать')**

**>>> q**

**deque(['ecTb', 'спать', 'программировать'])**

**>>> q.popleft()**

***'есть'***

**>>> q.popleft()**

***'спать'***

**>>> q.popleft()**

***'программировать'***

**>>> q.popleft()**

**IndexErrar: "pop from an empty deque"**

**queue . Queue — семантика блокирования  
для параллельных вычислений**

**Данная реализация очереди в стандартной библиотеке Python синхро-  
низирована и обеспечивает семантику блокирования с целью поддержки  
многочисленных параллельных производителей и потребителей[[58]](#footnote-58).**

**Модуль queue содержит несколько других классов, которые реализуют  
очереди с мультипроизводителями/мультипотребителями, которые ши-  
роко используются в параллельных вычислениях.**

**В зависимости от вашего варианта использования семантика блокирова-  
ния может оказаться полезной, а может накладывать ненужные издержки.  
В этом случае в качестве очереди общего назначения лучше всего исполь-  
зовать двустороннюю очередь collections.deque.**

**>>> from queue import Queue  
>>> q = Queue()**

**>>> q.put('ecTb')**

**>>> q.put('cnaTb')**

**>>> q.put('программировать')  
>>> q**

**<queue.Queue object at 0x1070f5b38>  
>>> q.get()**

***есть***

**>>> q.get()  
'спать'**

***>>> q.get()***

***'программировать'***

**>>> q.get\_nowait()  
queue.Empty**

**>>> q.get()**

***# Блокирует / ожидает бесконечно...***

**multiprocessing . Queue — очереди совместных заданий**

**Такая реализация очереди совместных заданий позволяет выполнять па-  
раллельную обработку находящихся в очереди элементов многочисленны-  
ми параллельными рабочими процессами[[59]](#footnote-59). Процессно-ориентированное  
распараллеливание популярно в Python из-за глобальной блокировки  
интерпретатора (GIL), которая препятствует некоторым формам парал-  
лельного исполнения в единственном процессе интерпретатора.**

**В качестве специализированной реализации очереди, предназначенной  
для обмена данными между процессами, очередь multiprocessing.Queue  
упрощает распределение работы по многочисленным процессам с целью  
преодоления ограничений GIL. Этот тип очереди может хранить и пере-  
давать любой консервируемый (модулем pickle) объект через границы  
процессов.**

**>>> from multiprocessing import Queue  
>>> q = Queue()**

**>>> q.put('ecTb')**

**>>> q.put('cnaTb')**

**>>> q.put('программировать')**

**>>> q**

**<multiprocessing.queues.Queue object at 0x1081c12b0>**

**>>> q.get()**

**'есть'**

**>>> q.get()**

**'спать'**

**>>> q.get()**

**'программировать'**

**>>> q.get()**

***# Блокирует / ожидает бесконечно...***

**Ключевые выводы**

* **Python содержит несколько реализаций очередей в качестве составной  
  части ядра языка и его стандартной библиотеки.**
* **Объекты-списки list могут использоваться в качестве очередей, но  
  это обычно не рекомендуется делать из-за низкой производитель-  
  ности.**
* **Если вы не ищете поддержку параллельной обработки, то реализация,  
  предлагаемая очередью collections.deque, является превосходным  
  вариантом по умолчанию для реализации в Python структуры данных  
  с дисциплиной доступа FIFO, то есть очереди. Она обеспечивает ха-  
  рактеристики производительности, которые можно ожидать от хоро-  
  шей реализации очереди, а также может применяться в качестве стека  
  (очереди с дисциплиной доступа LIFO).**

1. **Очереди с приоритетом**

**Очередь с приоритетом представляет собой контейнерную структуру дан-  
ных, которая управляет набором записей с полностью упорядоченными  
ключами[[60]](#footnote-60) (например, числовым значением веса) с целью обеспечения бы-  
строго доступа к записи с наименьшим или наибольшим ключом в наборе.**

**Очередь с приоритетом можно представить как видоизмененную очередь:  
вместо получения следующего элемента по времени вставки она получает  
элемент с самым высоким приоритетом. Приоритет отдельных элементов  
определяется примененным к их ключам упорядочением.**

**Очереди с приоритетом широко используются для решения задач пла-  
нирования, например предоставления предпочтений задачам с более  
высокой актуальностью.**

**Представьте работу планировщика задач операционной системы:**

**В идеальном случае высокоприоритетные задачи в системе (например,  
игра в компьютерную игру в реальном времени) должны иметь пред-  
почтение перед задачами с более низким приоритетом (например, ска-  
чивание обновлений в фоновом режиме) . Организовывая предстоящие  
задачи в очередь с приоритетом, которая использует актуальность за-  
дачи в качестве ключа, планировщик задач может быстро выбирать за-  
дачи с самым высоким приоритетом и давать им выполняться в первую  
очередь**

**В этом разделе вы увидите несколько вариантов реализации очередей  
с приоритетом в Python с помощью встроенных структур данных либо  
структур данных, которые поставляются вместе со стандартной библио-  
текой Python. Каждая реализация будет иметь свои собственные преиму-  
щества и недостатки, но, по моему мнению, в каждом распространенном  
сценарии есть свой победитель. Давайте узнаем, что лучше.**

**list — поддержание сортируемой очереди вручную**

**Вы можете использовать сортированный список list, который позволяет  
быстро идентифицировать и удалять наименьший или наибольший эле-  
мент. Недостатком является то, что вставка новых элементов в список  
является медленной O(n) операцией.**

**Несмотря на то что точка вставки может быть найдена за O(log n) время  
с помощью алгоритма bisect.insort1 стандартной библиотеки, это реше-  
ние всегда находится во власти медленного шага вставки.**

**Поддержание упорядоченности путем добавления в конец списка и пере-  
сортировки также занимает минимум O(n log n) времени. Еще один недо-  
статок — вам придется вручную заботиться о пересортировке списка во  
время вставки новых элементов. Пропустив этот шаг, можно легко внести  
ошибки, и ответственность за них всегда будет на вас как на разработчике.**

**Поэтому я убежден, что сортированные списки подходят как очереди  
с приоритетом только в тех случаях, когда вставок немного.**

**q = []**

**q.append((2, 'программировать'))  
q.append((1, 'есть'))  
q.append((3, 'спать'))**

* **ПРИМЕЧАНИЕ; *Не забудьте выполнить пересортировку всякий раз,***
* ***когда добавляется новый элемент, либо используйте***
* **bisect.insort().  
  q.sort(reverse=True)**

**while q:**

**next\_item = q.pop()  
print(next\_item)**

|  |  |
| --- | --- |
| **# Результат;** | |
| **# (1,** | **'есть')** |
| **# (2,** | **'программировать')** |
| **# (3,** | **'спать')** |

**heapq — двоичные кучи на основе списка**

**Данная реализация двоичной кучи обычно подкрепляется обыкновенным  
списком, и она поддерживает вставку и извлечение наименьшего элемента  
за O(log n) время[[61]](#footnote-61) [[62]](#footnote-62).**

**Этот модуль — хороший выбор для реализации очередей с приоритетом  
в Python. Поскольку двоичная куча heapq технически обеспечивает толь-  
ко реализацию min-heap (то есть кучи, где значение в любой вершине не  
больше, чем значения ее потомков), должны быть предприняты допол-  
нительные шаги, которые обеспечат стабильность сортировки и другие  
функциональные возможности, которые, как правило, ожидают от «прак-  
тической версии» очереди с приоритетом[[63]](#footnote-63).  
import heapq**

**q = []**

**heapq.heappush(q, (2, 'программировать'))  
heapq.heappush(q, (1, 'есть'))  
heapq.heappush(q, (3, 'спать'))**

**while q:**

**next\_item = heapq.heappop(q)  
print(next\_item)**

* ***Результат:***
* ***(1, 'есть')***
* ***(2, 'программировать')***
* ***(3, 'спать')***

**queue . PriorityQueue — красивые очереди с приоритетом**

**Данная реализация очереди с приоритетом во внутреннем представлении  
использует двоичную кучу heapq и имеет одинаковую временную и про-  
странственную вычислительную сложность[[64]](#footnote-64).**

**Разница состоит в том, что очередь с приоритетом PriorityQueue синхро-  
низирована и обеспечивает семантику блокирования с целью поддержки  
многочисленных параллельных производителей и потребителей.**

**В зависимости от вашего варианта использования она либо станет по-  
лезной, либо слегка замедлит вашу программу. В любом случае вы мо-  
жете предпочесть интерфейс на основе класса, предлагаемый классом  
PriorityQueue, использованию интерфейса на основе функций, предла-  
гаемого модулем heapq.**

**from queue import PriorityQueue**

**q = PriorityQueue()**

**q.put((2, 'программировать'))  
q.put((1, 'есть'))  
q.put((3, 'спать'))**

**while not q.empty():**

**next\_item = q.get()  
print(next\_item)**

|  |  |
| --- | --- |
| **# Результат:** | |
| **# (1,** | **'есть')** |
| **# (2,** | **'программировать')** |
| **# (3,** | **'спать')** |

**Ключевые выводы**

* **Python содержит несколько реализаций очередей с приоритетом, ко-  
  торые вы можете использовать в своих программах.**
* **Реализация queue.PriorityQueue выбивается из общего ряда, отлича-  
  ясь хорошим объектно-ориентированным интерфейсом и именем, ко-  
  торое четко указывает на ее направленность. Такая реализация должна  
  быть предпочтительным вариантом.**
* **Если требуется избежать издержек, связанных с блокировкой очере-  
  ди queue.PriorityQueue, то непосредственное использование моду-  
  ля heapq также будет хорошим выбором.**

1. **. 1 . Написание питоновских циклов**

**Один из самых легких способов отличить разработчика с опытом работы  
на C-подобных языках, который совсем недавно перешел на Python, — по-  
смотреть, как он пишет циклы.**

**Например, всякий раз, когда я вижу фрагмент кода, который выглядит,  
как показано ниже, сразу понимаю, что тут пытались программировать на  
Python так, будто это C или Java:**

**my\_items = ['a', 'b', 'c']**

**i = 0 while i < len(my\_items):  
print(my\_items[i])  
i += 1**

**Итак, вы спрашиваете, что же такого непитоновского в этом фрагменте  
кода?**

**Две вещи.**

**Во-первых, в коде вручную отслеживается индекс i — его инициализация  
нулем, а затем постепенное увеличение после каждой итерации цикла.**

**И во-вторых, в коде используется функция len(), которая получает раз-  
мер контейнера my\_items, чтобы определить количество итераций.**

**В Python можно писать циклы, которые справляются с этими двумя за-  
дачами автоматически. И будет просто замечательно, если вы возьмете это  
на вооружение. Например, если вашему коду не придется отслеживать на-  
растающий индекс, то будет намного труднее написать непреднамеренный  
бесконечный цикл. Это также сделает программный код более сжатым  
и поэтому удобочитаемым.**

**Чтобы рефакторизовать первый пример кода, я начну с того, что удалю  
фрагмент, который вручную обновляет индекс. В Python лучше всего для  
этого применить цикл for. При помощи встроенной фабричной функции  
range() я могу генерировать индексы автоматически:**

**>>> range(len(my\_items))  
range(0, 3)**

**>>> list(range(0, 3))**

**[0, 1, 2]**

**Тип range представляет неизменяемую последовательность чисел. Его  
преимущество перед обычным списком list в том, что он всегда занимает  
одинаково небольшое количество оперативной памяти. Объекты-диапа-  
зоны в действительности не хранят отдельные значения, представляющие  
числовую последовательность, вместо этого они функционируют как  
итераторы и вычисляют значения последовательности на ходу[[65]](#footnote-65).**

**Поэтому, вместо того чтобы на каждой итерации цикла вручную увели-  
чивать индекс i, я смог воспользоваться функцией range() и написать  
что-то подобное:**

**for i in range(len(my\_items)):  
print(my\_items[i])**

**Уже лучше. Однако этот вариант по-прежнему выглядит не совсем по-  
питоновски и ощущается больше как итеративная Java-конструкция, а не  
как настоящий цикл Python. Когда вы видите программный код, в кото-**

**ром для итеративного обхода контейнера используется range(len(.  
его, как правило, можно еще больше упростить и улучшить.**

**)),**

**Как я уже отмечал, циклы for в Python в действительности являются  
циклами «for each», которые могут выполнять непосредственный пере-  
бор элементов контейнера или последовательности без необходимости  
искать их по индексу. И этот факт я могу задействовать для дальнейшего  
упрощения этого цикла:**

**for item in my\_items:  
print(item)**

**Я считаю такое решение вполне питоновским. В нем применено несколько  
продвинутых функциональных средств Python, но при этом оно остается  
хорошим и чистым и читается почти как псевдокод из учебника по про-  
граммированию. Обратите внимание, что в этом цикле больше не отсле-  
живается размер контейнера, а для доступа к элементам не используется  
нарастающий индекс.**

**Теперь контейнер сам занимается раздачей элементов для их обработки.  
Если контейнер упорядочен, то и результирующая последовательность  
элементов будет такой же. Если контейнер не упорядочен, он будет воз-  
вращать свои элементы в произвольном порядке, но цикл по-прежнему  
охватит их все полностью.**

**Нужно сказать, что, конечно, вы не всегда будете в состоянии переписать  
свои циклы таким образом. А что, если, например, вам нужен индекс  
элемента?**

**Для таких случаев есть возможность писать циклы, которые поддержива-  
ют нарастающий индекс, избегая применения шаблона с range(len(...)),  
от которого я вас предостерег. Встроенная функция enumerate() поможет  
вам сделать подобного рода циклы безупречными и питоновскими:**

**>>> for i, item in enumerate(my\_items):**

**... print(f'{i}: {item}')**

**0: a  
1: b  
2: c**

**Дело в том, что итераторы в Python могут возвращать более одного зна-  
чения. Они могут возвращать кортежи с произвольным числом значений,  
которые затем могут быть распакованы прямо внутри инструкции for.**

**Это очень мощное средство. Например, тот же самый прием можно исполь-  
зовать, чтобы в цикле одновременно перебрать ключи и значения словаря:**

**>>> emails = {**

**... 'Боб': 'bob@example.com',**

**... 'Алиса': 'alice@example.com',**

**... }**

**>>> for name, email in emails.items():**

**... print(f'{name} -> {email}')**

***'Боб -> bob@exampLe.com'***

***'Алиса -> aLice@exampLe.com'***

**Есть еще один пример, который я хотел бы вам показать. Что, если вам  
совершенно точно нужно написать C-подобный цикл? Например, если  
вам требуется управлять размером шага индекса? Предположим, что вы  
начали со следующего цикла Java:**

**for (int i = a; i < n; i += s) {**

**// ...**

**}**

**Как этот шаблон перевести на Python? И снова на выручку приходит  
функция range() — она принимает необязательные параметры, которые  
управляют начальным значением (a), конечным значением (n) и размером  
шага (s) цикла. Перевод с Java на Python будет выглядеть так:**

**for i in range(a, n, s):**

**# ...**

**Ключевые выводы**

**□ Написание C-подобных циклов на Python считается непитоновским  
стилем. Если это возможно, следует избегать ручного управления ин-  
дексами цикла и условиями остановки.**

**□ Циклы for в Python в действительности являются циклами «for each»,  
которые могут напрямую перебирать элементы контейнера или по-  
следовательности.**

**6 . 2 . Осмысление включений**

**Одно из моих любимых функциональных средств языка Python — вклю-  
чения в список[[66]](#footnote-66). На первый взгляд эта конструкция может показаться  
немного загадочной, но когда вы разложите ее по полочкам, она окажется  
очень простой.**

**Ключ к пониманию конструкций включения в список состоит в том, что  
они попросту являются циклами с обходом коллекции, выраженными при  
помощи более сжатого и компактного синтаксиса.**

**Такие синтаксические конструкции, или синтаксический сахар, — неболь-  
шая краткая форма для часто используемой функциональности, которая  
делает нашу программистскую питоновскую жизнь легче. В качестве  
примера возьмем приведенное ниже включение в список:**

**>>> squares = [x \* x for x in range(10)]**

**В нем вычисляется квадрат всех чисел в списке от нуля до девяти:**

**>>> squares**

**[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]**

**Если бы вы хотели построить тот же самый список, использовав обыкно-  
венный цикл for, то вы, вероятно, написали бы что-то типа этого:**

**>>> squares = []**

**>>> for x in range(10):**

**... squares.append(x \* x)**

**Довольно-таки прямолинейный цикл, не правда ли? Если вы вернетесь  
и сопоставите пример с включением в список и версию с циклом for,  
то заметите общие черты, и в конечном счете у вас появятся некоторые  
шаблоны. Обобщив здесь часть общей структуры, вы в итоге придете  
к шаблону, похожему на следующий:**

**values = [expression for item in collection]**

**Приведенный выше «шаблон» включения в список эквивалентен пред-  
ставленному ниже обыкновенному циклу for:**

**values = []**

**for item in collection:**

**values.append(expression)**

**Здесь мы сначала настраиваем новый экземпляр списка list, который  
получит выходные значения. Затем мы выполняем обход всех значений  
в контейнере, преобразовывая каждый из них при помощи произволь-  
ного выражения, и затем добавляем отдельные результаты в выходной  
список.**

**Мы имеем типовой шаблон в стиле «формы для печенья», который вы  
можете применять ко многим циклам for. Этот шаблон предназначен  
для преобразования циклов в конструкцию включения в список, и на-  
оборот. Нужно сказать, что есть еще одно полезное дополнение, которое  
мы должны внести в этот шаблон, а именно — фильтрация элементов по  
условиям.**

**Включения в список могут фильтровать значения, основываясь на некоем  
произвольном условии, которое определяет, становится результирующее  
значение частью выходного списка или нет. Приведем пример:**

**>>> even\_squares = [x \* x for x in range(10)  
if x % 2 == 0]**

**Данное включение в список вычислит список квадратов всех четных  
целых чисел от нуля до девяти. Использованный здесь оператор остатка  
(%) возвращает остаток после деления одного числа на другое. В данном  
примере мы его используем, чтобы проверить, является ли число четным.  
И оно имеет требуемый результат:**

**>>> even\_squares [0, 4, 16, 36, 64]**

**Новое включение в список может быть преобразовано в эквивалентный  
цикл for аналогично первому примеру:**

**even\_squares = [] for x in range(10):  
if x % 2 == 0:**

**even\_squares.append(x \* x)**

**Давайте попробуем еще слегка обобщить указанный выше шаблон, где  
включение в список трансформируется в цикл for. На этот раз мы соби-  
раемся добавить в наш шаблон фильтрующее условие, которое определяет,  
какие значения попадут в выходной список. Вот обновленный шаблон  
включения в список:**values = [expression

**for item in collection  
if condition]**

**И снова, это включение в список можно преобразовать в цикл for с по-  
мощью следующего ниже шаблона:**

**values = [] for item in collection:  
if condition:**

**values.append(expression)**

**В очередной раз это преобразование было прямолинейным — мы просто  
применили обновленный типовой шаблон. Надеюсь, все это рассеяло  
часть «магии», связанной с тем, как работают включения в список. Они  
представляют собой полезный инструмент, который все программирую-  
щие на Python разработчики должны уметь применять.**

**Прежде чем мы пойдем дальше, хочу подчеркнуть, что Python поддер-  
живает не только включение в список. В нем также имеется аналогичный  
синтаксический сахар для множеств и словарей.**

Вот как выглядит ***включение в множество:***

**>>> { x \* x for x in range(-9, 10) }  
set([64, 1, 36, 0, 49, 9, 16, 81, 25, 4])**

**В отличие от списков, которые сохраняют порядок следования в них  
элементов, множества Python имеют тип неупорядоченных коллекций.  
Поэтому, когда вы будете добавлять элементы в контейнер множества  
set, вы будете получать более-менее «случайный» порядок следования.**

А вот ***включение в словарь:***

**>>> { x: x \* x for x in range(5) }**

**{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}**

**Оба включения являются весьма полезными инструментами на практике.  
Правда, относительно включений следует сделать одно предостережение:  
по мере накопления опыта их применения станет все легче и легче писать  
трудночитаемый программный код. Если вы не будете осторожны, то  
вскоре вам, возможно, придется столкнуться с чудовищными включени-  
ями в список, в множество и в словарь. Следует помнить, что слишком  
много хорошего — тоже плохо.**

**После долгих разочарований лично я для включений ставлю черту под од-  
ним уровнем вложенности. Я обнаружил, что за этими границами в боль-  
шинстве случаев лучше (имея в виду «более легкую удобочитаемость»  
и «более легкое сопровождение») использовать циклы for.**

**Ключевые выводы**

* **Включения в список, в множество и в словарь являются ключевым  
  функциональным средством языка Python. Их понимание и примене-  
  ние сделают ваш программный код намного более питоновским.**
* **Конструкции включения попросту являются причудливым синтаксиче-  
  ским сахаром для шаблона с простым циклом for. Как только вы разбе-  
  ретесь в этом шаблоне, то разовьете интуитивное понимание включений.**
* **Помимо включений в список есть и другие виды включений.**

**6.3 . Нарезки списков и суши-оператор**

**В Python объекты-списки имеют замечательное функциональное сред-  
ство, которое называется нарезкой (slicing). Его можно рассматривать как  
расширение синтаксиса индексации с использованием квадратных скобок.  
Нарезка широко используется для доступа к диапазонам (интервалам)  
элементов внутри упорядоченной коллекции. Например, с его помощью  
большой объект-список можно нарезать на несколько меньших по раз-  
меру подсписков.**

**Приведу пример. В операции нарезки используется знакомый синтаксис  
индексации «[]» со следующим шаблоном "[начало:конец:шаг]»:**

**>>> 1st = [1, 2, 3, 4, 5]**

**>>> 1st**

**[1, 2, 3, 4, 5]**

***# Lst[начало:конец:шаг]***

**>>> 1st[1:3:1]**

**[2, 3]**

**Добавление индекса [1:3:1] вернуло срез оригинального списка, начиная  
с индекса 1 и заканчивая индексом 2, с размером шага, равным одному  
элементу. Чтобы избежать ошибок смещения на единицу, важно помнить,  
что верхняя граница всегда не учитывается. Именно поэтому в качестве  
подсписка из среза [1:3:1] мы получили [2, 3].**

**Если убрать размер шага, то он примет значение по умолчанию, равное  
единице:**

**>>> 1st[1:3]**

**[2, 3]**

**С параметром шага, который также называется сдвигом (stride), можно  
делать другие интересные вещи. Например, можно создать подсписок,  
который включает каждый второй элемент оригинала:**

**>>> 1st[::2]  
[1, 3, 5]**

**Здорово, правда? Мне нравится называть оператор «:» суши-оператором.  
Выглядит как восхитительный маки-ролл, разрезанный пополам. Помимо  
того что он напоминает вкусное блюдо и получает доступ к диапазонам  
списка, у него есть еще несколько менее известных применений. Давайте  
покажу еще пару забавных и полезных трюков с нарезкой списка!**

**Вы только что увидели, как размер шага нарезки может использоваться  
для отбора каждого второго элемента списка. Ну хорошо. Вот вам еще хи-  
трость: если запросить срез [::-1] , то вы получите копию оригинального  
списка, только в обратном порядке:**

**>>> numbers[::-1]**

**[5, 4, 3, 2, 1]**

**Мы запросили Python дать нам весь список (::), но при этом чтобы он  
пробежался по всем элементам с конца в начало, назначив размер шага  
равным -1. Довольно ловко, но в большинстве случаев для того, чтобы  
инвертировать список, я по-прежнему придерживаюсь метода list.  
reverse() и встроенной функции reversed.**

**Вот другой трюк с нарезкой списка: оператор «:» можно использовать  
для удаления всех элементов из списка, не разрушая сам объект-список.**

**Это очень полезно, когда необходимо очистить список в программе,  
в которой имеются другие указывающие на него ссылки. В этом случае  
нередко вы не можете просто опустошить список, заменив его на новый  
объект-список, поскольку эта операция не будет обновлять другие ссылки  
на этот список. И тут на выручку приходит суши-оператор:**

**>>> 1st = [1, 2, 3, 4, 5]**

**>>> del 1st[:]**

**>>> 1st  
[]**

**Как видите, этот фрагмент удаляет все элементы из 1st, но оставляет сам  
объект-список неповрежденным. В Python 3 для выполнения такой же  
работы также можно применить метод 1st.c1ear(), который в зависимо-  
сти от обстоятельств, возможно, будет более удобочитаемым шаблоном.  
Однако имейте в виду, что метод c1ear() отсутствует в Python 2.**

**Помимо очистки списков, нарезку также можно использовать для замены  
всех элементов списка, не создавая новый объект-список. Это чудесная  
сокращенная запись для очистки списка и затем повторного его заполне-  
ния вручную:**

**>>> original\_lst = 1st  
>>> lst[:] = [7, 8, 9]**

**>>> lst  
[7, 8, 9]**

**>>> origina1\_1st  
[7, 8, 9]**

**>>> origina1\_1st is 1st  
True**

**Приведенный выше пример кода заменил все элементы в 1st, но не унич-  
тожил и воссоздал список как таковой. По этой причине старые ссылки  
на оригинальный объект-список по-прежнему действительны.**

**И еще один вариант использования суши-оператора — создание (мелких)  
копий существующих списков:**

**>>> copied\_lst = lst[:]**

**>>> copied\_1st  
[7, 8, 9]**

**>>> copied\_1st is 1st  
Fa1se**

**Создание мелкой копии означает, что копируется только структура эле-  
ментов, но не сами элементы. Обе копии списка совместно используют  
одинаковые экземпляры отдельных элементов.**

**Если необходимо продублировать абсолютно все, включая и элементы, то  
необходимо создать глубокую копию списка. Для этой цели пригодится  
встроенный модуль Python copy.**

**Ключевые выводы**

**□ Суши-оператор «:» полезен не только для отбора подсписков эле-  
ментов внутри списка. Он также может использоваться для очистки,  
реверсирования и копирования списков.**

**□ Но следует быть осторожным — для многих разработчиков Python эта  
функциональность граничит с черной магией. Ее применение может  
сделать исходный код менее легким в сопровождении для всех осталь-  
ных коллег в вашей команде.**

1. **. Красивые итераторы**

**Мне нравится то, как синтаксис Python отличается своей красотой и яс-  
ностью от других языков программирования. Например, давайте возьмем  
скромный цикл for-in. Красота Python говорит сама за себя — вы можете  
прочитать приведенный ниже питоновский цикл, как если бы это было  
английское предложение:**

**numbers = [1, 2, 3]  
for n in numbers:  
print(n)**

**Но как элегантные циклические конструкции Python работают за кадром?  
Каким образом этот цикл достает отдельные элементы из объекта, итера-  
ции по которому он выполняет? И как можно поддерживать одинаковый  
стиль программирования в собственных объектах Python?**

**Ответы на эти вопросы можно найти в протоколе итератора Python: объ-  
екты, которые поддерживают дандер-методы iter и next , автома-**

**тически работают с циклами for-in.**

**Однако вникнем во все шаг за шагом. Точно так же, как и декораторы,  
итераторы и связанные с ними методы на первый взгляд могут показаться  
довольно загадочными и сложными. Поэтому мы будем входить в курс  
дела постепенно.**

**В этом разделе вы увидите, как написать несколько классов Python, ко-  
торые поддерживают протокол итератора. Они послужат в качестве «не-  
магических» примеров и тестовых реализаций, на основе которых можно  
укрепить и углубить свое понимание.**

**Прежде всего мы сосредоточимся на ключевых механизмах итераторов  
в Python 3 и опустим любые ненужные сложности, чтобы вы четко уви-  
дели поведение итераторов на фундаментальном уровне.**

**Я свяжу все примеры с вопросом о цикле for-in, с которого мы начали  
этот раздел. И в его конце мы пробежимся по некоторым различиям, су-  
ществующим между Python 2 и Python 3 относительно итераторов.**

**Готовы? Тогда, поехали!**

**Бесконечное повторение**

**Начнем с того, что напишем класс, который демонстрирует скелетный  
протокол итератора. Используемый здесь пример, возможно, по виду от-  
личается от примеров, которые вы видели в других пособиях по итерато-  
рам, но наберитесь терпения. Считаю, что в таком виде он предоставит вам  
более компетентное понимание того, как итераторы работают в Python.**

**В последующих нескольких абзацах мы собираемся реализовать класс, ко-  
торый мы назовем повторителем Repeater, итерации по которому можно  
выполнять в цикле for-in следующим образом:**

**repeater = Repeater('Привет')  
for item in repeater:  
print(item)**

**Как следует из его имени, экземпляры класса Repeater при его итератив-  
ном обходе будут неизменно возвращать единственное значение. Поэтому  
приведенный выше пример кода будет бесконечно печатать в консоли  
строковый литерал 'Привет'.**

**Начиная реализацию, мы, прежде всего, определим и конкретизируем  
класс Repeater:**

**class Repeater:**

**def init (self, value):**

**self.value = value**

**def \_\_iter\_\_(self):**

**return RepeaterIterator(self)**

**При первоначальном осмотре класс Repeater похож на заурядный класс  
Python. Но обратите внимание, что он также включает метод \_\_iter\_\_.**

**Что за объект Repeaterlterator мы создаем и возвращаем из дандер-мето-**

**да iter ? Это вспомогательный класс, который нам нужно определить,**

**чтобы заработал наш пример итераций в цикле for^in:**

**class Repeaterlterator:**

**def init (self, source):**

**self.source = source**

**def next (self):**

**return self.source.value**

**И снова, RepeaterIterator похож на прямолинейный класс Python, но,  
возможно, вам стоит принять во внимание следующие две вещи:**

1. **В методе init мы связываем каждый экземпляр класса Repeater-**

**Iterator с объектом Repeater, который его создал. Благодаря этому  
мы можем держаться за «исходный» объект, итерации по которому  
выполняются.**

1. **В RepeaterIterator. next мы залезаем назад в «исходный» экзем-**

**пляр класса Repeater и возвращаем связанное с ним значение.**

**В этом примере кода Repeater и RepeaterIterator работают вместе, что-  
бы поддерживать протокол итератора Python. Два определенных нами**

**дандер-метода, init и next , являются центральными в создании**

**итерируемого объекта Python.**

**Мы рассмотрим ближе эти два метода и то, как они работают вместе,  
после того, как немного поэкспериментируем с кодом, который у нас  
есть сейчас.**

**Давайте подтвердим, что эта конфигурация с двумя классами действитель-  
но сделала объекты класса Repeater совместимыми с итерацией в цикле  
for^in. Для этого мы сначала создадим экземпляр класса Repeater, который  
будет бесконечно возвращать строковый литерал 'Привет':**

**>>> repeater = Repeater('npMBej')**

**И теперь попробуем выполнить итерации по объекту repeater в цикле  
for^in. Что произойдет, когда вы выполните приведенный ниже фрагмент  
кода?**

**>>> for item in repeater:**

**... print(item)**

**Точно! Вы увидите, как на экране будет напечатано 'Привет' \_ много раз.  
Объект repeater продолжает возвращать то же самое строковое значение,  
и этот цикл никогда не завершится. Наша небольшая программа обречена  
печатать в консоли 'Привет' до бесконечности:**

**Привет**

**Привет**

**Привет**

**Привет**

**Привет**

**И тем не менее примите поздравления — вы только что написали работа-  
ющий итератор на Python и применили его в цикле for^in. Этот цикл все  
еще не может завершиться^ но пока что все идет неплохо!**

**Теперь мы разделим этот пример на части, чтобы понять, как мето-  
ды init и next работают вместе, делая объект Python итерируемым.**

**Профессиональный совет: если вы выполнили предыдущий пример в се-  
ансе Python REPL или в терминале и хотите его остановить, нажмите со-  
четание клавиш Ctrl + C несколько раз, чтобы выйти из бесконечного цикла.**

**Как циклы for-in работают в Python?**

**На данном этапе у нас есть класс Repeater, который, несомненно, поддер-  
живает протокол итератора, и мы просто выполнили цикл for^in, чтобы  
это доказать:**

**repeater = Repeater('Привет')  
for item in repeater:  
print(item)**

**Итак, что же этот цикл for^in в действительности делает за кадром? Как  
он контактирует с объектом repeater, чтобы доставать из него новые  
элементы?**

**Чтобы рассеять часть этого «волшебства», мы можем расширить цикл  
в слегка удлиненном фрагменте кода, который дает тот же самый резуль-  
тат:**repeater = Repeater('Привет')

**iterator = repeater. iter ()**

**while True:**

**item = iterator. next ()**

**print(item)**

**Как видите, конструкция for^in была всего лишь синтаксическим саха-  
ром для простого цикла while:**

**□ Этот фрагмент кода сначала подготовил объект repeater к итерации,  
вызвав его метод iter . Он вернул фактический объект-итератор.**

**next объекта-ите-**

**□ После этого цикл неоднократно вызывал метод \_  
ратора, чтобы извлекать из него значения.**

**Если вы когда-либо работали с курсорами базы данных (database cursors),  
то эта ментальная модель будет выглядеть похожей: мы сначала иници-  
ализируем курсор и готовим его к чтению, а затем можем доставлять из  
него данные, один элемент за другим, в локальные переменные в нужном  
объеме.**

**Поскольку «в активном состоянии» никогда не находится более одного  
элемента, этот подход чрезвычайно эффективен с точки зрения потребля-  
емой оперативной памяти. Наш класс Repeater обеспечивает бесконечную  
последовательность элементов, и мы можем без проблем выполнять по  
нему итерации. Имитация того же самого при помощи списка Python  
list была бы невозможной — прежде всего, нет никакой возможности  
создать список с бесконечным количеством элементов. И это превращает  
итераторы в очень мощную концепцию.**

**Говоря более абстрактно, итераторы обеспечивают единый интерфейс,  
который позволяет вам обрабатывать каждый элемент контейнера, оста-  
ваясь полностью изолированным от внутренней структуры последнего.**

**Имеете ли вы дело со списком элементов, словарем, бесконечной после-  
довательностью, например такой, которая обеспечивается нашим классом**

**Repeater, или другим типом последовательности — все это просто детали  
реализации. Эти объекты все до единого можно проходить таким же об-  
разом при помощи мощных возможностей итераторов.**

**Как вы убедились, в Python нет ничего особенного в циклах for^in. Если  
вы заглянете за кулисы, то увидите, что все сводится к вызову правильных  
дандер-методов в нужное время.**

**На самом деле в сеансе интерпретатора Python можно вручную «эмули-  
ровать» то, как цикл использует протокол итератора:**

**>>> repeater = Repeater('Привет')**

**>>> iterator = iter(repeater)**

**>>> next(iterator)**

***'Привет'***

**>>> next(iterator)**

***'Привет'***

**>>> next(iterator)**

***'Привет'***

**Этот фрагмент кода дает тот же самый результат — бесконечный поток  
приветствий. Всякий раз, когда вы вызываете next(), итератор снова вы-  
дает то же самое приветствие.**

**Между прочим, здесь я воспользовался возможностью замены вызо-  
вов iter и next на вызовы встроенных в Python функций iter()**

**и next().**

**На внутреннем уровне эти встроенные функции вызывают те же самые  
дандер-методы, но они делают программный код немного симпатичнее  
и более удобочитаемым, предоставляя протоколу итератора чистый «фа-  
сад».**

**Python предлагает эти фасады также и для другой функциональности. На-  
пример, len(x) является краткой формой для вызова x. len . Точно так**

**же вызов функции iter(x) вызывает метод x. iter , а вызов функции**

**next(x) вызывает метод x. next .**

**В целом неплохая идея использовать встроенные фасадные функции,  
вместо того чтобы непосредственно обращаться к дандер-методам, реа-  
лизующим протокол итератора. Это намного упрощает восприятие ис-  
ходного кода.**

**Более простой класс-итератор**

**До этого момента наш пример итератора состоял из двух отдельных клас-  
сов, Repeater и Repeaterlterator. Они соответствовали непосредственно  
двум фазам, используемым в протоколе итератора Python: сначала подго-  
товке и получению объекта-итератора через вызов функции iter(), а затем  
неоднократной доставке из него значений через вызов функции next().**

**Во многих случаях обе эти функциональные обязанности можно взва-  
лить на один-единственный класс. Это позволит сократить объем про-  
граммного кода, необходимого для написания итератора, основанного  
на классах.**

**Я решил этого не делать с первым примером в данном разделе, потому что  
это внесло бы путаницу в чистоту ментальной модели в основе протокола  
итератора. Но теперь, когда вы увидели, как писать итератор на основе  
классов более долгим и более сложным способом, давайте потратим еще  
минуту, чтобы упростить то, что у нас есть на данный момент.**

**Помните, почему нам вновь потребовался класс Repeaterlterator? Он**

**был нужен, чтобы принять метод next для доставки новых значений**

**из итератора. Но место определения метода next вовсе не имеет ни-**

**какого значения. В протоколе итератора имеет значение только то, что**

**метод iter возвращает любой объект с определенным на нем методом**

**next .**

**Поэтому идея такая: Repeaterlterator без конца возвращает одинаковое  
значение, и он не должен отслеживать никакое внутреннее состояние.**

**Что, если вместо этого добавить метод next непосредственно в класс**

**Repeater?**

**Тем самым мы смогли бы целиком избавиться от Repeaterlterator и реа-  
лизовать итерируемый объект при помощи одного-единственного класса  
Python. Давайте попробуем! Наш пример с новым и упрощенным итера-  
тором выглядит так:**class Repeater:

**def init (self, value):**

**self.value = value**

**def iter (self):**

**return self**

**def next (self):**

**return self.value**

**Мы только что перешли от двух отдельных классов и десяти строк кода  
всего к одному классу и семи строкам кода. Наша упрощенная реализация  
по-прежнему без проблем поддерживает протокол итератора:**

**>>> repeater = Repeater('Привет')**

**>>> for item in repeater:**

**... print(item)**

**Привет**

**Привет**

**Привет**

**В подобной оптимизации итератора на основе класса часто есть смысл. По  
сути, большинство пособий Python по итераторам начинается именно так.  
Но я всегда чувствовал, что объяснять итераторы одним-единственным  
классом с самого начала — значит скрывать основные принципы прото-  
кола итератора и по этой причине еще больше затруднять его понимание.**

**Кто же захочет без конца выполнять итерации**

**На этом этапе у вас уже должно сложиться довольно хорошее понимание  
того, как итератор работает в Python. Но пока что мы реализовывали  
только такие итераторы, которые продолжают выполнять итерации бес-  
конечно.**

**Очевидно, бесконечное повторение не является главным вариантом ис-  
пользования итераторов в Python. На самом деле, когда вы обратитесь  
к самому началу этого раздела, то увидите, что в качестве мотивирующего  
примера я использовал приведенный ниже фрагмент кода:**

**numbers = [1, 2, 3]  
for n in numbers:  
print(n)**

**Вы вправе ожидать, что этот код выведет числа 1, 2 и 3, а затем остано-  
вится. И вероятно, вы не ожидаете, что он захламит окно вашего терми-  
нала, без устали выводя «3», пока вы в дикой панике не начнете жать  
на Ctrl+C\_**

**Пора узнать, как написать итератор, который в итоге прекращает гене-  
рировать новые значения вместо выполнения бесконечных итераций,  
потому что это именно то, что обычно делают объекты Python, когда мы  
используем их в цикле for^in.**

**Сейчас мы напишем еще один класс итератора, который назовем ограни-  
ченным повторителем BoundedRepeater. Он будет похож на наш предыду-  
щий пример с повторителем Repeater, но на этот раз мы хотим, чтобы он  
останавливался после предопределенного количества повторений.**

**Давайте задумаемся. Как это сделать? Как итератор сигнализирует о том,  
что он пуст и исчерпал элементы, выдаваемые во время выполнения  
итераций? Возможно, вы думали: «Хм, можно вернуть None из метода  
 next , и все».**

**И знаете, это неплохая идея, но проблема в следующем: что делать, если  
нам нужно, чтобы некий итератор был в состоянии возвращать None в ка-  
честве приемлемого значения?**

**Давайте посмотрим, что для решения этой проблемы делают другие  
итераторы Python. Я создам простой контейнер, список с несколькими  
элементами, а затем буду выполнять его итеративный обход до тех пор,  
пока он не исчерпает элементы, чтобы увидеть, что произойдет:**

**>>> my\_list = [1, 2, 3]**

**>>> iterator = iter(my\_list)**

**>>> next(iterator)**

**1**

**>>> next(iterator)**

**2**

**>>> next(iterator)**

**3**

**А теперь осторожно! Мы употребили все три имеющихся в списке элемента.  
Следите за тем, что произойдет, если еще раз вызвать метод next итератора:**

**>>> next(iterator)**

***StopIteration***

**Ага! Чтобы подать сигнал о том, что мы исчерпали все имеющиеся в ите-  
раторе значения, он вызывает исключение StopIteration.**

**Все верно: итераторы используют исключения для структуризации по-  
тока управления. Чтобы подать сигнал о завершении итераций, итератор  
Python просто вызывает встроенное исключение StopIteration.**

**Если я продолжу запрашивать значения из итератора, он продолжит вы-  
зывать исключения StopIteration, сигнализируя о том, что больше нет  
значений, доступных для итераций:**

**>>> next(iterator)**

**StopIteration  
>>> next(iterator)**

***StopIteration***

**Итераторы Python обычно не могут быть «обнулены» — как только они  
исчерпаны, им полагается вызывать исключение StopIteration при каж-  
дом вызове их функции next(). Чтобы возобновить итерации, вам нужно  
запросить свежий объект-итератор при помощи функции iter().**

**Теперь мы знаем все, что нужно для написания нашего класса  
BoundedRepeater, который прекращает итерации после заданного коли-  
чества повторений:**

**class BoundedRepeater:**

**def init (self, value, max\_repeats):**

**self.value = value  
self.max\_repeats = max\_repeats**

**self.count = 0**

**def iter (self):**

**return self**

**def next (self):**

**if self.count >= self.max\_repeats:  
raise StopIteration  
self.count += 1  
return self.value**

**И он дает нам требуемый результат. Итерации прекращаются после ряда  
повторений, определенных в параметре max\_repeats:**

**>>> repeater = BoundedRepeater('Привет', 3)**

**>>> for item in repeater:  
print(item)**

**Привет**

**Привет**

**Привет**

**Если переписать этот последний пример цикла for^in, устранив часть  
синтаксического сахара, то в итоге мы получим следующий ниже рас-  
ширенный фрагмент кода:**

**repeater = BoundedRepeater('Привет', 3)  
iterator = iter(repeater)  
while True:  
try:**

**item = next(iterator)  
except StopIteration:  
break**

**print(item)**

**При каждом вызове функции next() в этом цикле мы выполняем про-  
верку на исключение StopIteration и при необходимости выходим из  
цикла while.**

**Возможность написать трехстрочный цикл for^in вместо восьмистроч-  
ного цикла while представляет собой вполне хорошее улучшение. И в ре-  
зультате программный код становится проще для восприятия и удобнее  
в сопровождении. И это еще одна причина, почему в Python итераторы  
являются таким мощным инструментом.**

**Совместимость с Python 2 . x**

**Все примеры кода, которые я здесь показал, были написаны на Python 3.  
Существует одна небольшая, но важная разница между Python 2  
и Python 3 в том, что касается реализации итераторов на основе класса:**

* **в Python 3 метод, который извлекает следующее значение из итератора,**

**называется next ;**

* **в Python 2 тот же самый метод называется next (без символов под-  
  черкивания).**

**Эта разница в обозначении может привести к небольшой проблеме при  
попытке писать итераторы на основе класса, которые должны работать  
в обеих версиях Python. К счастью, существует простой подход, который  
можно применить, чтобы обойти эту разницу.**

**Ниже приведена обновленная версия класса InfiniteRepeater, который  
будет работать как в Python 2, так и в Python 3:**

**class InfiniteRepeater(object):**

**def init (self, value):**

**self.value = value**

**def iter (self):**

**return self**

**def next (self):**

**return self.value**

***# Совместимость с Python 2:*def next(self):**

**return self. next ()**

**Чтобы сделать этот класс-итератор совместимым с Python 2, я внес в него  
два небольших изменения.**

**Во-первых, я добавил метод next, который просто вызывает оригиналь-  
ный метод next и пересылает возвращаемое из него значение. По су-**

**ществу, тем самым создается псевдоним для существующей реализации  
метода next для того, чтобы его нашел Python 2. Благодаря этому мы**

**можем поддерживать обе версии Python, при этом сохраняя все фактиче-  
ские детали реализации в одном месте.**

**И во-вторых, я модифицировал определение класса, и теперь он наследует  
от object, чтобы обеспечить создание класса Python 2 в новом стиле. Это  
изменение не имеет никакого отношения к итераторам, что совершенно  
понятно, но, тем не менее, является хорошей практикой.**

**Ключевые выводы**

* **Итераторы предоставляют объектам Python интерфейс последователь-  
  ности, который эффективен с точки зрения потребляемой оперативной  
  памяти и который считается чисто питоновским. Любуйтесь красотой  
  цикла for ... in!**
* **Чтобы поддерживать итерации, в объекте должен быть реализован**

**протокол итератора за счет обеспечения дандер-методов iter**

**и next .**

* **Итераторы на основе класса являются лишь одним из способов на-  
  писания итерируемых объектов в Python. Следует также рассмотреть  
  генераторы и выражения-генераторы.**

1. **. Генераторы — это  
   упрощенные итераторы**

**В разделе, посвященном итераторам, мы потратили довольно много  
времени на написание итератора на основе класса. Это было неплохой  
идеей с точки зрения обучения, но итератор на основе класса также  
продемонстрировал, что написание класса итератора требует большо-  
го объема шаблонного кода. И если говорить по правде, то как «лени-  
вому» разработчику мне не нравится утомительная и однообразная  
работа.**

**И все же итераторы очень полезны в Python. Они позволяют писать  
симпатичные циклы for^in и помогают делать код более питоновским**

**и эффективным^ если бы только не существовало более удобного способа  
писать эти итераторы изначально.**

**Сюрприз! Вот же он! В который раз Python нас выручает, предлагая еще  
немного синтаксического сахара, чтобы облегчить написание итераторов.  
В этом разделе вы увидите, как писать итераторы быстрее и с меньшим  
объемом кода, используя генераторы и ключевое слово yield.**

**Бесконечные генераторы**

**Давайте начнем с того, что посмотрим еще раз на пример с классом  
Repeater, который я уже использовал, чтобы познакомить вас с идеей  
итераторов. В нем реализована итеративная обработка бесконечной по-  
следовательности значений на основе класса. Вот так этот класс выглядел  
в своей второй (упрощенной) версии:**

**class Repeater:**

**def init (self, value):**

**self.value = value**

**def iter (self):**

**return self**

**def next (self):**

**return self.value**

**Если вы думаете, что «для такого простого итератора тут довольно много  
исходного кода», то вы абсолютно правы. Некоторые части этого класса  
кажутся довольно стереотипными, как будто они переносились под ко-  
пирку с одного итератора на основе класса на другой.**

**И вот где на сцену выходят генераторы Python. Если я перепишу этот  
класс итератора в качестве генератора, то он будет выглядеть так:**

**def repeater(value):  
while True:**

**yield value**

**Мы только что перешли от семи строк кода к трем. Неплохо, правда? Как  
видите, генераторы похожи на обычные функции, но вместо инструкции  
возврата return в них для передачи данных назад источнику вызова ис-  
пользуется инструкция yield.**

**Будет ли эта новая реализация генератора по-прежнему работать так  
же, как и наш итератор на основе класса? Давайте стряхнем пыль с теста  
в цикле for^in, чтобы это выяснить:**

**>>> for x in гереаіег('Привет'):**

**... print(x)**

***'Привет'***

***'Привет'***

***'Привет'***

***'Привет'***

***'Привет'***

**Да! Мы по-прежнему без конца прокручиваем в цикле наши приветствия.  
Эта намного более короткая реализация генератора, по всей видимости,  
выполняется таким же образом, что и класс Repeater. (Не забудьте нажать  
Ctrl+C, если хотите выйти из бесконечного цикла в сеансе интерпретатора.)**

**Итак, каким же образом эти генераторы работают? Они похожи на нор-  
мальные функции, но их поведение очень различается. Начнем с того, что  
вызов функции-генератора вообще не выполняет функцию. Он просто  
создает и возвращает объект-генератор:**

**>>> гереаіег('Эй')**

**<generator object repeater at 0x107bcdbf8>**

**Программный код в функции-генератора исполняется только тогда,  
когда функция next() вызывается с объектом-генератором в качестве  
аргумента:**

**>>> generator\_obj = repeater('Эй')**

**>>> next(generator\_obj)**

***'Эй'***

**Если вы еще раз прочитаете код функции repeater, то увидите, что, судя  
по всему, ключевое слово yield каким-то образом останавливает эту  
функцию-генератор посередине исполнения, а затем возобновляет ее на  
более позднем этапе:**

**def repeater(value):  
while True:**

**yield value**

**И это вполне подходящая ментальная модель того, что здесь происходит.  
Дело в том, что, когда инструкция return вызывается внутри функции,  
она безвозвратно передает управление назад источнику вызова функции.  
Когда же вызывается инструкция yield, она тоже передает управление  
назад источнику вызова функции — но она это делает лишь временно.**

**В отличие от инструкции return, которая избавляется от локального  
состояния функции, инструкция yield приостанавливает функцию и со-  
храняет ее локальное состояние. На практике это означает, что локальные  
переменные и состояние исполнения функции-генератора лишь откла-  
дываются в сторону и не выбрасываются полностью. Исполнение может  
быть возобновлено в любое время вызовом функции next() с генератором  
в качестве аргумента:**

**>>> iterator = repeater('Привет')**

**>>> next(iterator)**

***'Привет'***

**>>> next(iterator)**

***'Привет'***

**>>> next(iterator)**

***'Привет'***

**Это делает генераторы полностью совместимыми с протоколом итератора.  
По этой причине мне нравится представлять их прежде всего как синтак-  
сический сахар для реализации итераторов.**

**Вы убедитесь, что в отношении большинства типов итераторов написание  
функции-генератора будет проще, а восприятие легче, чем определение  
многословного итератора на основе класса.**

**Генераторы, которые прекращают генерацию**

**Этот раздел мы начали с того, что еще раз написали бесконечный генера-  
тор. Сейчас вы, вероятно, задаетесь вопросом, как написать генератор,  
который через некоторое время прекращает порождать значения вместо  
того, чтобы без конца продолжать это делать.**

**Напомним, что в нашем итераторе на основе класса мы смогли подать  
сигнал об окончании итераций путем вызова исключения StopIteration  
вручную. Поскольку генераторы полностью совместимы с итераторами  
на основе класса, за сценой будет по-прежнему происходить то же самое.**

**К счастью, на этот раз мы будем работать с более приятным интерфейсом.  
Генераторы прекращают порождать значения, как только поток управле-  
ния возвращается из функции-генератора каким-либо иным способом,  
кроме инструкции yield. Это означает, что вам больше вообще не нужно  
заботиться о вызове исключения StopIteration!**

**Приведу пример:**

**def repeat\_three\_times(value):  
yield value  
yield value  
yield value**

**Обратите внимание: эта функция-генератор не содержит никакого цикла.  
В действительности она проста как божий день и состоит всего из трех  
инструкций yield. Если yield временно приостанавливает выполнение  
функции и передает значение назад источнику вызова, то что произойдет,  
когда мы достигнем конца этого генератора? Давайте узнаем:**

**>>> for x in repeat\_three\_times('BceM привет'):**

**... print(x)**

***'Всем привет'***

***'Всем привет'***

***'Всем привет'***

**Как вы, возможно, и ожидали, этот генератор прекратил порождать новые  
значения после трех итераций. Можно предположить, что он это сделал  
путем вызова исключения StopIteration, когда исполнение достигло  
конца функции. Но чтобы быть до конца уверенными, давайте подтвердим  
это еще одним экспериментом:**

**>>> iterator = repeat\_three\_times('BceM привет')**

**>>> next(iterator)**

***'Всем привет'***

**>>> next(iterator)  
'Всем привет'**

**>>> next(iterator)**

***'Всем привет'***

**>>> next(iterator)**

***StopIteration  
>>>* next(iterator)**

***StopIteration***

**Этот итератор вел себя именно так, как мы и ожидали. Как только мы до-  
стигаем конца функции-генератора, он начинает вызывать StopIteration,  
сигнализируя о том, что у него больше нет значений, которые он мог бы  
предоставить.**

**Давайте вернемся к еще одному примеру из раздела об итераторах. Класс  
BoundedIterator реализовал итератор, который будет повторять значение,  
заданное определенное количество раз:**

**class BoundedRepeater:**

**def init (self, value, max\_repeats):**

**self.value = value  
self.max\_repeats = max\_repeats  
self.count = 0**

**def \_\_iter\_\_(self):  
return self**

**def \_\_next\_\_(self):**

**if self.count >= self.max\_repeats:  
raise StopIteration  
self.count += 1  
return self.value**

**Почему бы не попробовать реализовать класс BoundedRepeater заново как  
функцию-генератор? Сделаю первую попытку:**

**def bounded\_repeater(value, max\_repeats):  
count = 0  
while True:**

**if count >= max\_repeats:  
return  
count += 1  
yield value**

**Я преднамеренно сделал цикл while в этой функции несколько громозд-  
ким. Я хотел продемонстрировать, как вызов инструкции return из ге-  
нератора приводит к остановке итераций с исключением StopIteration.  
Мы вскоре подчистим и еще немного упростим эту функцию-генератор,  
но сначала давайте испытаем то, что у нас есть сейчас:**

**>>> for x in bounded\_repeater('Привет', 4):**

**... print(x)**

***'Привет'***

***'Привет'***

***'Привет'***

***'Привет'***

**Великолепно! Теперь у нас есть генератор, который прекращает по-  
рождать значения после настраиваемого количества повторений. Он  
использует инструкцию yield, чтобы передавать значения назад до тех  
пор, пока он наконец не натолкнется на инструкцию return и итерации  
не прекратятся.**

**Как я вам обещал, мы можем упростить этот генератор еще больше. Мы  
воспользуемся тем, что в конец каждой функции Python добавляет не-  
явную инструкцию return None. И вот как будет выглядеть наша оконча-  
тельная реализация:**

**def bounded\_repeater(value, max\_repeats):  
for i in range(max\_repeats):  
yield value**

**Не стесняйтесь подтвердить, что этот упрощенный генератор по-  
прежнему работает таким же образом. Учитывая все обстоятельства, мы  
прошли путь от 12-строчной реализации в классе BoundedRepeater до  
трехстрочной реализации на основе генератора, обеспечив ту же самую  
функциональность. А это 75 %-ное сокращение количества строк кода —  
нехило!**

**Как вы только что убедились, генераторы помогают «абстрагироваться  
от» большей части шаблонного кода, который в других обстоятельствах  
был бы необходим во время написания итераторов на основе класса. Они  
способны очень облегчить вашу программистскую жизнь и позволяют  
писать более чистые, короткие и удобные в сопровождении итераторы.  
Функции-генераторы представляют собой отличное функциональное  
средство языка Python, и вам следует решительно и смело использовать  
их в своих собственных программах.**

**Ключевые выводы**

* **Функции-генераторы являются синтаксическим сахаром для написа-  
  ния объектов, которые поддерживают протокол итератора. Генераторы  
  абстрагируются от большей части шаблонного кода, необходимого во  
  время написания итераторов на основе класса.**
* **Инструкция yield позволяет временно приостанавливать исполнение  
  функции-генератора и передавать из него значения назад.**
* **Генераторы начинают вызывать исключения StopIteration после того,  
  как поток управления покидает функцию-генератор каким-либо иным  
  способом, кроме инструкции yield.**

1. **. Выражения-генераторы**

**По мере того как я все больше узнавал о протоколе итератора Python  
и различных способах его реализации в собственном коде, я стал пони-  
мать, что синтаксический сахар является повторяющейся темой.**

**Дело в том, что итераторы на основе класса и функции-генераторы вы-  
ражают один и тот же лежащий в основе шаблон проектирования.**

**Функции-генераторы предоставляют краткую форму для поддержки  
протокола итератора в своем собственном коде и по большей части  
избегают многословности итераторов на основе класса. Благодаря не-  
значительному объему специализированного синтаксиса или «горсти»  
синтаксического сахара, они экономят время и облегчают вашу жизнь  
как разработчика.**

**В Python и в других языках программирования эта тема довольно часто  
повторяется. Вместе с увеличением числа разработчиков, которые приме-  
няют шаблоны проектирования в своих программах, у создателей языков  
растет стимул предлагать абстракции и укороченные пути их реализации.**

**Именно так происходит эволюция языков, и, как разработчики, мы полу-  
чаем от этого выгоду. Мы приступаем к работе со все более и более мощ-  
ными структурными блоками, которые сокращают бесполезную рутину  
и позволяют достигать большего за меньшее время.**

**Ранее в этой книге вы увидели, как генераторы предлагают синтаксиче-  
ский сахар для написания итераторов на основе класса. Выражения-гене-  
раторы (generator expressions), которые мы рассмотрим в этом разделе,  
добавят сверху еще один слой синтаксического сахара.**

**Выражения-генераторы представляют собой еще более эффективную  
краткую форму для создания итераторов. Благодаря простому и сжато-  
му синтаксису, который похож на конструкцию включения в список, вы  
сможете определять итераторы в одной строке кода.**

**Приведу пример:**

**iterator = ('Привет' for i in range(3))**

**Во время выполнения итераций данное выражение-генератор порождает  
ту же самую последовательность значений, что и функция-генератор  
bounded\_repeater, которую мы написали в предыдущем разделе. Ниже  
привожу ее снова, чтобы освежить вашу память:**

**def bounded\_repeater(value, max\_repeats):  
for i in range(max\_repeats):  
yield value**

**iterator = bounded\_repeater('Привет', 3)**

**Разве не удивительно, что однострочное выражение-генератор теперь де-  
лает работу, для выполнения которой ранее требовалась четырехстрочная  
функция-генератор или намного более длинный итератор на основе класса?**

**Но я бегу впереди паровоза. Давайте убедимся, что наш итератор, опре-  
деленный при помощи выражения-генератора, действительно работает  
как ожидалось:**>>> iterator = ('Привет' for i in range(3))

**>>> for x in iterator:**

**... print(x)**

***'Привет'***

***'Привет'***

***'Привет'***

**Как по мне, смотрится весьма неплохо! Из нашего однострочного выра-  
жения-генератора мы, похоже, получили те же самые результаты, которые  
мы получали из функции-генератора bounded\_repeater.**

**Правда, есть одно маленькое предостережение: после того как выраже-  
ние-генератор было использовано, оно не может быть перезапущено или  
использовано снова. Поэтому в некоторых случаях предпочтительнее  
использовать функции-генераторы или итераторы на основе класса.**

**Выражения-генераторы против включений в список**

**Как вы уже поняли, выражения-генераторы несколько напоминают вклю-  
чения в список:**

**>>> listcomp = ['Привет' for i in range(3)]**

**>>> genexpr = ('Привет' for i in range(3))**

**Однако в отличие от включений в список выражения-генераторы не  
конструируют объекты-списки. Вместо этого они генерируют значения  
«точно в срок» подобно тому, как это сделал бы итератор на основе класса  
или функция-генератор.**

**Присваивая выражение-генератор переменной, вы просто получите ите-  
рируемый «объект-генератор»:**

**>>> listcomp**

**['Привет', 'Привет', 'Привет']  
>>> genexpr**

**<generator object <genexpr> at 0x1036c3200>**

**Для того чтобы получить доступ к значениям, порожденным выражением-  
генератором, вам нужно вызвать с ним метод next() точно так же, как вы  
бы сделали с любым другим итератором:**

**>>> next(genexpr)**

***'Привет'***

**>>> next(genexpr)**

***'Привет'***

**>>> next(genexpr)**

***'Привет'***

**>>> next(genexpr)**

***StopIteration***

**Как вариант, вы также можете вызвать функцию list() c выражением-  
генератором, в результате чего вы сконструируете объект-список, содер-  
жащий все произведенные значения:**

**>>> genexpr = ('Привет' for i in range(3))**

**>>> list(genexpr)**

**['Привет', 'Привет', 'Привет']**

**Разумеется, это был всего лишь игрушечный пример, который показывает,  
как можно «преобразовывать» выражение-генератор (или любой другой  
итератор, если уж на то пошло) в список. Если же вам нужен объект-спи-  
сок прямо на месте, то в большинстве случаев вы с самого начала просто  
пишете включение в список.**

**Давайте рассмотрим синтаксическую структуру этого простого выраже-  
ния-генератора поближе. Шаблон, который вы должны увидеть, выглядит  
следующим образом:**

**genexpr = (expression for item in collection)**

**Приведенный выше «образец» выражения-генератора соответствует сле-  
дующей ниже функции-генератору:**

**def generator():**

**for item in collection:  
yield expression**

**Точно так же, как и с включением в список, он дает вам типовой шаблон  
в стиле «формы для печенья», который можно применять ко многим  
функциям-генераторам с целью их преобразования в сжатые выражения-  
генераторы.**

**Фильтрация значений**

**В этот шаблон можно добавить еще одно полезное дополнение, и это  
фильтрация элемента по условиям. Приведем пример:**

**>>> even\_squares = (x \* x for x in range(10)  
if x % 2 == 0)**

**Данный генератор порождает квадрат всех четных целых чисел от нуля  
до девяти. Фильтрующее условие с использованием оператора остатка %  
(оператора модуля) отклонит любое значение, которое не делится на два:**

**>>> for x in even\_squares:**

**... print(x) 0  
4  
16  
36  
64**

**Давайте обновим наш шаблон выражения-генератора. После добавления  
фильтрации элементов посредством условия if шаблон выглядит так:**

**genexpr = (expression for item in collection  
if condition)**

**И снова этот шаблон соответствует относительно прямолинейной, но бо-  
лее длинной функции-генератору. Синтаксический сахар в своих лучших  
проявлениях:**

**def generator():**

**for item in collection:  
if condition:**

**yield expression**

**Встраиваемые выражения-генераторы**

**Поскольку выражения-генераторы являются, скажем так, выражениями,  
вы можете их использовать в одной строке вместе с другими инструкци-  
ями. Например, вы можете определить итератор и употребить его прямо  
на месте при помощи цикла for:**

**for x in ('Buongiorno' for i in range(3)):  
print(x)**

**Есть и другой синтаксический трюк, который можно использовать для  
того, чтобы сделать выражения-генераторы красивее. Круглые скобки,  
окружающие выражение-генератор, могут быть опущены, если выраже-  
ние-генератор используется в качестве единственного аргумента функции:**

**>>> sum((x \* 2 for x in range(10)))**

**90**

***# Сравните с:***

**>>> sum(x \* 2 for x in range(10))**

**90**

**Это позволяет писать сжатый и высокопроизводительный код. Посколь-  
ку выражения-генераторы генерируют значения «точно в срок» подобно  
тому, как это делает итератор на основе класса или функция-генератор,  
они эффективно используют оперативную память.**

**Слишком много хорошего^**

**Как и включения в список, выражения-генераторы оставляют место для  
большей сложности, чем та, которую мы рассмотрели на данный момент.  
Посредством вложенных циклов for и состыкованных в цепочки формул  
фильтрации они могут охватывать более широкий диапазон вариантов  
использования:**

**(expr for x in xs if condl  
for y in ys if cond2**

**for z in zs if condN)**

**Образец выше переводится в следующую ниже логику функции-генера-  
тора:**

**for x in xs:  
if condl:**

**for y in ys:  
if cond2:**

**for z in zs:  
if condN:**

**yield expr**

**И вот здесь я хотел бы разместить большое предостережение.**

**Пожалуйста, не пишите такие глубоко вложенные выражения-генераторы.  
В дальнейшем окажется, что их будет очень трудно сопровождать.**

**Это одна из тех ситуаций, о которых говорят, что «вещество становится  
ядом, начиная с определенной дозы», где злоупотребление красивым  
и простым инструментом может создать плохо воспринимаемую и трудно  
отлаживаемую программу.**

**Точно так же, как и с включениями в список, лично я стремлюсь избегать  
любого выражения-генератора, которое содержит более двух уровней  
вложенности.**

**Выражения-генераторы являются полезным и питоновским инструмен-  
том в вашем наборе, но это не значит, что они должны использоваться для  
решения каждой задачи, с которой вы сталкиваетесь. В случае составных  
итераторов часто лучше написать функцию-генератор или даже итератор  
на основе класса.**

**Если у вас есть потребность использовать вложенные генераторы и со-  
ставные условия фильтрации, обычно лучше вынести их в подгенераторы  
(чтобы им можно было назначить имя) и затем состыковать их в цепочку  
еще раз, на верхнем уровне. Вы увидите, как это делается, в следующем  
далее разделе, посвященном цепочкам итераторов (iterator chains).**

**Если вы до сих пор не определились, то попробуйте другие реализации,  
а затем выберите ту, которая кажется самой удобочитаемой. Поверьте,  
в итоге это сэкономит вам время.**

**Ключевые выводы**

* **Выражения-генераторы похожи на включения в список. Однако они не  
  конструируют объекты-списки. Вместо этого выражения-генераторы  
  генерируют значения «точно в срок» подобно тому, как это делают  
  итераторы на основе класса или функции-генераторы.**
* **Как только выражение-генератор было использовано, оно не может  
  быть перезапущено или использовано заново.**
* **Выражения-генераторы лучше всего подходят для реализации простых  
  «ситуативных» итераторов. В случае составных итераторов лучше на-  
  писать функцию-генератор или итератор на основе класса.**

1. **Цепочки итераторов**

**Вот еще одно замечательное функциональное свойство итераторов  
в Python: состыковывая многочисленные итераторы в цепочку, можно  
писать чрезвычайно эффективные «конвейеры» обработки данных. Когда  
я впервые увидел этот шаблон в действии на презентации Дэвида Бизли  
в ходе конференции PyCon, то был совершенно потрясен.**

**Если вы воспользуетесь преимуществами функций-генераторов и вы-  
ражений-генераторов Python, то вы в мгновение ока будете строить сжа-  
тые и мощные цепочки итераторов. В этом разделе вы узнаете, как этот  
технический прием выглядит на практике и как вы можете его применять  
в своих собственных программах.**

**В качестве краткого резюме: генераторы и выражения-генераторы пред-  
ставляют собой синтаксический сахар для написания итераторов на  
Python. Они абстрагируются от большей части шаблонного кода, необ-  
ходимого во время написания итераторов на основе класса.**

**В то время как обычная функция производит одно-единственное возвра-  
щаемое значение, генераторы производят последовательность результа-  
тов. Можно сказать, что они генерируют поток значений на протяжении  
своего жизненного цикла.**

**Например, я могу определить следующий ниже генератор, который произ-  
водит серию целочисленных значений от одного до восьми, поддерживая  
нарастающий счетчик и выдавая новое значение всякий раз, когда с ним  
вызывается функция next():**

**def integersQ:**

**for i in range(1, 9):  
yield i**

**Вы можете подтвердить такое поведение, выполнив данный ниже фраг-  
мент кода в интерпретаторе REPL Python:**

**>>> chain = integersQ  
>>> list(chain)**

**[1, 2, 3, 4, 5, 6, 7, 8]**

**Пока что не очень интересно. Но сейчас мы быстро это изменим. Дело  
в том, что генераторы могут быть «присоединены» друг к другу, благодаря  
чему можно строить эффективные алгоритмы обработки данных, которые  
работают как конвейер.**

**Вы можете взять «поток» значений, выходящих из генератора integers(),  
и направить их в еще один генератор. Например, такой, который прини-  
мает каждое число, возводит его в квадрат, а затем передает его дальше:**

**def squared(seq):  
for i in seq:  
yield i \* i**

**Ниже показано, что будет теперь делать наш «конвейер данных», или  
«цепочка генераторов»:**

**>>> chain = squared(integers())**

**>>> list(chain)**

**[1, 4, 9, 16, 25, 36, 49, 64]**

**И мы можем продолжить добавлять в этот конвейер новые структурные  
блоки. Данные текут только в одном направлении, и каждый шаг обработ-  
ки защищен от других четко определенным интерфейсом.**

**Это похоже на то, как работают конвейеры в UNIX. Мы состыковываем  
последовательность процессов в цепочку так, чтобы результат каждого  
процесса подавался непосредственно на вход следующего.**

**Почему бы в наш конвейер не добавить еще один шаг, который инверти-  
рует каждое значение, а потом передает его на следующий шаг обработки  
в цепи:**

**def negated(seq):  
for i in seq:  
yield -i**

**Если мы перестроим нашу цепочку генераторов и добавим negated в ко-  
нец, то вот что мы получим на выходе:**

**>>> chain = negated(squared(integers()))**

**>>> list(chain)**

**[-1, -4, -9, -16, -25, -36, -49, -64]**

**Моя любимая фишка формирования цепочки генераторов состоит в том,  
что обработка данных происходит по одному элементу за один раз. Буфе-  
ризация между шагами обработки в цепочке отсутствует:**

1. **Генератор integers выдает одно-единственное значение, скажем, 3.**
2. **Это значение «активирует» генератор squared, который обрабатывает  
   значение и передает его на следующую стадию как 3 х 3 = 9.**
3. **Квадрат целого числа, выданный генератором squared, немедленно  
   передается в генератор negated, который модифицирует его в -9 и вы-  
   дает его снова.**

**Вы можете продолжать расширять эту цепочку генераторов, чтобы от-  
строить конвейер обработки со многими шагами. Он по-прежнему будет  
выполняться эффективно и может легко быть модифицирован, потому  
что каждым шагом в цепочке является отдельная функция-генератор.**

**Каждая отдельная функция-генератор в этом конвейере обработки до-  
вольно сжатая. С помощью небольшой уловки мы можем сжать опреде-  
ление этого конвейера еще больше, не сильно жертвуя удобочитаемостью:**

**integers = range(8)**

**squared = (i \* i for i in integers)**

**negated = (-i for i in squared)**

**Обратите внимание, как я заменил каждый шаг обработки в цепочке  
на выражение-генератор, строящийся на выходе из предыдущего шага.  
Этот программный код эквивалентен цепочке генераторов, которые мы  
построили в этом разделе выше:**

**>>> negated**

**<generator object <genexpr> at 0x1098bcb48>**

**>>> list(negated)**

**[0, -1, -4, -9, -16, -25, -36, -49]**

**Единственным недостатком применения выражений-генераторов явля-  
ется то, что их не получится сконфигурировать с использованием ар-  
гументов функции и вы не сможете повторно использовать то же самое  
выражение-генератор многократно в том же самом конвейере обработки.**

**Но, безусловно, во время сборки конвейеров вы можете свободно ком-  
бинировать выражения-генераторы и обычные генераторы на свой вкус.  
В случае с составными конвейерами это поможет улучшить удобочита-  
емость.**

**Ключевые выводы**

* **Генераторы могут состыковываться в цепочки, формируя очень эф-  
  фективные и удобные в сопровождении конвейеры обработки данных.**
* **Состыкованные в цепочки генераторы обрабатывают каждый элемент,  
  проходящий сквозь цепь по отдельности.**
* **Выражения-генераторы могут использоваться для написания сжатого  
  определения конвейера, но это может повлиять на удобочитаемость.**

**7. 1 . Значения словаря, принимаемые  
по умолчанию**

**У словарей Python есть метод get() для поиска ключа, которому передают  
запасное значение. Это может пригодиться в самых разных ситуациях.  
Приведу простой пример, который покажет, что я имею в виду. Предпо-  
ложим, что у нас есть представленная ниже структура данных, которая  
ставит идентификаторы в соответствие именам пользователей:**

**name\_for\_userid = {**

**382: 'Элис',**

**950: 'Боб',**

**590: 'Дилберт',**

**}**

**Теперь мы хотели бы использовать эту структуру данных, чтобы напи-  
сать функцию greeting(), которая будет возвращать пользователю при-  
ветствие на основе его идентификатора. Наша первая реализация может  
выглядеть примерно так:**

**def greeting(userid):**

**return 'Привет, %s!' % name\_for\_userid[userid]**

**В ней представлен прямолинейный поиск в словаре. Это первая реализа-  
ция технически работает — но только если идентификатор пользователя**

**является допустимым ключом в словаре name\_for\_userid. Если в функ-  
цию greeting передать недопустимый идентификатор пользователя, то  
она вызовет исключение:**

**>>> greeting(382)**

***'Привет, Элис!'***

**>>> greeting(33333333)**

**КеуВггаг: 33333333**

**Исключение KeyError — это совсем не тот результат, который мы хотели  
бы видеть. Было бы намного лучше, если бы в качестве запасного варианта  
функция возвращала универсальное приветствие, если идентификатор  
пользователя не может быть найден.**

**Давайте реализуем эту идею. Наш первый подход мог бы заключаться  
в простой проверке принадлежности в формате ключ в словаре (key in dict)  
и возврате приветствия по умолчанию, если идентификатор пользователя  
неизвестен:**

**def greeting(userid):**

**if userid in name\_for\_userid:**

**return 'Привет, %s!' % name\_for\_userid[userid]  
else:**

**return 'Привет всем!'**

**Давайте посмотрим, как эта реализация функции greeting() проявит себя  
с нашими предыдущими тестовыми случаями:**

**>>> greeting(382)**

***'Привет, Элис!'***

**>>> greeting(33333333)**

***'Привет всем!'***

**Намного лучше. Мы теперь получаем универсальное приветствие для  
неизвестных пользователей, и мы поддерживаем персонализированное  
приветствие, когда допустимый идентификатор пользователя найден.**

**Но все равно есть простор для совершенствования. Несмотря на то что эта  
новая реализация дает нам ожидаемые результаты и выглядит небольшой  
и чистой, ее все еще можно улучшить. К этому подходу у меня есть не-  
сколько претензий:**

* **он неэффективен, потому что он опрашивает словарь дважды;**
* **он многословен, поскольку, например, часть строки с приветствием по-  
  вторяется;**
* **он не является питоновским — официальная документация Python,  
  в частности, для таких ситуаций рекомендует использовать стиль про-  
  граммирования «легче попросить прощения, чем разрешения» (EAFP):**

**Этот общепринятый стиль программирования на Python исходно предпо-  
лагает существование допустимых ключей или атрибутов и отлавливает  
исключения, если предположение оказывается ложным[[67]](#footnote-67) .**

**Более эффективная реализация, которая следует принципам EAFP, могла  
бы вместо выполнения явной проверки на принадлежность ключа слова-  
рю задействовать блок try^except, чтобы поймать исключение KeyError:**

**def greeting(userid):  
try:**

**return 'Привет, %s!' % name\_for\_userid[userid]  
except KeyError:**

**return 'Привет всем'**

**Эта реализация по-прежнему верна в том, что касается наших первона-  
чальных требований, и теперь мы устранили необходимость опрашивать  
словарь дважды.**

**Но мы до сих пор можем улучшить ее и предложить более чистое решение.  
В словаре Python есть метод get(), поддерживающий параметр «по умол-  
чанию», который можно использовать в качестве запасного значения[[68]](#footnote-68):**

**def greeting(userid):**

**return 'Привет, %s!' % name\_for\_userid.get(  
userid, 'всем')**

**Во время вызова метода get() он проверяет, существует ли заданный  
ключ в словаре. Если это так, то возвращается значение, соответствующее  
этому ключу. Если же он не существует, то вместо этого возвращается  
значение по умолчанию. Как вы видите, эта реализация функции greeting  
по-прежнему работает как надо:**

**>>> greeting(950)**

***'Привет, Боб!'***

**>>> greeting(333333)**

***'Привет, всем!'***

**Наша заключительная реализация функции greeting() является сжатой,  
чистой и использует средства только из стандартной библиотеки Python.  
Поэтому убежден, что для этой конкретной ситуации такое решение яв-  
ляется наилучшим.**

**Ключевые выводы**

* **Во время проверки принадлежности ключа словарю избегайте явных  
  проверок в формате ключ в словаре.**
* **Предпочтительной является обработка исключений в стиле EAFP или  
  использование встроенного метода get().**
* **В некоторых случаях класс collections.defaultdict из стандартной  
  библиотеки также может оказаться полезным.**

1. **. 2 . Сортировка словарей для дела и веселья**

**В словарях Python нет внутренней упорядоченности ключей. Можно без  
проблем выполнять обход словарей, но при этом нет никакой гарантии,  
что итерация возвращает элементы словаря в каком-то определенном по-  
рядке следования (хотя, начиная с Python 3.6, это и меняется).**

**Однако очень часто полезно получить сортированное представление  
(sorted representation) словаря, поместив элементы словаря в произ-  
вольном порядке на основе их ключа, значения или иного производного  
свойства. Предположим, что у вас есть словарь xs со следующими парами  
ключ-значение:**

**>>> xs = {'a': 4, 'c': 2, 'b': 3, 'd': 1}**

**Чтобы получить сортированный список пар ключ-значение в этом сло-  
варе, вы можете применить метод items() словаря и затем отсортировать  
результирующую последовательность на втором обходе:**

**>>> sorted(xs.items())**

**[('a', 4), ('b', 3), ('c', 2), ('d', 1)]**

**Кортежи ключ-значение упорядочены с использованием стандартного  
лексикографического упорядочивания Python для сравнения последо-  
вательностей.**

**Чтобы сравнить два кортежа, Python сначала сравнивает элементы, хра-  
нящиеся в индексной позиции 0. Если они различаются, то он определяет  
исход сравнения. Если они равны, то сравниваются следующие два эле-  
мента в индексной позиции 1, и т. д.**

**Так вот, поскольку мы взяли эти кортежи из словаря, в каждом кортеже  
все значения в нулевой индексной позиции, бывшие ранее ключами  
словаря, являются уникальными. Поэтому здесь не придется решать про-  
блемы с повторами.**

**В некоторых случаях лексикографическое упорядочивание может быть  
именно тем, что вам нужно. В других случаях, возможно, вместо этого  
стоит выполнить сортировку словаря по значению.**

**К счастью, есть способ взять полный контроль над тем, как упорядочи-  
ваются элементы. Вы можете управлять порядком их следования путем  
передачи функции ключа во встроенную функцию sorted(), которая из-  
менит то, как будут сравниваться элементы словаря.**

**Функция ключа — это просто обычная функция Python, которая будет вы-  
зываться с каждым элементом перед тем, как делать сравнения. Функция  
ключа на входе получает элемент словаря, а на выходе возвращает требу-  
емый «ключ» для сравнения порядка следования элементов.**

**К сожалению, слово «ключ» здесь используется в двух контекстах одно-  
временно: функция ключа не касается ключей словаря, она просто отме-  
чает каждый входной элемент произвольным ключом сравнения.**

**Теперь, возможно, нам стоит взглянуть на пример. Поверьте, понять функ-  
ции ключа будет намного легче, как только вы увидите их в реальном коде.**

**Допустим, вы хотите получить отсортированное представление словаря  
на основе его значений. Чтобы получить этот результат, вы можете исполь-  
зовать следующую ниже функцию ключа, которая возвращает значение  
каждой пары ключ-значение путем поиска второго элемента в кортеже:**

**>>> sorted(xs.items(), key=lambda x: x[1])**

**[('d', 1), ('c', 2), ('b', 3), ('a',** 4**)]**

**Видите, как теперь результирующий список пар ключ-значение отсортиро-  
ван по значениям, хранящимся в оригинальном словаре? Чтобы осмыслить  
принцип работы функции ключа, стоит потратить немного времени. Этот  
мощный принцип можно применять во всех видах контекстов Python.**

**На самом деле этот принцип настолько распространен, что стандартная  
библиотека Python включает модуль operator. Этот модуль реализует  
часть наиболее часто используемых функций ключа в качестве структур-  
ных блоков, автоматически конфигурируемых по принципу plug-and-play,  
таких как operator.itemgetter и operator.attrgetter.**

**Ниже приведен пример того, как можно заменить поиск по индексу на  
основе лямбды в первом примере на operator.itemgetter:**

**>>> import operator**

**>>> sorted(xs.items(), key=operator.itemgetter(1))**

**[('d', 1), ('c', 2), ('b', 3), ('a', 4)]**

**Использование модуля operator в некоторых случаях помогает яснее  
передавать замысел вашего программного кода. С другой стороны, про-  
стое лямбда-выражение может быть столь же удобочитаемым и более  
очевидным. В этом конкретном случае я предпочитаю лямбда-выражение.**

**Еще одна выгода от использования лямбд в качестве собственной функ-  
ции ключа состоит в том, что вам удается управлять порядком сортировки**

**гораздо детальнее. Например, вы можете отсортировать словарь на основе  
абсолютной числовой величины каждого хранящегося в нем значения:**

**>>> sorted(xs.items(), key=lambda x: abs(x[1]))**

**Если вам нужно инвертировать порядок сортировки так, чтобы более  
крупные значения шли вначале, то во время вызова sorted() вы можете  
применить именованный аргумент reverse=True:**

**>>> sorted(xs.items(),**

**key=lambda x: x[1],  
reverse=True)**

**[('a', 4), ('b', 3), ('c', 2), ('d', 1)]**

**Как я отмечал ранее, точно стоит потратить немного времени на то, чтобы  
твердо усвоить принцип работы функций ключа в Python. Они обеспечат  
вас гибкостью и смогут уберечь от написания исходного кода, единствен-  
ная цель которого — преобразовать одну структуру данных в другую.**

**Ключевые выводы**

* **Создавая сортированные «представления» словарей и другие коллек-  
  ции, вы можете влиять на порядок сортировки при помощи функции  
  ключа.**
* **Функции ключа являются в Python важным принципом. Наиболее  
  часто используемые из них были даже добавлены в модуль operator  
  стандартной библиотеки.**
* **В Python функции являются объектами первого класса. Вы обнаружи-  
  те, что это мощное средство языка применяется повсюду.**

**7. 3 . Имитация инструкций выбора  
на основе словарей**

**В Python нет инструкций выбора switch-case, поэтому иногда в качестве  
обходного пути возникает необходимость писать цепочки инструкций  
if^elif^else. В данном разделе вы узнаете прием, который сможете при-  
менять для имитации инструкций выбора switch-case в Python при по-  
мощи словарей и первоклассных функций. Звучит заманчиво? Отлично,  
тогда поехали!**

**Предположим, что в нашей программе есть такая цепочка инструкций if:**

**>>> if cond == 'cond\_a':  
handle\_a()**

**elif cond == 'cond\_b':  
handle\_b()  
else:**

**handle\_default()**

**В случае лишь трех разных условий это, конечно, не так страшно. Но  
представьте, если бы в этой инструкции у нас было десять или более от-  
ветвлений elif. Все стало бы выглядеть немного иначе. Я рассматриваю  
длинные цепочки инструкций if как код «с душком», который делает  
программы труднее для восприятия и в сопровождении.**

**Один из путей преодоления длинных инструкций if^elif^else состоит  
в их замене на таблицы поиска по словарю, которые имитируют поведение  
инструкций выбора switch-case.**

**Мы знаем, что в Python есть функции первого класса. А это означает, что  
их можно передавать в качестве аргументов в другие функции, возвра-  
щать в качестве значений из других функций, присваивать переменным  
и хранить в структурах данных.**

**Например, мы можем определить функцию, а затем сохранить ее в списке  
для доступа к ней в дальнейшем:**

**>>> def myfunc(a, b):**

**... return a + b  
>>> funcs = [myfunc]**

**>>> funcs[0]**

**<function myfunc at 0x107012230>**

**Синтаксис вызова этой функции работает именно так, как вы интуи-  
тивно ожидаете: мы просто обращаемся к списку по индексу и исполь-  
зуем вызывной синтаксис «()», чтобы вызвать функцию и передать ей  
аргументы:**>>> funcs[0](2, 3)

**Итак, каким же образом мы собираемся использовать функции первого  
класса, чтобы подрезать нашу цепочечную инструкцию if по размеру?  
Центральная идея здесь - определить словарь, отображающий ключи по-  
иска входных условий на функции, которые выполнят предназначенные  
операции:**

**>>> func\_dict = {**

**... 'cond\_a': handle\_a,**

**... 'cond\_b': handle\_b**

**... }**

**Вместо процеживания сквозь инструкции if, проверяя по ходу каждое  
условие, мы можем выполнить поиск ключа по словарю, чтобы получить  
функцию-обработчик, а затем вызвать ее:**

**>>> cond = 'cond\_a'**

**>>> func\_dict[cond]()**

**Эта реализация уже почти рабочая, по крайней мере, если условие cond  
можно найти в словаре. Если же его там нет, то мы получим исключение  
KeyError.**

**Давайте отыщем способ поддержки случая по умолчанию, который будет  
соответствовать исходному ответвлению else. К счастью, все словари  
Python располагают методом get(), который возвращает либо значение  
по заданному ключу, либо значение по умолчанию, если ключ не может  
быть найден. Это именно то, что нам здесь и нужно:**

**>>> func\_dict.get(cond, handle\_default)()**

**Поначалу этот фрагмент кода, возможно, будет выглядеть синтаксически  
странным, но когда вы разложите его по полочкам, то поймете, что он  
работает в точности как предыдущий пример. Опять-таки, мы использу-  
ем функции Python первого класса, чтобы передать в поисковый метод  
get() функцию handle\_default в качестве запасного значения. Благодаря  
этому, если условие в словаре не может быть найдено, мы избегаем вызова  
исключения KeyError и вместо него вызываем заданную по умолчанию  
функцию-обработчик.**

**Посмотрим на более законченный пример применения поиска по словарю  
и функций первого класса для замены цепочек инструкций if. После оз-  
накомления с приведенным ниже примером вы сможете увидеть шаблон,  
необходимый для сведения определенных видов инструкций if к диспет-  
черизации на основе словаря.**

**Мы собираемся написать еще одну функцию с цепочкой инструкций if,  
которую затем преобразуем. Данная функция принимает строковый код  
операции, к примеру «add» или «mul», и затем выполняет соответствую-  
щие математические расчеты на операндах x и у:**

**>>> def dispatch\_if(operator, x, у):  
if operator == 'add':**

|  |  |  |
| --- | --- | --- |
| **return x +** | **y** |  |
| **elif operator** | **==** | **'sub'** |
| **return x -** | **y** |  |
| **elif operator** | **==** | **'mul'** |
| **return x \*** | **y** |  |
| **elif operator** | **==** | **'div'** |

**return x / y**

**Сказать по правде, это очередной игрушечный пример (не хотелось бы  
вам здесь докучать нескончаемыми страницами исходного кода), но он  
будет служить наглядной иллюстрацией лежащего в основе шаблона  
проектирования. Как только вы «въедете» в образец, то сможете его при-  
менять в самых разных сценариях.**

**Вы можете испытать функцию dispatch\_if() на предмет выполнения  
простых вычислений, вызвав эту функцию со строковым кодом операции  
и двумя числовыми операндами:**>>> dispatch\_if('mul', 2, 8)

**16**

**>>> dispatch\_if('неизвестно', 2, 8)  
None**

**Обратите внимание на то, что ' неизвестный' случай срабатывает, пото-  
му что Python добавляет в конец любой функции неявную инструкцию  
return None.**

**Пока все неплохо. Теперь преобразуем первоначальную функцию dispatch\_  
if() в новую функцию, использующую словарь для отображения кодов  
операций в арифметические операции с функциями первого класса:**

**>>> def dispatch\_dict(operator, x, y):  
return {**

**'add'**

**'sub'**

**'mul'**

**'div'**

**lambda: x + y,  
lambda: x - y,  
lambda: x \* y,  
lambda: x / y,**

**}.get(operator, lambda: None)()**

**Такая реализация на основе словаря дает те же самые результаты, что  
и первоначальная функция dispatch\_if(). Мы можем вызвать обе функ-  
ции точно таким же образом:**

**>>> dispatch\_dict('mul', 2, 8)**

**16**

**>>> dispatch\_dict('неизвестно', 2, 8)**

**None**

**Есть пара способов, которыми этот код можно усовершенствовать еще  
больше, если бы он был реален и предназначался для эксплуатации.**

**Во-первых, всякий раз, когда мы вызываем dispatch\_dict(), он создает  
временный словарь и кучу лямбд для поиска кода операции. С точки зре-  
ния производительности это не идеально. В случае, если программный код  
нуждаетется в быстродействии, имеет больше смысла единожды создать  
словарь в качестве константы и затем ссылаться на него во время вызова  
функции. Не стоит воссоздавать словарь всякий раз, когда мы должны  
выполнить по нему поиск.**

**Во-вторых, если бы мы и правда захотели выполнить несколько простых  
арифметических операций типа x + y, то вместо используемых в этом при-  
мере лямбда-функций было бы гораздо лучше использовать встроенный  
модуль Python operator. Модуль operator предоставляет реализации  
всех операторов Python, в частности operator.mul, operator.div и т. д.  
Хотя эта деталь малозначительна. В этом примере лямбды использованы  
намеренно, чтобы сделать его более универсальным. Он должен помочь  
вам применять этот шаблон и в других ситуациях.**

**Итак, теперь у вас есть еще один инструмент в наборе хитрых приемов,  
который вы можете использовать для упрощения некоторых цепочек ин-  
струкций if на случай, если они будут становиться громоздкими. Просто  
запомните: этот прием применим не во всех ситуациях, и иногда будет  
лучше обойтись простой инструкцией if.**

**Ключевые выводы**

* **В Python нет инструкции выбора switch-case. Но в некоторых случа-  
  ях вы можете избежать длинных цепочек инструкций if при помощи  
  таблицы диспетчеризации на основе словаря.**
* **Функции первого класса Python в очередной раз доказывают, что они  
  являются мощным инструментом. Но чем больше сила, тем больше  
  ответственность.**

**7. 4 . Самое сумасшедшее выражение-словарь  
на западе**

**Иногда вы наталкиваетесь на крошечный пример кода, который обладает  
поистине неожиданной глубиной — одна-единственная строка кода, кото-  
рая способна многому научить, если хорошенько над ней поразмыслить.  
Такой фрагмент код — это как коан в дзен-буддизме: вопрос или утвержде-  
ние, используемое в практике дзен, чтобы вызвать сомнение и проверить  
достижения ученика.**

**Крошечный фрагмент кода, который мы обсудим в этом разделе, являет-  
ся одним из таких примеров. На первый взгляд он может выглядеть как  
прямолинейное выражение-словарь, но при ближайшем рассмотрении  
он отправляет вас в расширяющий сознание психоделический круиз по  
интерпретатору СPython.**

**От этого однострочника я получаю такой кайф, что как-то раз я даже на-  
печатал его на своем значке участника конференции по Python в качестве  
повода для беседы. Это привело к нескольким конструктивным диалогам  
с участниками моей электронной рассылки по Python.**

**Итак, без дальнейших церемоний, вот этот фрагмент кода. Возьмите па-  
узу, чтобы поразмышлять над приведенным ниже выражением-словарем  
и тем, к чему его вычисление должно привести:**

**>>> {True: 'да', 1: 'нет', 1.0: 'возможно'}**

**Я подожду здесь\_**

**О’кей, готовы?**

**Ниже показан результат, который мы получим при вычислении приведен-  
ного выше выражения-словаря в сеансе интерпретатора Python:**

**>>> {True: 'да', 1: 'нет', 1.0: 'возможно'}**

**{True: 'возможно'}**

**Признаюсь, когда увидел этот результат впервые, я был весьма ошарашен.  
Но все встанет на свои места, когда вы проведете неспешное пошаговое  
изучение того, что тут происходит. Давайте поразмыслим, почему мы  
получаем этот, надо сказать, весьма не интуитивный результат.**

**Когда Python обрабатывает наше выражение-словарь, он сначала строит  
новый пустой объект-словарь, а затем присваивает ему ключи и значения  
в том порядке, в каком они переданы в выражение-словарь.**

**Тогда, когда мы его разложим на части, наше выражение-словарь будет  
эквивалентно приведенной ниже последовательности инструкций, кото-  
рые исполняются по порядку:**

**>>> xs = dict()**

**>>> xs[True] = 'да'**

**>>> xs[1] = 'нет'**

**>>> xs[1.0] = 'возможно'**

**Как ни странно, Python считает все ключи, используемые в этом примере  
словаря, эквивалентными:**

**>>> True == 1 == 1.0  
True**

**Ладно, но погодите минуточку. Уверен, вы сможете интуитивно признать,  
что 1.0 == 1, но вот почему True считается также эквивалентным и 1?  
В первый раз, когда я увидел это выражение-словарь, оно действительно  
меня озадачило.**

**Немного покопавшись в документации Python, я узнал, что Python рас-  
сматривает тип bool как подкласс типа int. Именно так обстоит дело  
в Python 2 и Python 3:**

**Булев тип — это подтип целочисленного типа, и булевы значения ведут  
себя, соответственно, как значения 0 и 1 почти во всех контекстах, при  
этом исключением является то, что при преобразовании в строковый тип,  
соответственно, возвращаются строковые значения 'False' или 'True'[[69]](#footnote-69) .**

**И разумеется, это означает, что в Python булевы значения технически  
можно использовать в качестве индексов списка или кортежа:**

**>>> ['нет', 'да'][Тгие]**

***'да'***

**Но вам, пожалуй, не следует использовать подобного рода логические  
переменные во имя ясности (и душевного здоровья ваших коллег).**

**Так или иначе, вернемся к нашему выражению-словарю.**

**Что касается языка Python, то все эти значения — True, 1 и 1.0 — пред-  
ставляют одинаковый ключ словаря. Когда интерпретатор вычисляет вы-  
ражение-словарь, он неоднократно переписывает значение ключа True.  
Это объясняет, почему в самом конце результирующий словарь содержит  
всего один ключ.**

**Прежде чем мы пойдем дальше, взглянем еще раз на исходное выраже-  
ние-словарь:**

**>>> {True: 'да', 1: 'нет', 1.0: 'возможно'}**

**{True: 'возможно'}**

**Почему здесь в качестве ключа мы по-прежнему получаем True? Разве  
не должен ключ из-за повторных присваиваний в самом конце тоже по-  
меняться на 1.0?**

**После небольших изысканий в исходном коде интерпретатора Python  
я выяснил, что, когда с объектом-ключом ассоциируется новое значение,  
словари Python сам этот объект-ключ не обновляют:**

**>>> ys = {1.0: 'нет'}**

**>>> ys[True] = 'да'**

**>>> ys  
{1.0: 'да'}**

**Безусловно, это имеет смысл в качестве оптимизации производительно-  
сти: если ключи рассматриваются идентичными, то зачем тратить время  
на обновление оригинала?**

**В последнем примере вы видели, что первоначальный объект True как  
ключ никогда не заменяется. По этой причине строковое представление  
словаря по-прежнему печатает ключ как True (вместо 1 или 1.0).**

**С тем, что мы знаем теперь, по всей видимости, значения в результирую-  
щем словаре переписываются только потому, что сравнение всегда будет  
показывать их как эквивалентные друг другу. Вместе с тем оказывается,  
что этот эффект не является следствием проверки на эквивалентность  
методом eq тоже.**

**Словари Python опираются на структуру данных хеш-таблица. Когда  
я впервые увидел это удивительное выражение-словарь, моя первая  
мысль заключалась в том, что такое поведение было как-то связано с хеш-  
конфликтами.**

**Дело в том, что хеш-таблица во внутреннем представлении хранит имеющи-  
еся в ней ключи в различных «корзинах» в соответствии с хеш-значением  
каждого ключа. Хеш-значение выводится из ключа как числовое значение  
фиксированной длины, которое однозначно идентифицирует ключ.**

**Этот факт позволяет выполнять быстрые операции поиска. Намного бы-  
стрее отыскать числовое хеш-значение ключа в поисковой таблице, чем**

**сравнивать полный объект-ключ со всеми другими ключами и выполнять  
проверку на эквивалентность.**

**Вместе с тем способы вычисления хеш-значений, как правило, не иде-  
альны. И в конечном счете два или более ключа, которые на самом деле  
различаются, будут иметь одинаковое производное хеш-значение, и они  
в итоге окажутся в той же самой корзине поисковой таблицы.**

**Когда два ключа имеют одинаковое хеш-значение, такая ситуация на-  
зывается хеш-конфликтом и является особым случаем, с которым долж-  
ны разбираться алгоритмы вставки и нахождения элементов в хеш-  
таблице.**

**Исходя из этой оценки, весьма вероятно, что хеширование как-то связано  
с неожиданным результатом, который мы получили из нашего выраже-  
ния-словаря. Поэтому давайте выясним, играют ли хеш-значения ключей  
здесь тоже какую-то определенную роль.**

**Я определяю приведенный ниже класс как небольшой сыскной инстру-  
мент:**

**class AlwaysEquals:**

**def eq (self, other):**

**return True**

**def \_\_hash\_\_(self):  
return id(self)**

**Этот класс характерен двумя аспектами.**

**Во-первых, поскольку дандер-метод eq всегда возвращает True, все**

**экземпляры этого класса притворяются, что они эквивалентны любому  
объекту:**

**>>> AlwaysEquals() == AlwaysEquals()**

**True**

**>>> AlwaysEquals() == 42  
True**

**>>> AlwaysEquals() == 'штаа?'**

**True**

**И во-вторых, каждый экземпляр AlwaysEquals также будет возвращать  
уникальное хеш-значение, генерируемое встроенной функцией id():**

**>>> objects = [AlwaysEquals(),**

**AlwaysEquals(),**

**AlwaysEquals()]**

**>>> [hash(obj) for obj in objects]**

**[4574298968, 4574287912, 4574287072]**

**В Python функция id() возвращает адрес объекта в оперативной памяти,  
который гарантированно является уникальным.**

**При помощи этого класса теперь можно создавать объекты, которые при-  
творяются, что они являются эквивалентными любому другому объекту,  
но при этом с ними будет связано уникальное хеш-значение. Это позво-  
лит проверить, переписываются ли ключи словаря, опираясь только на  
результат их сравнения на эквивалентность.**

**И, как вы видите, ключи в следующем ниже примере не переписываются,  
несмотря на то что сравнение всегда будет показывать их как эквивалент-  
ные друг другу:**

**>>> {AlwaysEquals(): 'да', AlwaysEquals(): 'нет'}**

**{ <AlwaysEquals object at 0x110a3c588>: 'да',**

**<AlwaysEquals object at 0x110a3cf98>: 'нет' }**

**Мы также можем взглянуть на эту идею с другой стороны и проверить,  
будет ли возврат одинакового хеш-значения достаточным основанием для  
того, чтобы заставить ключи быть переписанными:**

**class SameHash:**

**def hash (self):**

**return 1**

**Сравнение экземпляров класса SameHash будет показывать их как не  
эквивалентные друг другу, но они все будут обладать одинаковым хеш-  
значением, равным 1:**

**>>> a = SameHash()**

**>>> b = SameHash()**

**>>> a == b  
False**

**>>> hash(a), hash(b)**

**(1, 1)**

**Давайте посмотрим, как словари Python реагируют, когда мы пытаемся  
использовать экземляры класса SameHash в качестве ключей словаря:**

**>>> {a: 'a', b: 'b'}**

**{ <SameHash instance at 0x7f7159020cb0>: 'a',**

**<SameHash instance at 0x7f7159020cf8>: 'b' }**

**Как показывает этот пример, эффект «ключи переписываются» вызыва-  
ется не одними только конфликтами хеш-значений.**

**Словари выполняют проверку на эквивалентность и сравнивают хеш-  
значение, чтобы определить, являются ли два ключа одинаковыми.  
Попробуем резюмировать результаты нашего исследования.**

**Выражение-словарь {True: 'да', 1: 'нет', 1.0: 'возможно'} вычисляется  
как {True: ' возможно'}, потому что сравнение всех ключей этого примера,  
True, 1, и 1.0, будет показывать их как эквивалентные друг другу, и они  
все имеют одинаковое хеш-значение:**

**>>> True == 1 == 1.0  
True**

**>>> (hash(True), hash(1), hash(1.0))**

**(1, 1, 1)**

**Пожалуй, теперь уже не так удивительно, что мы получили именно такой  
результат в качестве конечного состояния словаря:**

**>>> {True: 'да', 1: 'нет', 1.0: 'возможно'}**

**{True: 'возможно'}**

**Здесь мы затронули много тем, и этот конкретный трюк Python поначалу  
может не укладываться в голове — вот почему в самом начале раздела  
я сравнил его с коаном в дзен.**

**Если вы с трудом понимаете, что происходит в этом разделе, попробуйте  
поэкспериментировать по очереди со всеми примерами кода в сеансе  
интерпретатора Python. Вы будете вознаграждены расширением своих  
познаний о внутренних механизмах языка Python.**

**Ключевые выводы**

* **Словари рассматривают ключи как идентичные, если результат их**

**сравнения методом eq говорит о том, что они эквивалентны, и если**

**их хеш-значения одинаковы.**

* **Неожиданные конфликты ключей словаря могут и будут приводить  
  к неожиданным результатам.**

**7. 5 . Так много способов объединить словари**

**Вы когда-нибудь конструировали систему конфигурации для одной из  
ваших программ Python? В таких системах принято принимать структуру  
данных с параметрами конфигурации, заданными по умолчанию, а затем  
предоставлять возможность селективно переопределять эти параметры  
на основе вводимых пользователем данных или некоторого другого ис-  
точника конфигурации.**

**Я нередко использовал словари в качестве базовой структуры данных  
для представления ключей и значений конфигурации. И поэтому мне  
часто был нужен способ объединения, или слияния (merge), принятых по  
умолчанию параметров конфигурации с пользовательскими переопре-  
делениями в один-единственный словарь с окончательными значениями  
конфигурации.**

**Или, обобщая: иногда вам нужен способ объединить два или более слова-  
ря в один, чтобы результирующий словарь содержал комбинацию ключей  
и значений исходных словарей.**

**В этом разделе я покажу несколько способов сделать это. Сначала по-  
смотрим на простой пример, чтобы можно было что-то обсуждать. Пред-  
положим, что у вас имеется два исходных словаря:**

**>>> xs = {'a': 1, 'b': 2}**

**>>> ys = {'b': 3, 'c': 4}**

**И вы хотите создать новый словарь zs, который содержит все ключи и зна-  
чения xs и все ключи и значения ys. Кроме того, если вы внимательно  
прочли этот пример, то вы заметили, что строка ' b' появляется в качестве  
ключа в обоих словарях, — нам также придется продумать стратегию раз-  
решения конфликтов для повторяющихся ключей.**

**В Python классическое решение задачи «слияния многочисленных сло-  
варей» состоит в том, чтобы использовать встроенный в словарь метод  
update():**

**>>> zs = {}**

**>>> zs.update(xs)**

**>>> zs.update(ys)**

**Если вам любопытно, то наивная реализация функции update() могла бы  
выглядеть примерно следующим образом. Мы просто перебираем в цикле  
все элементы словаря с правой стороны и добавляем каждую пару ключ-  
значение в словарь с левой стороны, по ходу переписывая существующие  
ключи:**

**def update(dict1, dict2):**

**for key, value in dict2.items():  
dict1[key] = value**

**В результате мы получим новый словарь zs, который теперь содержит  
ключи, определенные в xs и ys:**

**>>> zs**

**>>> {'c': 4, 'a': 1, 'b': 3}**

**Вы также увидите, что порядок, в котором мы вызываем update(), опре-  
деляет то, как будут разрешаться конфликты. Выигрывает последнее  
обновление, и повторяющийся ключ 'b' ассоциируется со значением 3,  
которое поступило из ys, то есть второго исходного словаря.**

**Разумеется, вы можете расширить эту цепочку вызовов update() настоль-  
ко, насколько захотите, для того, чтобы объединить любое количество  
словарей в один словарь. Такое практическое и удобочитаемое решение  
работает в Python 2 и в Python 3.**

**Еще один прием, который работает в Python 2 и в Python 3, использует  
встроенную функцию dict() совместно с оператором \*\* для «распаковки»  
объектов:**

**>>> zs = dict(xs, \*\*ys)**

**>>> zs**

**{'a': 1, 'c': 4, 'b': 3}**

**Однако, как и в случае с повторными вызовами update(), этот подход  
работает только для слияния исключительно двух словарей и не может  
быть обобщен для объединения произвольного количества словарей за  
один шаг.**

**Начинания с Python 3.5, оператор \*\* стал гибче[[70]](#footnote-70). Поэтому в Python 3.5+  
есть еще один — и, пожалуй, более приятный — способ объединения про-  
извольного количества словарей:**

**>>> zs = {\*\*xs, \*\*ys}**

**У этого выражения в точности такой же результат, что и у цепочки вы-  
зовов update(). Ключи и значения задаются в порядке слева направо,  
поэтому мы получаем ту же самую стратегию разрешения конфликтов:  
правая сторона имеет приоритет, а значение в ys переопределяет любое  
существующее значение под тем же самым ключом в xs. Это станет по-  
нятным, когда мы посмотрим на словарь, который является результатом  
этой операции слияния:**

**>>> zs**

**>>> {'c': 4, 'a': 1, 'b': 3}**

**Лично мне нравится краткость этой новой синтаксической конструкции  
и то, как она по-прежнему остается достаточно удобочитаемой. Всегда  
приходится находить равновесие между многословностью и краткостью,  
сохраняя программный код максимально удобочитаемым и легким в со-  
провождении.**

**В данном случае я склоняюсь к использованию нового синтаксиса при  
условии, что работаю с Python 3. Более того, при использовании опера-  
тора \*\* операция слияния выполняется быстрее, чем при использовании  
цепочки вызовов update(), что является еще одним преимуществом.**

**Ключевые выводы**

* **В Python 3.5 и выше для слияния многочисленных объектов-словарей  
  в один можно использовать оператор \*\* с использованием одного-  
  единственного выражения, переписывая существующие ключи слева  
  направо.**
* **Чтобы оставить программный код совместимым с более ранними  
  версиями Python, можно использовать встроенный в словарь метод  
  update().**

**7. 6 . Структурная печать словаря**

**Вы когда-либо пытались выявить баг в одной из своих программ, усеивая  
ее кучей отладочных инструкций print, чтобы проследить поток испол-  
нения? Или, возможно, вам приходилось генерировать диагностическое  
сообщение, чтобы выводить некоторые параметры конфигурации^**

**Я был разочарован, и часто, тем, насколько трудно в Python читать не-  
которые структуры данных, когда они печатаются как текстовые стро-  
ки. Например, ниже приведен простой словарь. Он напечатан в сеансе  
интерпретатора, при этом порядок следования ключей произвольный  
и в результирующей строке отсутствует выделение отступами:**

**>>> mapping = {'a': 23, 'b': 42, 'c': 0xc0ffee}**

**>>> str(mapping)**

**{'b': 42, 'c': 12648430, 'a': 23}**

**К счастью, есть несколько простых в использовании альтернатив нераз-  
борчивому преобразованию в стиле to-string, дающих более удобочитаемый  
результат. Один из вариантов состоит в использовании встроенного модуля  
Python json. Чтобы выполнить структурную печать словаря с более при-  
ятным форматированием, можно применить функцию json.dumps():**>>> import json

**>>> json.dumps(mapping, indent=4, sort\_keys=True)  
{**

**"a"**

**"b"**

**"c"**

**23,**

**42,**

**12648430  
}**

**Эти настройки конфигурации в результате получают хорошее и выделен-  
ное отступами строковое представление, которое к тому же нормализует  
порядок следования ключей словаря для оптимальной удобочитаемости.**

**Несмотря на то что это решение дает внешне красивый и удобочитаемый  
результат, оно не является идеальным. Печать словарей при помощи мо-  
дуля json работает только со словарями, которые содержат примитивные  
типы, — вы столкнетесь с проблемой при попытке распечатать словарь,  
который содержит непримитивный тип данных, таких как функция:**

**>>> json.dumps({all: 'yup'})**

**TypeError: "keys must be a string"**

**Еще один недостаток использования функции json.dumps() состоит  
в том, что она не способна сериализовать составные типы данных, такие  
как множества:**>>> mapping['d'] = {1, 2, 3}

**>>> json.dumps(mapping)**

**TypeError: "set([1, 2, 3]) is not JSON serializable"**

**Кроме того, вы можете столкнуться с такой проблемой, как представление  
текста в кодировке Юникод, — в некоторых случаях вы не сможете взять  
результат на выходе из j son. dumps и скопипастить его в сеансе интерпре-  
татора Python, чтобы реконструировать первоначальный объект-словарь.**

**Классическим решением задачи структурной печати объектов Python  
является встроенный модуль pprint. Приведем пример:**

**>>> import pprint**

**>>> pprint.pprint(mapping)**

**{'a': 23, 'b': 42, 'c': 12648430, 'd': set([1, 2, 3])}**

**Вы видите, что функция pprint способна печатать такие типы данных,  
как множества, и она также печатает ключи словаря в воспроизводимом  
порядке. По сравнению со стандартным строковым представлением сло-  
варей, здесь мы получаем то, что воспринимается значительно легче.**

**Вместе с тем, по сравнению с json.dumps(), она не представляет вло-  
женные структуры визуально столь же хорошо. В зависимости от об-  
стоятельств это может быть преимуществом или недостатком. Я иногда  
использую json.dumps(), чтобы выводить словари из-за улучшенной  
удобочитаемости и форматирования, но только если я уверен, что в них  
нет непримитивных типов данных.**

**Ключевые выводы**

* **В Python принятое по умолчанию преобразование объектов-словарей  
  в строковое представление может оказаться трудночитаемым.**
* **Модули pprint и json представляют собой варианты «более высокого  
  качества», встроенные в стандартную библиотеку Python.**
* **Будьте осторожны с использованием функции json .dumps() и непри-  
  митивных ключей и значений, поскольку это вызовет исключение  
  TypeError.**

**Питоновские методы**

**повышения**

**производительности**

1. **.1 . Исследование модулей и объектов Python**

**Вы можете в интерактивном режиме исследовать модули и объекты не-  
посредственно из интерпретатора Python. Это недооцененное функцио-  
нальное средство легко упустить из виду, особенно если вы переходите на  
Python с другого языка.**

**Многие языки программирования затрудняют инспектирование пакета  
или класса без сверки с онлайн-документацией или заучивания опреде-  
лений интерфейсов наизусть.**

**В Python дела обстоят по-другому — эффективный разработчик будет  
проводить массу времени в сеансах интерпретатора REPL, работая ин-  
терактивно с интерпретатором Python. Например, я часто это делаю для  
разработки коротких фрагментов кода и логики, после чего копирую их  
и вставляю в файл Python, с которым я работаю в своем редакторе.**

**В этой главе вы познакомитесь с двумя простыми приемами, которые  
можно использовать для исследования классов и методов Python инте-  
рактивно, находясь внутри интерпретатора.**

**Эти приемы будут работать с любой версией Python — надо лишь за-  
пустить интерпретатор Python командой python из командной строки  
и приступить к работе. Интерпретатор прекрасно подойдет для сеансов  
отладки в системах, где, например, у вас нет доступа к причудливому**

**редактору или IDE, потому что вы работаете по Сети в терминальном  
сеансе.**

**Готовы? Поехали! Представьте, что вы пишете программу, которая ис-  
пользует модуль Python datetime стандартной библиотеки. Как узнать,  
какие функции или классы этот модуль экспортирует и какие методы  
и атрибуты находятся в его классах?**

**Один из способов заключается в том, чтобы обратиться за советом к по-  
исковой системе или заглянуть в официальную документацию Python  
в Сети. Однако встроенная в Python функция dir() позволяет вам полу-  
чать доступ к этой информации непосредственно из Python REPL:**

**>>> import datetime  
>>> dir(datetime)**

**['MAXYEAR', 'MINYEAR', ' builtins ', ' cached ',**

**' doc ' file ' loader ' name**

**' package ', ' spec ', '\_divide\_and\_round',**

**'date', 'datetime', 'datetime\_CAPI', 'time',**

**'timedelta', 'timezone', 'tzinfo']**

**В приведенном выше примере я сначала импортировал модуль datetime  
из стандартной библиотеки, а затем проинспектировал его функцией  
dir(). Вызов dir() с модулем в качестве аргумента выдаст расположен-  
ный в алфавитном порядке список имен и атрибутов, которые этот модуль  
предоставляет.**

**Поскольку в Python абсолютно «все» является объектом, тот же самый  
прием будет работать не только с модулями как таковыми, но и с классами  
и структурами данных, экспортируемыми этим модулем.**

**В действительности можно продолжить углубляться в подробности модуля,  
снова вызывая dir() с отдельными объектами, которые вызывают интерес.  
Например, ниже показано, как инспектируется класс datetime.date:**

**>>> dir(datetime.date)**

**[' add ', ' class ', 'day', 'fromordinal',**

**'isocalendar', 'isoformat', 'isoweekday', 'max',**

**'min', 'month', 'replace', 'resolution', 'strftime',**

**'timetuple', 'today', 'toordinal', 'weekday', 'year']**

**Как видите, функция dir() дает вам краткий обзор того, что доступно  
в модуле или классе. Если вы не помните точного написания конкретного  
класса или функции, то, возможно, это все, что вам нужно, чтобы продол-  
жать работу, не прерывая процесс программирования.**

**Иногда вызов функции dir() с объектом в качестве аргумента выдаст  
слишком много информации — составной модуль или класс вызовет  
длинную распечатку, которую трудно быстро прочитать. Ниже приведен  
небольшой трюк, который можно применять для сведения списка атри-  
бутов к тем, которыми вы интересуетесь:**

**>>> [\_ for \_ in dir(datetime) if 'date' in \_.lowerQ]**

**['date', 'datetime', 'datetime\_CAPI']**

**Здесь я использовал конструкцию включения в список для фильтрации  
результатов вызова dir(datetime), чтобы получить только имена, которые  
включают слово «date». Обратите внимание на то, как я вызывал метод  
lower() с каждым именем, тем самым гарантируя, что фильтрация будет  
нечувствительна к регистру.**

**Получение сырого списка атрибутов объекта не всегда обеспечивает нас  
достаточной информацией, необходимой для решения текущей задачи.  
Тогда каким образом можно получить больше информации и более под-  
робные описания функций и классов, экспортируемых модулем datetime?**

**В этом случае вам поможет встроенная в Python функция help(). С ее по-  
мощью вы можете вызывать интерактивную справочную систему Python  
и просматривать автоматически сгенерированную документацию Python  
по любому объекту:**

**>>> help(datetime)**

**Если выполнить приведенный выше пример в сеансе интерпретатора  
Python, то ваш терминал покажет текстоориентированный экран справки  
для модуля datetime, который будет выглядеть примерно так:**

**Help on module datetime:**

**NAME**

**datetime - Fast implementation of the datetime type.**

**CLASSES**

**builtins.object**

**date**

**datetime**

**time**

**Вы можете использовать клавиши «курсор вверх» и «курсор вниз», чтобы  
прокрутить документацию на экране. Как вариант, также можно нажимать  
клавишу «пробел», чтобы прокручивать вниз сразу несколько строк. Что-  
бы выйти из режима интерактивной справки, нужно нажать клавишу q.  
Это вернет вас назад к командной строке интерпретатора. Неплохая воз-  
можность, да?**

**Между прочим, вы можете вызывать help() с произвольными объектами  
Python, включая другие встроенные функции и ваши собственные классы  
Python. Интерпретатор Python автоматически сгенерирует эту докумен-  
тацию на основе атрибутов, определенных в объекте, и его строки доку-  
ментации docstring (при ее наличии). Все приведенные ниже примеры  
являются допустимыми применениями функции help:**

**>>> help(datetime.date)**

**>>> help(datetime.date.fromtimestamp)**

**>>> help(dir)**

**Разумеется, функции dir() и help() не заменят собой красиво отформа-  
тированную HTML-документацию, мощь поисковой системы или поиск  
на сайте Stack Overflow. Но они являются великолепными инструментами  
для оперативной сверки, не требующим от вас переключения с интерпре-  
татора Python. Они также доступны вне Сети и работают без подключе-  
ния к интернету, что может оказаться очень полезным в случае крайней  
необходимости.**

**Ключевые выводы**

* **Используйте встроенную функцию dir(), чтобы интерактивно исследо-  
  вать модули и классы Python, находясь внутри сеанса интерпретатора.**
* **Встроенная функция help() позволяет просматривать документацию  
  прямо из вашего интерпретатора (для выхода нажмите клавишу q).**

**8.2 . Изоляция зависимостей проекта  
при помощи Virtualenv**

**Python содержит мощную систему управления пакетами, позволяющую  
управлять модулями, от которых зависят ваши программы. Вы, вероятно,  
ее использовали, когда устанавливали сторонние пакеты при помощи  
команды менеджера пакетов pip.**

**Сбивающим с толку аспектом установки пакетов при помощи pip являет-  
ся то, что он по умолчанию пытается устанавливать их в вашу глобальную  
среду Python.**

**Несомненно, это делает любые устанавливаемые вами новые пакеты  
доступными в вашей системе глобально, что очень удобно. Но это также  
быстро превращается в кошмар, если вы работаете с многочисленными  
проектами, которые требуют разных версий того же самого пакета.**

**Что, если один из ваших проектов нуждается в версии 1.3 библиотеки,  
в то время как для другого проекта нужна версия 1.4 той же самой би-  
блиотеки?**

**Когда вы устанавливаете пакеты глобально, может существовать только  
одна версия библиотеки Python для всех ваших программ. Это означает,  
что вы быстро столкнетесь с конфликтами версий — как Горец, который  
должен остаться только один.**

**И чем дальше, тем хуже. У вас также могут быть разные программы, для  
которых нужны различные версии самого языка Python. Например, неко-  
торые программы могут по-прежнему выполняться в Python 2, в то время  
как основная часть вашей новой разработки происходит в Python 3. Или  
что, если для одного из ваших проектов нужен Python 3.3, в то время как  
все остальное работает в Python 3.6?**

**Помимо этого, глобальная установка пакетов Python также может стать  
фактором риска с точки зрения обеспечения безопасности. Для модифи-  
кации глобальной среды нередко требуется, чтобы вы выполняли коман-  
ды pip install с правами суперпользователя (root-, или админ-правами).  
Когда вы устанавливаете новый пакет, менеджер пакетов pip скачивает  
и исполняет код из интернета, а это обычно не рекомендуется. Хотелось  
бы надеяться, что устанавливаемый программный код заслуживает до-  
верия, но кто его знает, что он делает на самом деле\_**

**Виртуальные среды спешат на помощь**

**Решение этих проблем заключается в том, чтобы отделить вашу среду  
Python так называемыми виртуальными средами (virtual environment).  
Они позволяют вам отделять зависимости Python на основе того или ино-  
го проекта и предоставляют возможность выбирать между различными  
версиями интерпретатора Python.**

**Виртуальная среда — это изолированная среда Python. Физически она  
располагается внутри папки, содержащей все пакеты и другие программ-  
ные средства, от которых они зависят, в виде библиотек с нативным (плат-  
форменно-ориентированным) кодом и средой выполнения интерпрета-  
тора, в которых нуждается проект Python. (За кадром, чтобы сэкономить  
место, эти файлы могут быть символическими ссылками, а не реальными  
копиями.)**

**Чтобы продемонстрировать работу виртуальной среды, я представлю не-  
большую пошаговую демонстрацию, в которой мы выполним настройку  
новой виртуальной среды (или virtualenv, как ее называют для краткости),  
а затем установим в нее сторонний пакет.**

**Прежде всего проверим, где в настоящее время располагается глобальная  
среда Python. В Linux или macOS для проверки пути к менеджеру пакетов  
pip мы можем использовать инструмент командной строки which:**

**$ which pip3  
/usr/local/bin/pip3**

**Я обычно размещаю свои виртуальные среды прямо в папки проектов, что-  
бы держать их в безупречном виде и отделенными от остальных. Но вы так-  
же можете где-нибудь иметь специальный каталог «python-environments»,  
который содержит все ваши среды для проектов. Выбор за вами.**

**Давайте создадим новую виртуальную среду Python:**

**$ python3 -m venv ./venv**

**$ Is venv/  
bin**

**Include**

**Lib**

**pyvenv.cfg**

**Эта команда за одну минуту создаст новую папку venv в текущем каталоге,  
а также заполнит ее базовой средой Python 3:**

**Если вы проверите активную версию pip (командой which), то увидите,  
что она по-прежнему указывает на глобальную среду, в моем случае /usr/  
local/bin/pip3:**

**(venv) $ which pip3  
/usr/local/bin/pip3**

**Это означает, что если установить пакеты сейчас, то они по-прежнему  
окажутся в глобальной среде Python. Одного создания папки виртуальной  
среды недостаточно — вам нужно явным образом активировать новую  
виртуальную среду, чтобы последующие выполнения команды pip ука-  
зывали на нее:**

**$ source ./venv/bin/activate  
(venv) $**

**Выполнение команды activate конфигурирует текущий сеанс вашей  
оболочки, чтобы вместо этого использовать Python и команды pip из  
виртуальной среды[[71]](#footnote-71).**

**Обратите внимание на то, как это изменило вид подсказки в строке  
командной оболочки, и теперь она содержит название активной вирту-  
альной среды в круглых скобках: (venv). Давайте проверим, какой ис-  
полняемый файл pip теперь активен:**

**(venv) $ which pip3  
/Users/dan/my-project/venv/bin/pip3**

**Как видите, выполнение команды pip3 теперь будет запускать ту версию,  
которая находится в виртуальной среде, а не глобальной. То же касается  
и исполняемого файла интерпретатора Python. Выполнение python из  
командной строки теперь также загрузит интерпретатор из папки venv:**

**(venv) $ which python  
/Users/dan/my-project/venv/bin/python**

**Обратите внимание: она по-прежнему представляет собой «чистую доску»,  
абсолютно пустую среду Python. Выполнение команды pip list покажет  
почти пустой список установленных пакетов, который будет включать  
только базовые модули, необходимые для поддержки pip как такового:**

**(venv) $ pip list  
pip (9.0.1)  
setuptools (28.8.0)**

**Давайте пойдем дальше и теперь установим пакет Python в виртуальную  
среду. Для этого вам следует применить знакомую команду pip install:**

**(venv) $ pip install schedule  
Collecting schedule**

**Downloading schedule-0.4.2-py2.py3-none-any.whl  
Installing collected packages: schedule  
Successfully installed schedule-0.4.2**

**Здесь вы заметите два важных изменения. Во-первых, для выполнения  
этой команды вам больше не будут нужны права доступа администра-  
тора. И во-вторых, инсталляция или обновление пакета с активной вир-  
туальной средой означают, что все файлы окажутся в подпапке каталога  
виртуальной среды.**

**По этой причине программные средства, от которых зависит ваш проект,  
будут физически отделены от всей другой среды Python в вашей системе,  
включая глобальную. Практически вы получаете клон среды выполнения  
Python, который предназначен только для одного проекта.**

**Еще раз выполнив pip list, вы увидите, что библиотека schedule была  
успешно установлена в новую среду:**

**(venv) $ pip list  
pip (9.0.1)**

**schedule (0.4.2)  
setuptools (28.8.0)**

**Если запустить сеанс интерпретатора Python командой python или вы-  
полнить им автономный файл .py, то он будет использовать интерпретатор  
Python и зависимости, установленные в эту виртуальную среду, — при  
условии, что эта среда по-прежнему активна в текущем сеансе оболочки.**

**Но как снова деактивировать или «покинуть» виртуальную среду? Ана-  
логично команде activate, существует команда deactivate, которая воз-  
вращает вас назад к глобальной среде:**

**(venv) $ deactivate  
$ which pip3  
/usr/local/bin**

**Использование виртуальных сред поможет сохранить вашу систему  
лаконичной, а ваши зависимости Python аккуратно организованными.  
В качестве практической рекомендации: все ваши проекты Python долж-  
ны использовать виртуальные среды, чтобы отделять их зависимости от  
других и избегать конфликтов версий.**

**Понимание и использование виртуальных сред также направит вас по  
правильному пути использования более продвинутых методов управле-  
ния зависимостями, таких как определение зависимостей проекта при  
помощи файлов requirements.txt.**

**Если вы ищете материал с глубоким изложением этой темы и с дополни-  
тельными советами по поводу производительности, обратитесь к моему Кур-  
су управления зависимостями Python^, который можно найти на dbader. org.**

**Ключевые выводы**

**□ Виртуальные среды отделяют зависимости одних ваших проектов от  
других. Они помогают избегать конфликтов версий между пакетами  
и различными версиями среды выполнения Python. [[72]](#footnote-72)**

**□ В качестве практической рекомендации: все ваши проекты Python  
должны использовать виртуальные среды, в которых будут храниться  
их зависимости. Это поможет избежать головной боли в будущем.**

**8.3 . По ту сторону байткода**

**Когда интерпретатор СPython исполняет вашу программу, он сначала ее  
транслирует в последовательность байткодовых инструкций. Байткод —  
это промежуточный язык для виртуальной машины Python, который  
используется в качестве оптимизации производительности.**

**Вместо того чтобы непосредственно исполнить человекочитаемый исход-  
ный код, в Python используются компактные цифровые коды, константы  
и ссылки, которые представляют результат лексического и семантическо-  
го анализа, выполняемого компилятором.**

**Это экономит время и оперативную память для повторных исполнений  
программ или частей программ. Например, байткод, который получается  
в результате этого шага компиляции, кэшируется на диске в файлах .pyc  
и .pyo, чтобы во второй раз исполнение того же самого файла Python  
проходило быстрее.**

**Все это абсолютно прозрачно для программиста. Вам не нужно знать о том,  
что происходит в этот промежуточный шаг трансляции или как виртуаль-  
ная машина Python работает с байткодом. На самом деле формат байткода  
считается деталью реализации, и не гарантируется, что он останется ста-  
бильным или совместимым между различными версиями Python.**

**И все же, я убежден, что возможность увидеть, как делается колбаса,  
и заглянуть по ту сторону абстракций, обеспечиваемых интерпретатором  
Python, весьма информативна. Понимание, по крайней мере, некоторых  
внутренних механизмов поможет вам писать более производительный  
код (когда производительность имеет значение). И это также обеспечит  
массу удовольствия.**

**В качестве подопытного образца давайте возьмем простую функцию  
greet(), с которой можно поэкспериментировать и которую можно ис-  
пользовать, чтобы разобраться в байткоде Python:**def greet(name):

**return 'Привет, ' + name + '!'**

**>>> greet('Гвидо')**

***'Привет, Гвидо!'***

**Если помните, я уже отмечал, что Python сначала транслирует наш ис-  
ходный код в промежуточный язык, прежде чем он его «выполнит». Так  
вот, если это правда, то мы должны увидеть результаты этого шага ком-  
пиляции. И мы можем.**

**Каждая функция имеет атрибут code (в Python 3), который мы мо-**

**жем использовать, чтобы получить инструкции виртуальной машины,  
константы и переменные, используемые нашей функцией greet:**

**>>> greet. code .co\_code**

**b'dx0l|x00x17x00dx02x17x00Sx00'**

**>>> greet. code .co\_consts**

**(None, 'Привет, ', '!')**

**>>> greet. code .co\_varnames**

**('name',)**

**Вы видите, что co\_consts содержит части строки приветствия, которую  
собирает наша функция. Константы и код хранятся отдельно, чтобы сэко-  
номить пространство памяти. Константы... как бы сказать^ константны,  
то есть они не подлежат изменению и используются попеременно в раз-  
ных местах.**

**Поэтому вместо того, чтобы повторять фактические постоянные вели-  
чины в потоке команд co\_code, Python хранит константы отдельно в по-  
исковой таблице. Поток команд затем может ссылаться на константу  
по индексу в поисковой таблице. То же самое верно и для переменных,  
хранящихся в поле co\_varnames.**

**Надеюсь, что этот общий принцип начинает проясняться. Но рассмотре-  
ние потока команд co\_code по-прежнему заставляет меня чувствовать  
себя нехорошо. Этот промежуточный язык явно предназначен для того,  
чтобы с ним было легко работать виртуальной машине Python, а не  
людям. В конце концов, для это существует текстоориенированный ис-  
ходный код.**

**Разработчики, работающие над CPython, тоже это поняли. Поэтому они  
дали нам еще один инструмент, который называется дизассемблером,  
чтобы сделать инспектирование байтокда легче.**

**Дизассемблер байткода Python располагается в модуле dis, который яв-  
ляется составной частью стандартной библиотеки. Поэтому мы можем  
его просто импортировать и вызвать dis.dis() с функцией greet в каче-  
стве аргумента, чтобы получить более удобочитаемое представление о ее  
байткоде:**

**>>> import dis  
>>> dis.dis(greet)**

1. **('Привет, ')  
   0 (name)**
2. **('!')**

**2 0 LOAD\_CONST**

**2 LOAD\_FAST  
4 BINARY\_ADD  
6 LOAD\_CONST  
8 BINARY\_ADD  
10 RETURN VALUE**

**Главное, что сделал дизассемблер, было разбиение потока команд и на-  
значение каждому находящемуся в нем коду операции человекочитаемого  
имени, как, например, LOAD\_CONST.**

**Вы также видите, как ссылки на константы и переменные теперь череду-  
ются с байткодом и выведены полностью, чтобы уберечь нас от мозговой  
гимнастики относительно поиска по таблице co\_const или co\_varnames.  
Круто!**

**Глядя на человекочитаемые коды операций, мы начинаем понимать, как  
Python представляет и исполняет выражение 'Привет, ' + name + '!' в ис-  
ходной функции greet().**

**Сначала он извлекает константу в индексе 1 ('Привет, ') и помещает ее  
в стек. Затем он загружает содержимое переменной name и также поме-  
щает ее в стек.**

**Стек является структурой данных, которая используется в качестве  
внутренней рабочей памяти виртуальной машины. Существуют разные  
классы виртуальных машин, и один из них называется стековой маши-  
ной. Виртуальная машина Python является реализацией такой стековой  
машины. Если вся эта штука названа в честь стека, то вы можете себе  
представить, какую важную роль играет эта структура данных.**

**Между прочим, здесь я касаюсь лишь верхов. Если вы интересуетесь  
этой темой, то дальше найдете рекомендации для дальнейшего изучения.  
Более глубокое ознакомление с принципами работы виртуальных машин  
открывает глаза на многое (а еще это весело).**

**Самое интересное относительно стека как абстрактной структуры данных  
состоит в том, что на минимальном уровне он поддерживает всего две опе-  
рации: вталкивание (push) и выталкивание (pop). Вталкивание добавляет  
значение на вершину стека, а выталкивание удаляет и возвращает самое  
верхнее значение. В отличие от массива, в стеке отсутствует способ полу-  
чить доступ к элементам «ниже» верхнего уровня.**

**Просто поразительно, что такая простая структура данных пользуется  
столь большой популярностью. Однако я увлекся.**

**Давайте предположим, что вначале стек пустой. После того как первые два  
кода операции были исполнены, содержимое стека виртуальной машины  
будет выглядеть следующим образом (0 — это самый верхний элемент):**

**0: 'Гвидо' (содержимое "name")**

**1: 'Привет, '**

**Инструкция BINARY\_ADD выталкивает два строковых значения из стека,  
конкатенирует их, а затем вталкивает результат снова в стек:**

**0: 'Привет, Гвидо'**

**Затем идет еще одна инструкция LOAD\_CONST, которая помещает в стек  
строку с восклицательным знаком:**

**0: '!'**

**1: 'Привет, Гвидо'**

**Следующий код операции BINARY\_ADD снова объединяет два значения,  
чтобы сгенерировать заключительную приветственную строку:**

**0: 'Привет, Гвидо!'**

**Последняя байткодовая инструкция — RETURN\_VALUE, которая сообщает  
виртуальной машине следующее: то, что в настоящее время находится на  
вершине стека, является возвращаемым значением этой функции, и по-  
этому оно может быть передано источнику вызова.**

**И — вуаля! — мы только что проследили за тем, как наша функция greet()  
была исполнена на внутреннем уровне виртуальной машиной Python.  
Разве не круто?**

**Можно еще много рассказывать о виртуальных машинах, но эта книга по-  
священа не им. Однако если эта захватывающая тема вас заинтриговала,  
то настоятельно рекомендую заняться ее изучением.**

**Можно получить массу удовольствия от создания и определения своих  
собственных байткодовых языков и построения для них небольших экс-  
периментов с использованием виртуальной машины. По этой теме я по-  
рекомендовал бы книгу Проектирование компиляторов: виртуальные  
машины (Compiler Design: Virtual Machines, Wilhelm and Seidl).**

**Ключевые выводы**

* **СPython исполняет программы, сначала транслируя их в промежу-  
  точный байткод, а затем выполняя байткод на виртуальной машине  
  со стековой архитектурой.**
* **Вы можете использовать встроенный модуль dis, чтобы заглянуть за  
  кулисы и проинспектировать байткод.**
* **Займитесь плотнее виртуальными машинами — оно того стоит.**

**9**

**Итоги**

**Примите поздравления — вы прошли весь путь до самого конца! Самое  
время похлопать себя по плечу, поскольку большинство людей покупают  
книгу и даже ее не открывают или не доходят до конца первой главы.**

**Но теперь, когда вы прочитали эту книгу, начинается настоящая работа.  
Ведь, как известно, читать и делать — это две большие разницы. Возьми-  
те новые навыки и идиомы, которые вы узнали из этой книги, и начните  
их использовать на практике. Не позвольте этой книге стать просто оче-  
редной прочитанной книгой по программированию.**

**Что, если с этого момента вы начнете усеивать свой программный код рас-  
ширенными функциональными возможностями языка Python? Изящное  
и чистое выражение-генератор тут, элегантное применение инструкции  
with там\_**

**В мгновение ока вы привлечете внимание своих товарищей — и по хо-  
рошему поводу, если вы все сделаете правильно. Когда вы наработаете  
небольшой опыт, у вас не будет никаких затруднений в правильном  
применении этих продвинутых функциональных средств Python и ис-  
пользовании их только там, где они имеют смысл и помогают делать про-  
граммный код выразительнее.**

**И поверьте, через некоторое время ваши коллеги подхватят тренд. Если  
они задают вам вопросы, делитесь с ними полезными знаниями. Подтя-  
гивайте окружающих и помогайте им. Возможно, через пару недель вы  
даже устроите коллегам небольшую презентацию по теме «написания  
чистого Python». Не стесняйтесь использовать мои примеры из этой  
книги.**

**Для разработчика Python есть разница между выполнением отличной  
работы и тем, как выполнение отличной работы смотрится со стороны.  
Не бойтесь выделиться. Если вы поделитесь своими навыками и вновь  
приобретенными знаниями с окружающими, то ваша карьера от этого  
только выиграет.**

**В своей работе и проектах я следую тем же самым принципам. И поэто-  
му я всегда ищу способы улучшить эту книгу и другие мои обучающие  
материалы по языку Python. Если вы хотите сообщить мне об ошибке,  
у вас просто есть вопрос или же вы хотите дать какой-то конструктивный  
отзыв, то пишите мне по адресу mail@dbader org.**

**Успешного программирования на Python!**

— ***Дэн Бейдер***

**P. S. Навестите меня в Сети и обязательно продолжите свою экскурсию по  
Python на dbader. org и на моем канале YouTube[[73]](#footnote-73). Кроме того, непременно  
получите бесплатный экземпляр Трюки Python: цифровой комплект ин-  
струментов, доступный по адресу dbader. org/python-tricks-toolkit .**

1. **. 1 . Бесплатные еженедельные советы  
   для разработчиков на Python**

**Хотите еженедельную порцию советов для разработчика на Python, чтобы  
улучшить свою производительность и оптимизировать свой рабочий про-  
цесс? Есть хорошие новости! Я веду бесплатную электронную рассылку  
для таких, как вы, разработчиков на Python.**

**Электронные письма, которые я рассылаю, не являются обычными со-  
общениями в стиле «а вот и список популярных статей». Вместо этого  
я стремлюсь делиться по крайней мере одной оригинальном мыслью  
в неделю в формате короткого эссе.**

**Если вы хотели бы увидеть, из-за чего же такой ажиотаж, то отправляй-  
тесь прямиком на dbader org/newsletter и впишите свой адрес электронной  
почты в регистрационной форме. С нетерпением жду встречи!**

**9.2 . PythonistaCafe: сообщество  
разработчиков на Python**

**Освоение языка Python — не только про то, как доставать книги и курсы  
для учебы. Чтобы быть успешным, вам также нужен способ оставаться  
мотивированным и в конечном счете развивать свои способности.**

**Многие питонисты, которых я знаю, с трудом с этим справляются. Слож-  
но развивать свой опыт программирования на Python в полном одиноче-  
стве.**

**Если вы разработчик-самоучка с нетехнической работой на полный день,  
то весьма трудно развивать свои навыки самостоятельно. Особенно если  
среди вашего окружения нет кодеров и тех, кто мог бы вас подбодрить  
или поддержать в ваших усилиях стать лучше в области программиро-  
вания.**

**Возможно, вы уже работаете разработчиком, но в вашей компании никто  
больше не разделяет вашу любовь к Python. Печально, когда вы ни с кем  
не можете поделиться своими достижениями в изучении языка или по-  
просить совета, когда вы чувствуете, что зашли в тупик.**

**Из личного опыта знаю, что и в существующих онлайн-сообществах,  
и в социальных сетях тоже не очень получается обеспечивать эту под-  
держку. Вот несколько самых лучших сообществ, но даже они все еще  
оставляют желать лучшего:**

**□ Веб-сайт Stack Overflow в формате FAQ, предназначен для четких  
разовых вопросов. На платформе трудно установить человеческие  
отношения с коллегами-комментаторами. Все подчинено фактам, а не  
людям. Например, модераторы свободно редактируют вопросы, ответы**

**и комментарии других людей. Он больше похож на Википедию, чем  
на форум.**

* **Социальная сеть Twitter похожа на виртуальный кулер, у которого  
  можно поболтаться в перерыв, но она ограничена 140 знаками на одно  
  сообщение, что не особо хорошо для обсуждения чего-либо существен-  
  ного. Кроме того, если вы не будете постоянно в Сети, то пропустите  
  большую часть разговоров. А если вы постоянно в Сети, то ваша про-  
  изводительность пострадает от бесконечного потока уведомлений.  
  Слабые чат-группы страдают теми же самыми недостатками.**
* **Социальный новостной сайт Hacker News предназначен для обсужде-  
  ния и комментирования технических новостей. Он не способствует  
  установлению долговременных отношений между комментаторами.  
  Кроме того, на сегодня это одно из самых агрессивных сообществ в тех-  
  нологической сфере со слабой модерацией и пограничной токсичной  
  культурой.**
* **Социальный новостной сайт Reddit занимает более широкую позицию  
  и поощряет более «человеческие» обсуждения, чем разовый формат  
  вопросов и ответов сайта Stack Overflow. Вместе с тем это огромный  
  форум с миллионами пользователей, который имеет все связанные  
  с этим проблемы: токсичное поведение, властный негативизм, набра-  
  сывающиеся друг на друга люди, ревность^ Короче говоря, все «самые  
  лучшие» проявления человеческого поведения.**

**В итоге я понял, что разработчиков сдерживает только их ограниченный  
доступ к глобальному сообществу программистов на Python. Вот почему  
я основал PythonistaCafe, образовательное сообщество для разработчиков  
на языке Python, где люди могут быть на равных.**

**©PYTHONISTACAFE**

**Сообщество PythonistaCafe хорошо рассматривать как клуб взаимного  
совершенствования для энтузиастов Python.**

**В PythonistaCafe вы будете взаимодействовать с профессиональными  
разработчиками и любителями со всего мира, которые делятся опытом  
в безопасном окружении, чтобы учиться у них и избегать тех же самых  
ошибок, которые они совершили.**

**Задайте любой вопрос, который хотите, и он останется приватным. Чтобы  
читать и писать комментарии, у вас должен быть статус активного за-  
регистрированного участника, и, так как сообщество платное, троллинга  
и оскорбительного поведения там не существует.**

**Люди, которых вы встречаете внутри сообщества, активно стремятся  
улучшать свои навыки программирования на Python, потому что членство  
в PythonistaCafe возможно только по приглашению. Все потенциальные  
участники обязаны подавать заявки, так мы можем увидеть, что они под-  
ходят нашему сообществу.**

**Вы будете приглашены в сообщество, которое понимает вас, ваши навыки  
и карьеру, которую вы строите, и то, чего вы пытаетесь достигнуть. Если  
вы пытаетесь развить свои навыки программирования на Python, но не  
нашли систему поддержки, которая вам нужна, то мы существуем для вас.**

**Сообщество PythonistaCafe основано на частной дискуссионной плат-  
форме, где можно задавать вопросы, получать ответы и делиться успе-  
хом. Наши участники есть по всему миру и обладают разными уровнями  
мастерства.**

**Подробнее о сообществе PythonistaCafe, ценностях нашего сообщества,  
и о том, кто мы такие, вы можете узнать на www . pythonistacafe . com .**

***Дэн Бейдер***

**Чистый Python. Тонкости программирования для профи**

***Ю. Сергиенко  
К. Тульцева  
Д. Дрошнев  
С. Заматевская  
Н. Викторова, И. Тимофеева  
Л. Егорова***

**Заведующая редакцией  
Ведущий редактор  
Литературный редактор  
Художественный редактор  
Корректоры  
Верстка**

***Перевел с английского А. Логунов***

**Изготовлено в России. Изготовитель: ООО «Прогресс книга».**

**Место нахождения и фактический адрес: 194044, Россия, г Санкт-Петербург,**

**Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.**

**Дата изготовления: 07.2018. Наименование: книжная продукция. Срок годности: не ограничен.**

**Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —**

**Книги печатн^іе профессиональные, технические и научн^іе.**

**Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.  
Подписано в печать 27.06.18. Формат 70^100/16. Бумага офсетная. Усл. п. л. 23,220. Тираж 1000. Заказ 0000.  
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатн^ій Двор».**

**142300, Московская область, г. Чехов, ул. Полиграфистов, 1.**

**Сайт:** [**www.chpk.ru**](http://www.chpk.ru)**. E-mail:** [**marketing@chpk.ru**](mailto:marketing@chpk.ru) **Факс: 8(496) 726-54-10, телефон: (495) 988-63-87**

**І^ППТЕР^**

[**WWW.PITER.COM**](http://WWW.PITER.COM)

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает  
профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: **м. «Выборгская», Б. Сам пеон иевский пр., д. 29а  
тел./факс: (812) 703-73-83, 703-73-72; e-mail:** [**sales@piter.eom**](mailto:sales@piter.eom)

Москва: **м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж  
тел./факс: (495) 234-38-15; e-mail:** [**sales@msk.piter.com**](mailto:sales@msk.piter.com)

Воронеж: **тел.: 8 951 861-72-70; e-mail:** [**hitsenko@piter.com**](mailto:hitsenko@piter.com)

Екатеринбург: **ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;  
e-mail:** [**office@ekat.piter.com**](mailto:office@ekat.piter.com)**; skype: ekat.manager2**

Нижний Новгород: **тел.: 8 930 712-75-13; e-mail:** [**yashny@yandex.ru**](mailto:yashny@yandex.ru)**; skype: yashnyl**

Ростов-на-Дону: **ул. Ульяновская, д. 26  
тел./факс: (863) 269-91-22, 269-91-30; e-mail:** [**piter-ug@rostov.piter.com**](mailto:piter-ug@rostov.piter.com)

Самара: **ул. Молодогвардейская, д. 33а, офис 223  
тел./факс: (846) 277-89-79, 277-89-66; e-mail:** [**pitvolga@mail.ru**](mailto:pitvolga@mail.ru)**,**[**pitvolga@samara-ttk.ru**](mailto:pitvolga@samara-ttk.ru)

БЕЛАРУСЬ

Минск: **ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;**

**e-mail:** [**og@minsk.piter.com**](mailto:og@minsk.piter.com)

Издательский дом «Питер» приглашает **к** сотрудничеству авторов:

**тел./факс: (812) 703-73-72, (495) 234-38-15; е-таіі:** [**ivanovaa@piter.com**](mailto:ivanovaa@piter.com) **Подробная информация здесь:** [**http://www.piter.com/page/avtoru**](http://www.piter.com/page/avtoru)

Издательский дом «Питер» приглашает к сотрудничеству зарубежных  
торговых партнеров или посредников, имеющих выход на зарубежный  
рынок: ***тел./факс: (812) 703-73-73; е-таіІ:*** [sales@piter.com](mailto:sales@piter.com)

Заказ книг для вузов и библиотек:

**тел./факс: (812) 703-73-73, доб. 6243; е-таіІ:** [**uchebnik@piter.com**](mailto:uchebnik@piter.com)

*Заказ книг по почте:* **на сайте wvm.piter.com; тел.: (812) 703-73-74, доб. 6216;  
е-таіІ:** [**books@piter.com**](mailto:books@piter.com)

*Вопросы по продаже электронных книг:* **тел.: (812) 703-73-74, доб. 6217;  
е-таіІ:** [**Kuznetsov@piter.com**](mailto:Kuznetsov@piter.com)



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»

МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

* **на нашем сайте:** [**www.piter.com**](http://www.piter.com)
* **по электронной почте:** [**books@piter.com**](mailto:books@piter.com)
* **по телефону:** (812) 703-73-74

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

**^ Наложенным платежом с оплатой при получении в ближайшем  
почтовом отделении.**

**^ С помощью банковской карты. Во время заказа вы будете**

**перенаправлены на защищенный сервер нашего оператора, где сможете  
ввести свои данные для оплаты.**

**@ Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги,  
Webmoney и Kiwi-кошелек.**

**^ В любом банке, распечатав квитанцию, которая формируется**

**^ автоматически после совершения вами заказа.**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

* **Посылки отправляются через «Почту России». Отработанная  
  система позволяет нам организовывать доставку ваших покупок  
  максимально быстро. Дату отправления вашей покупки и дату  
  доставки вам сообщат по e-mail.**
* **Вы можете оформить курьерскую доставку своего заказа (более  
  подробную информацию можно получить на нашем сайте** [**www.piter.com**](http://www.piter.com)**).**
* **Можно оформить доставку заказа через почтоматы (адреса почтоматов  
  можно узнать на нашем сайте** [**www.piter.com**](http://www.piter.com)**).**

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

* **фамилию, имя, отчество, телефон, e-mail;**
* **почтовый индекс, регион, район, населенный пункт, улицу, дом,  
  корпус, квартиру;**

**БЕСПЛАТНАЯ ДОСТАВКА:**

**курьером по Москве и Санкт-Петербургу  
при заказе на сумму** от 2000 руб.

**почтой России при предварительной оплате  
заказа на сумму** от 2000 руб.

* **название книги, автора, количество заказываемых экземпляров.**

**[^ППТЕР^**

[**WWW.PITER.COM**](http://WWW.PITER.COM)

ВАША УНИКАЛЬНАЯ КНИГА

**Хотите издать свою книгу? Она станет идеальным подарком для партнеров  
и друзей, отличным инструментом для продвижения вашего бренда, презентом  
для памятных событий! Мы сможем осуществить ваши любые, даже самые  
смелые и сложные, идеи и проекты.**

МЫ ПРЕДЛАГАЕМ:

* **издать вашу книгу**
* **издание книги для использования в маркетинговых активностях**
* **книги как корпоративные подарки**
* **рекламу в книгах**
* **издание корпоративной библиотеки**

Почему надо выбрать именно нас:

**Издательству «Питер» более 20 лет. Наш опыт - гарантия высокаго качества.***Мы предлагаем:*

* **услуги по обработке и доработке вашего текста**
* **современный дизайн от профессионалов**
* **высокий уровень полиграфического исполнения**
* **продажу вашей книги во всех книжных магазинах страны**

Обеспечим продвижение вашей книги:

* **рекламой в профильных СМИ и местах продаж**
* **рецензиями в ведущих книжных изданиях**
* **интернет-поддержкой рекламной кампании**

**Мы имеем сабственную сеть дистрибуции па всей России, а также на Украине  
и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами.  
Издательство «Питер» является постоянным участником многих конференций  
и семинаров, которые предоставляют широкую возможность реализации книг.**

**Мы обязательна проследим, чтобы ваша книга пастоянно имелась в наличии  
в магазинах и была выложена на самых видных местах.**

**Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн,  
любой тираж.**

**Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим  
ее в крупнейших интернет-магазинах. Книга будет сверстана в фармате еРиЬ  
или PDF - самых папулярных и надежных фарматах на сегадняшний день.**

Свяжитесь с нами прямо сейчас:

Санкт-Петербург - **Анна Титова, (812) 703-73-73,** [**titova@piter.cam**](mailto:titova@piter.cam)Москва - **Сергей Клебанов, (495) 234-38-15,** [**klebanov@piter.com**](mailto:klebanov@piter.com)

1. См. документацию Python «Инструкция assert»: <https://docs> . python . org/3/reference/  
   simple\_stmts . html%23the-assert-statement [↑](#footnote-ref-1)
2. См. Википедию: <https://en> .wikipedia .org/wiki/Heisenbug и <https://ru> .wikipedia . org/wiki/Гейзенбаг [↑](#footnote-ref-2)
3. См. документацию Python «Константы ( debug )»: <https://docs> . python . org/3/library/

   constants . html%23 debug [↑](#footnote-ref-3)
4. Нулевая операция (null-operation) — это операция, которая не возвращает данные  
   и оставляет состояние программы без изменений. См. <https://en> .wikipedia . org/wiki/NuN\_  
   function — Примеч. пер. [↑](#footnote-ref-4)
5. **Проверка has\_product() пропускается, когда assert отключена.** Это

   означает, что метод get\_product() теперь можно вызывать с недо-  
   пустимыми идентификаторами товаров, что может привести к более  
   серьезным ошибкам, — в зависимости от того, как написана наша про-  
   грамма. В худшем случае она может стать началом запуска DoS-атак. [↑](#footnote-ref-5)
6. Я написал статью о том, как в своих тестах Python можно избежать поддельных утверж-  
   дений. Ее можно найти тут: dbader. org/blog/catching-bogus-python-asserts [↑](#footnote-ref-6)
7. См. PEP8: «Руководство по стилю оформления исходного кода Python»: <https://www>.  
   python . org/dev/peps/pep-0008/ [↑](#footnote-ref-7)
8. См. документацию Python «Импортирование с подстановочным знаком \* из пакета»:  
   <https://docs> . python . org/3/tutorial/modules . html#importing-from-a-package [↑](#footnote-ref-8)
9. См. PEP 8 «Импортирование»: <http://pep8> . org/#imports [↑](#footnote-ref-9)
10. См. документацию Python «Форматирование строк в стиле printf»: <https://docs> . python .  
    org/3/library/stdtypes . html#printf-style-string-formatting [↑](#footnote-ref-10)
11. См. документацию Python «str.format()»: <https://docs> . python . org/3/library/string . html#format-  
    string-syntax [↑](#footnote-ref-11)
12. См. документацию Python «Синтаксис форматной строки»: <https://docs> . python, org/3/  
    library/string . html#format-string-syntax [↑](#footnote-ref-12)
13. См. Python 3. Спорный вопрос в трекере ошибок № 27078: <https://bugs> . python . org/  
    issue27078 [↑](#footnote-ref-13)
14. См. документацию Python «Форматированные строковые литералы»: https://  
    docs . python . org/3/reference/lexical\_analysis . html#formatted-string-literals [↑](#footnote-ref-14)
15. Язык Python создал нидерландский программист Гвидо ван Россум. [↑](#footnote-ref-15)
16. См. документацию Python «Объекты, значения и типы»: <https://docs> . python . org/3/reference/  
    datamodel . html#objects-values-and-types [↑](#footnote-ref-16)
17. Начиная с Python 3.3, также имеется атрибут qualname , который служит для такой

    же цели и обеспечивает строку с квалифицированным именем для устранения неодно-  
    значности между именами функций и классов (см. PEP 3155: <https://www>. python . org/dev/  
    peps/pep-3155/). [↑](#footnote-ref-17)
18. См. раздел 3.4 «Веселье с \*args и \*\*kwargs». [↑](#footnote-ref-18)
19. См. документацию Python «functools»: <https://docs> . python . org/3/library/functools . html [↑](#footnote-ref-19)
20. См. раздел 3.5 «Распаковка аргументов функции». [↑](#footnote-ref-20)
21. DRY (от англ. Don’t Repeat Yourself, то есть «не повторяйся») — это принцип разработки  
    программного обеспечения, нацеленный на снижение повторения информации различ-  
    ного рода. См. <https://ru> . wikipedia . org/wiki/Don't\_repeat\_yourself. — Примеч. пер. [↑](#footnote-ref-21)
22. См. документацию Python 2 «Модель данных»: <https://docs> . python, org/2/reference/  
    datamodel . html [↑](#footnote-ref-22)
23. См. документацию Python «Операции мелкого и глубокого копирования»: <https://docs> .  
    python . org/3/library/copy. html [↑](#footnote-ref-23)
24. См. документацию Python «Модуль abc»: <https://docs> . python . org/3/library/abc . html [↑](#footnote-ref-24)
25. См. документацию Python «@classmethod»: <https://docs> . python . org/3/library/functions .  
    html#classmethod [↑](#footnote-ref-25)
26. См. документацию Python «@staticmethod»: <https://docs> . python, org/3/library/functions .  
    html#staticmethod [↑](#footnote-ref-26)
27. См. Википедию: «Фабрика (объектно-ориентированное программирование)»: https://  
    en . Wikipedia . org/wiki/Factory\_(object-oriented\_programming) и <https://ru> . wikipedia . org/wiki/Аб-  
    страктная\_фабрика\_(шаблон\_проектирования) [↑](#footnote-ref-27)
28. См. документацию Python «Ассоциативные типы — dict»: <https://does>. python .org/3/library/  
    stdtypes . html#mapping-types-diet [↑](#footnote-ref-28)
29. См. глоссарий документации Python «hashable»: <https://does> . python . org/3/glossary. html [↑](#footnote-ref-29)
30. См. документацию Python «collections.OrderedDict»: <https://docs> . python, org/3/library/  
    coNections . html#coNections . OrderedDict [↑](#footnote-ref-30)
31. См. список рассылки CPython: <https://mail>. python, org/pipermail/python-dev/2016-  
    September/146327. html [↑](#footnote-ref-31)
32. См. документацию Python «collections.ChainMap»: <https://docs> . python, org/3/library/  
    collections . html#coNections . ChainMap [↑](#footnote-ref-32)
33. См. документацию Python «types.MappingProxyType»: <https://docs> . python . org/3/library/  
    types html [↑](#footnote-ref-33)
34. См. документацию Python «list»: <https://docs> . python . org/3/tutorial/introduction . html#lists  
    и <https://docs> . python . org/3/tutorial/datastructures . html#more-on-lists [↑](#footnote-ref-34)
35. См. документацию Python «str»: <https://docs> . python, org/3.6/library/stdtypes . html#text-  
    sequence-type-str [↑](#footnote-ref-35)
36. См. документацию Python «bytearray»: <https://docs> . python, org/3/library/stdtypes .  
    html#bytearray [↑](#footnote-ref-36)
37. См. www. numpy. org [↑](#footnote-ref-37)
38. См. <https://pandas> . pydata . org/ [↑](#footnote-ref-38)
39. См. раздел «Словари, ассоциативные массивы и хеш-таблицы» настоящей главы. [↑](#footnote-ref-39)
40. См. документацию Python «tuple»: <https://docs> . python, org/3/tutorial/datastructures .  
    html#tuples-and-sequences [↑](#footnote-ref-40)
41. См. CPython: «tupleobject.c» (<https://github> . com/python/cpython/blob/master/Objects/  
    tupleobject . c) и «listobject.c» (<https://github> . com/python/cpython/blob/master/Objects/listobject . c) [↑](#footnote-ref-41)
42. См. раздел «Преобразование строк (каждому классу по repr )» главы 4. [↑](#footnote-ref-42)
43. См. документацию Python «property»: <https://docs>. python . org/3/library/functions. html#property [↑](#footnote-ref-43)
44. См. раздел «Чем полезны именованные кортежи» главы 4. [↑](#footnote-ref-44)
45. См. документацию Python «typing.NamedTuple»: <https://docs>. python . org/3.6/library/typing .  
    html [↑](#footnote-ref-45)
46. См. mypy-lang.org [↑](#footnote-ref-46)
47. См. документацию Python «struct.Struct»: <https://docs> . python, org/3/library/struct .  
    html#module-struct [↑](#footnote-ref-47)
48. См. документацию Python «types.SimpleNamespace»: <https://docs> . python . org/3.3/library/  
    types . html [↑](#footnote-ref-48)
49. См. <https://wiki> . python . org/moin/TimeComplexity [↑](#footnote-ref-49)
50. См. документацию Python «set»: <https://docs> . python . org/3/tutorial/datastructures . html#sets [↑](#footnote-ref-50)
51. См. документацию Python «hashable»: <https://docs> . python . org/3/glossary. html [↑](#footnote-ref-51)
52. См. документацию Python «frozenset»: <https://docs> . python . org/3/library/stdtypes . html#frozenset [↑](#footnote-ref-52)
53. См. документацию Python «collections.Counter»: <https://docs> . python .org/3/library/coNections .  
    html#counter-objects [↑](#footnote-ref-53)
54. См. документацию Python «collections.deque»: <https://docs> . python . org/3.6/library/coNections .  
    html#coNections . deque [↑](#footnote-ref-54)
55. См. CPython «\_collectionsmodule.c»: <https://github> . com/python/cpython/blob/master/Modules/\_  
    collectionsmodule . c [↑](#footnote-ref-55)
56. См. документацию Python «collections.deque»: <https://docs> . python . org/3.6/library/coNections .  
    html#coNections . deque [↑](#footnote-ref-56)
57. См. CPython «\_collectionsmodule.c»: <https://github> . com/python/cpython/blob/master/Modules/\_  
    collectionsmodule . c [↑](#footnote-ref-57)
58. См. документацию Python «queue.Queue»: <https://docs> . python, org/3.6/library/queue .  
    html#module-queue [↑](#footnote-ref-58)
59. См. документацию Python «multiprocessing.Queue»: <https://docs> . python . org/3.6/library/  
    multiprocessing . html#multiprocessing . Queue [↑](#footnote-ref-59)
60. См. Википедию «Полная упорядоченность»: <https://en> . wikipedia. org/wiki/Total\_order  
    и <https://ru> . wikipedia . org/wiki/Линейно\_упорядоченное\_множество [↑](#footnote-ref-60)
61. См. документацию Python «bisect.insort»: <https://docs> . python . org/3.6/library/bisect. html [↑](#footnote-ref-61)
62. См. документацию Python «heapq»: <https://docs> . python . org/3.6/library/heapq . html [↑](#footnote-ref-62)
63. См. документацию Python «Примечания к реализации очереди с приоритетом»: там же. [↑](#footnote-ref-63)
64. См. документацию Python «queue.PriorityQueue»: <https://docs>. python .org/3.6/library/queue .

    html [↑](#footnote-ref-64)
65. Чтобы получить такое экономное для оперативной памяти поведение в Python 2, вам  
    придется использовать встроенную функцию xrange(), так как функция range() будет  
    в действительности конструировать объект-список. [↑](#footnote-ref-65)
66. Термин list comprehension также переводится не совсем удобным термином «списковое  
    включение». Дело в том, что в Python, помимо включения собственно в список, еще  
    существуют конструкции включения в словарь (dictionary comprehension) и включения  
    в множество (set comprehension). — Примеч. пер. [↑](#footnote-ref-66)
67. См. глоссарий Python «EAFP»: <https://docs> . python . org/3.6/glossary. html?highlight=glossary [↑](#footnote-ref-67)
68. См. документацию Python «dict.get()»: <https://docs> . python . org/3/tutorial/datastructures .  
    html#dictionaries [↑](#footnote-ref-68)
69. См. документацию Python «Иерархия стандартных типов»: <https://docs> . python . org/3/  
    reference/datamodel . html#the-standard-type-hierarchy [↑](#footnote-ref-69)
70. См. PEP 448 «Дополнительные обобщения распаковки»: <https://www>. python .org/dev/peps/  
    pep-0448/ [↑](#footnote-ref-70)
71. В Windows команда activate выполняется напрямую, то есть ее не нужно загружать  
    вместе с источником. [↑](#footnote-ref-71)
72. См. <https://dbader>. org/products/managing-python-dependencies/?utm\_source=python-tricks-  
    book&utm\_medium=pdf&utm\_campaign=pytricks-book [↑](#footnote-ref-72)
73. См. <https://dbader>. org/youtube [↑](#footnote-ref-73)