БИБЛИОТЕКА ПРОГРАММИСТА

Дэн Бейдер

ЧИСТЫЙ PYTHON

ТОНКОСТИ ПРОГРАММИРОВАНИЯ ДЛЯ ПРОФИ

ПИТЕР®

Python Tricks: The Book

Dan Bader

Дэн Бейдер

ЧИСТЫЙ PYTHON

ТОНКОСТИ ПРОГРАММИРОВАНИЯ ДЛЯ ПРОФИ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

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 сосредоточиться на практических навыках, которые действительно важны. Раскопайте «скрытое золото» в стандартной библиотеке Python и начните писать чистый код уже сегодня.

Если у вас есть опыт работы со старыми версиями Python, вы сможете ускорить работу с современными 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 авторских прав.

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

ISBN 978-1775093305 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

Глава 5. Общие структуры данных Python 153

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

«Мне эта книга безумно нравится. Она похожа на репетитора, который 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 фантастического 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 словари», «лямбды» и «генераторы». Я всегда впечатлен и даже поражен тем, насколько же Python мощный, когда вы владеете этими приемами и можете их правильно реализовать.

Ваша книга стала именно той, которая нужна, чтобы превратиться 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 образование. Для меня огромное удовольствие читать эту книгу. И кроме того, я также подписан на электронную рассылку, что и помогло мне выйти на это издание».

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

«Впервые о вашей книге я услышал от коллеги, который хотел запутать 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 использовал 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 главах для меня тоже не новы, но, признаюсь, читая книгу, я все время узнаю что-нибудь новое из каждой главы, поэтому респект за написание очень приличной книги и за фантастическую работу по объяснению принципов, лежащих в основе всех тонкостей! С большим 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 важные 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 всесторонне обсуждаются соответствующие детали реализации. По-настоящему важно то, что эта книга реально позволяет вам писать программный код на Python лучше!

Благодаря этой книге я вдохновился новейшими практическими приемами 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 собственного 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 ресурсах 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 каком-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 написание 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 комментариев 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, гласящему, что должен существовать один и желательно только один очевидный 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*» представляет собой еще один великолепный ресурс. Она охватывает аспекты, лежащие за пределами языка программирования Python, а также содержит советы о том, как отлаживать свои программы, как управлять библиотеками, 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 напоминают 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 понять 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 переводчика 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 программированием на Python.

Базовый набор библиотек для разработчика

В обычных условиях библиотеки Python можно скачать и установить из каталога библиотек Python `PyPi` (https://pypi.python.org/) с помощью менеджера пакетов `pip`. Однако следует учесть, что в ОС Windows для работы 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 библиотеки (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.lfd.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-разрядной 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 основополагающих 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 обучения. 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 графического интерфейса пользователя, требуется для работы инструментальной 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 информацию о наших книгах.

1 Введение

1.1. Что такое идиома Python

> **Идиома Python** (Python Trick) — короткий фрагмент исходного кода на Python, используемый как инструмент обучения. Идиома Python обучает 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» началась 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 распространять 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 серии 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 обращаться разработчики на Python. Они благодарили за то, что я довел до них ту часть языка, которая оставалась для них камнем преткновения. 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 помощью.

Именно тогда я решил удвоить ставку на моем эксперименте с идиомами Python и довел его до серии из порядка 30 электронных сообщений. Каждое 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 аудитории. 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 экспериментировать с ними.

По мере того как все больше разработчиков подписывалось на электронную 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 читателей или подсказывал им дополнительные ресурсы, где можно было бы расширить свое понимание.

Скажем так: этот аспект был еще одной большой областью для улучшения. Основная задача моего веб-сайта dbader.org состоит в том, чтобы помогать 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 коротких 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 эксперимент. Надеюсь, что вы получите удовольствие от ее прочтения и по ходу узнаете еще что-то о Python!
>
> — Дэн Бейдер

1.2. 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 поможет 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 начальных 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 основополагающие 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 законченный 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 разобраться в тонкостях языка Python. Вы обнаружите массу практических советов и шаблонов проектирования, которые сделают вас более эффективным и квалифицированным программистом на Python.

1.3. Как читать эту книгу

Оптимальный способ чтения книги «*Чистый Python. Тонкости программирования для профи*» — рассматривать ее как копилку потрясающих функциональных 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 заработала 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 называю «*Тонкости Python: цифровой комплект инструментов*»[1].

Среди всего прочего этот комплект инструментов включает 12 видеоуроков 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.

[1] См. https://dbader.org/python-tricks-toolkit

2 Шаблоны для чистого Python

2.1. Прикрой свой з** инструкциями assert

Иногда по-настоящему полезное функциональное средство языка привлекает меньше внимания, чем оно того заслуживает. По некоторым причинам это именно то, что произошло со встроенной в Python инструкцией `assert`.

В этой главе я собираюсь дать вам представление об использовании `assert` в Python. Вы научитесь ее применять для автоматического обнаружения 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 выполняться 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 работаете 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 сниженные цены не могут быть ниже 0 $ и они не могут быть выше первоначальной цены товара.

Давайте убедимся, что эта функция действительно работает как задумано, если вызвать ее, применив допустимую скидку. В этом примере товары в нашем магазине будут представлены в виде простых словарей. 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 неплохое 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 применить 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']
AssertionError
```

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 вызвавшее 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 посмотрев на отчет об обратной трассировке исключения.

Это значительно ускорит отладку и в дальнейшем сделает ваши программы 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 сообщать 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 попробовать еще раз.

Инструкции призваны быть *внутренними самопроверками* (internal self-checks) вашей программы. Они работают путем объявления неких условий, 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 исключением `AssertionError`, говорящим, какое именно «невозможное» условие 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 разработчикам 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 практически реализуется в Python. Поэтому давайте бегло взглянем на синтаксис инструкции `assert` в соответствии с документацией Python[1]:

```
инструкция_assert ::= "assert" выражение1 ["," выражение2]
```

[1] См. документацию Python «Инструкция `assert`»: https://docs.python.org/3/reference/simple_stmts.html%23the-assert-statement

В данном случае выражение1 — это условие, которое мы проверяем, а необязательное выражение2 — это сообщение об ошибке, которое выводится на экран, если утверждение дает сбой. Во время исполнения программы интерпретатор Python преобразовывает каждую инструкцию `assert` примерно в следующую ниже последовательность инструкций:

```
if __debug__:
     if not выражение1:
         raise AssertionError(выражение2)
```

В этом фрагменте кода есть две интересные детали.

Перед тем как данное условие инструкции `assert` будет проверено, проводится дополнительная проверка глобальной переменной `__debug__`. Это встроенный булев флажок, который при нормальных обстоятельствах имеет значение `True`, — и значение `False`, если запрашивается оптимизация. Мы поговорим об этом подробнее чуть позже в разделе, посвященном «распространенным ловушкам».

Кроме того, вы можете применить выражение2, чтобы передать необязательное 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 плавающей ошибкой Гейзенбаг[1].

[1] См. Википедию: https://en.wikipedia.org/wiki/Heisenbug и https://ru.wikipedia.org/wiki/Гейзенбаг

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, связанных 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 предостережения 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 отключены[1] переключателями командной строки `-O` и `-OO`, а также переменной окружения `PYTHONOPTIMIZE` в CPython.

Это превращает любую инструкцию `assert` в нулевую операцию: утверждения `assert` просто компилируются и вычисляться не будут, это означает, что ни одно из условных выражений не будет выполнено[2].

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

---

[1] См. документацию Python «Константы (`__debug__`)»: https://docs.python.org/3/library/constants.html%23__debug__

[2] Нулевая операция (null-operation) — это операция, которая не возвращает данные и оставляет состояние программы без изменений. См. https://en.wikipedia.org/wiki/Null_function — *Примеч. пер.*

Поясню: если в вашей программе утверждения `assert` используются для проверки того, содержит ли аргумент функции «неправильное» или неожиданное 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 проблему. И снова представьте, что вы создаете приложение Python с интернет-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 произойдет, 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 интерпретаторе Python, то проверка полномочий превращается в нулевую операцию. И поэтому *теперь любой пользователь может удалять товары*. Проверка полномочий вообще не выполняется. В результате повышается 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.

2. **Проверка `has_product()` пропускается, когда `assert` отключена.** Это означает, что метод `get_product()` теперь можно вызывать с недопустимыми идентификаторами товаров, что может привести к более серьезным ошибкам, — в зависимости от того, как написана наша программа. В худшем случае она может стать началом запуска DoS-атак.

Например, если приложение магазина аварийно завершается при попытке 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 показано ниже:

```
def delete_product(product_id, user):
     if not user.is_admin():
         raise AuthError('Для удаления необходимы права администратора')
     if not store.has_product(product_id):
         raise ValueError('Идентификатор неизвестного товара')
     store.get_product(product_id).delete()
```

Этот обновленный пример также обладает тем преимуществом, что вместо того, чтобы вызывать неопределенные исключения `AssertionError`, он теперь вызывает семантически правильные исключения, а именно `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 понести 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 являются 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` станет *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`. Например, 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 модульных тестов имеется приведенное ниже утверждение:

```
assert (
     counter == 10,
     'Это должно было сосчитать все элементы'
)
```

На первый взгляд этот тестовый случай выглядит абсолютно приемлемым. Однако он никогда не выловит неправильный результат: это утверждение `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, — использовать линтер (linter), инструмент статического анализа кода[1]. 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.

[1] Я написал статью о том, как в своих тестах Python можно избежать поддельных утверждений. Ее можно найти тут: dbader.org/blog/catching-bogus-python-asserts

Инструкции assert — резюме

Несмотря на данные выше предостережения, я полагаю, что инструкции `assert` являются мощным инструментом отладки, который зачастую недостаточно 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 знания Python до более качественного уровеня и стать всесторонним питонистом. 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 помогать разработчикам идентифицировать ошибки. Они не являются механизмом обработки ошибок периода исполнения программы.
- Инструкции `assert` могут быть глобально отключены в настройках интерпретатора.

2.2. Беспечное размещение запятой

Вот вам полезный совет, когда вы добавляете и удаляете элементы из константы 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 сказать, 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 выделением многочисленных изменений, вносимых в одной-единственной строке.

Быстрым решением будет принятие стиля оформления кода, при котором вы разворачиваете константы списка, словаря или множества на нескольких 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 вызывать 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 нужно 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 «конкатенацией строковых литералов», является преднамеренным и задокументированным. И оно также предоставляет фантастическую возможность выстрелить себе в ногу, внося в ваши программы трудноотлавливаемые ошибки:

> Применение многочисленных смежных строковых или байтовых литералов (разделенных пробелом), в некоторых случаях с использованием разных согласованных правилами оформления кавычек, допустимо, и их значение идентично их конкатенации[1].

Вместе с тем в некоторых случаях конкатенация строковых литералов является 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 функциональное 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:

---

[1] См. документацию Python «Конкатенация строковых литералов»: https://docs.python.org/3/reference/lexical_analysis.html#string-literal-concatenation

```
>>> names = [
...     'Элис',
...     'Боб',
...     'Дилберт',
...     'Джейн'
]
```

Но теперь мы совершили полный круг и вернулись к изначальной проблеме. 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 маневра, 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 трудноотлавливаемые ошибки.

2.3. Менеджеры контекста и инструкция with

Некоторые разработчики считают инструкцию Python `with` малопонятным 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 содействует написанию более чистого и более удобочитаемого программного кода Python.

Итак, в чем же прелесть инструкции `with`? Она помогает упростить некоторые распространенные шаблоны управления ресурсами, абстрагируясь 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 стандартной библиотеке Python. Встроенная функция `open()` предоставляет превосходный вариант ее применения:

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

Существует общая рекомендация открывать файлы, используя инструкцию `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 фрагменту кода:

```
f = open('hello.txt', 'w')
try:
     f.write('привет, мир!')
finally:
     f.close()
```

Вы сразу можете сказать, что он довольно многословен. Обратите внимание: 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 реализация не будет гарантировать, что файл будет закрыт, и поэтому наша программа может допустить утечку дескриптора файла. Вот почему инструкция `with` имеет столь важное значение. Она превращает надлежащее получение и высвобождение ресурсов в пустяковую работу.

Еще одним хорошим примером, где инструкция `with` эффективно используется 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 абстрагироваться от большей части логики обработки ресурса. Вместо необходимости 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 высвобождение 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 применяться 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 функционировал 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 теперь поддерживает инструкцию `with` точно так же, как и первоначальный пример с функцией `open()`:

---

[1] См. документацию Python «Менеджеры контекста инструкции with»: https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers

```
>>> with ManagedFile('hello.txt') as f:
 ...     f.write('привет, мир!')
 ...     f.write('а 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 единственным способом поддержки инструкции `with` в Python. Служебный модуль `contextlib`[1] стандартной библиотеки обеспечивает еще несколько абстракций, 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`, переписанный 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('а теперь, пока!')
```

В данном случае `managed_file()` является генератором, который сначала получает ресурс. После этого он временно приостанавливает собственное

[1] См. документацию Python «contextlib»: https://docs.python.org/3/library/contextlib.html

исполнение и передает ресурс инструкцией `yield`, чтобы его использовал источник вызова. Когда источник вызова покидает контекст `with`, генератор продолжает выполняться до тех пор, пока не произойдут любые оставшиеся шаги очистки, после чего ресурс будет высвобожден и возвращен системе.

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

Оборотной стороной реализации на основе `@contextmanager` может являться то, что такая реализация требует некоторого вникания в продвинутые понятия языка Python, такие как декораторы и генераторы. Если чувствуете, что вам необходимо в них разобраться, то не стесняйтесь поменять 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 применению инструкции `with` подойти творчески, то для своих модулей и классов вы сможете определять удобные API.

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

```
with Indenter() as indent:
     indent.print('привет!')
     with indent:
         indent.print('здорово')
         with indent:
             indent.print('бонжур')
     indent.print('эй')
```

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

```
привет!
    здорово
        бонжур
эй
```

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 контекста 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 попробуйте реализовать менеджер контекста, измеряющий время исполнения блока программного кода с использованием функции `time.time`. Обязательно 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 инкапсуляции стандартных случаев применения инструкций `try/finally` в так называемые менеджеры контекста.
- Чаще всего менеджер контекста используется для управления безопасным получением и высвобождением системных ресурсов. Ресурсы выделяются при помощи инструкции `with` и высвобождаются автоматически, когда поток исполнения покидает контекст `with`.
- Эффективное применение инструкции `with` помогает избежать утечки ресурсов и облегчает ее восприятие.

2.4. Подчеркивания, дандеры и другое

У символов одинарного и двойного подчеркивания в Python есть особый смысл в именах переменных и методов. Отчасти этот смысл существует исключительно по договоренности и предназначен в качестве подсказки программисту — и частично он обеспечивается интерпретатором Python.

Если вам интересно, каков смысл символов одинарного и двойного подчеркивания в именах переменных и методов, то здесь я приложу все усилия, 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 договоренности 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 переменная 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[1].

Однако эта договоренность не обеспечивается интерпретатором Python. В Python отсутствует строгое разграничение между «приватными» и «публичными» 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 открытого интерфейса этого класса. Оставь-ка ее в покое*».

[1] См. PEP8: «Руководство по стилю оформления исходного кода Python»: https://www.python.org/dev/peps/pep-0008/

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 получить доступ к атрибутам `foo` и `_bar`, определенным в его конструкторе `__init__`?

Давайте узнаем:

```
>>> t = Test() >>> t.foo
11
>>> t._bar
23
```

Как видите, одинарный начальный символ подчеркивания в `_bar` не помешал нам «залезть» в класс и получить доступ к значению этой переменной.

Все потому, что в Python префикс, состоящий из одинарного подчеркивания, 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 импортируются 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* импортировать имена с начальным символом подчеркивания (если только в модуле не определен список `__all__`, который отменяет такое поведение[1]):

[1] См. документацию Python «Импортирование с подстановочным знаком * из пакета»: https://docs.python.org/3/tutorial/modules.html#importing-from-a-package

```
>>> from my_module import *
>>> external_func()
23
>>> _internal_func() NameError: "name '_internal_func' is not defined"
```

К слову сказать, подстановочного импорта следует избегать, поскольку он вносит неясность в то, какие имена присутствуют в пространстве имен[1]. Ради ясности лучше придерживаться обычного импорта. Согласованные правила именования с одинарным подчеркиванием, в отличие от подстановочного 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 запомнить, состоит в следующем:

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

2. 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 избежать конфликта из-за совпадения имен:

[1] См. PEP 8 «Импортирование»: http://pep8.org/#imports

```
>>> def make_object(name, class):
SyntaxError: "invalid syntax"

>>> def make_object(name, class_):
...     pass
```

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

3. Двойной начальный символ подчеркивания: __var

Шаблоны именования, которые мы рассмотрели к этому моменту, получают 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 подклассах 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 небольшой пример кода, который мы сможем использовать для экспериментирования:

```
class Test:
     def __init__(self):
        self.foo = 11
        self._bar = 23
        self.__baz = 23
```

Давайте взглянем на атрибуты объекта, использовав встроенную функцию `dir()`:

```
>>> t = Test()
>>> dir(t)
['_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']
```

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 неизмененной 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 программисту.

Однако с атрибутом `self.__baz` все выглядит немного по-другому. Когда вы попытаетесь отыскать в списке атрибут `__baz`, вы увидите, что переменной 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 пытается переопределить его существующие атрибуты, добавленные в конструкторе:

```
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 экземплярах класса `ExtendedTest`? Давайте посмотрим:

```
>>> t2 = ExtendedTest()
>>> t2.foo
'переопределено'
>>> t2._bar
'переопределено'
>>> t2.__baz
AttributeError:
"'ExtendedTest' object has no 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 предотвратить 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 программиста 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
AttributeError:
"'ManglingTest' object has no attribute '__mangled'"
```

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()
AttributeError:
"'MangledMethod' object has no attribute '__method'"
>>> 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 переменной. 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 примерами из этого раздела.

Пусть эти принципы впитаются как следует, чтобы вы наконец осознали общую идею искажения имен и некоторые другие формы поведения, которые 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 присутствовали 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). Причина 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». Аналогичным 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 правил именования. Для разработчиков Python это все равно что секретное рукопожатие.

4. Двойной начальный и замыкающий символ подчеркивания: __var__

Пожалуй, это удивляет, но искажение имен не применяется, если имя *начинается* и *заканчивается* двойными символами подчеркивания. Интерпретатор 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 «магического» или тайного.

Тем не менее в контексте согласованных правил именования лучше воздержаться 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 иногда используется в качестве имени, чтобы подчеркнуть, что эта переменная временная или незначительная.

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

```
>>> for _ in range(32):
...     print('Привет, Мир.')
```

Одинарные символы подчеркивания также можно применять в распаковке 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. Одинарный символ подчеркивания — это просто имя допустимой переменной, которое иногда используется с этой целью.

В следующем ниже примере исходного кода я распаковываю кортеж в отдельные 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 взаимодействовать 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 интерпретатором 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. Обеспечивается 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.

2.5. 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, возможно, 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 также 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 константы), 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 0xbadc0ffee!'
```

Надо сказать, что *такая* ошибка и впрямь испортит разработчику утро понедельника! Но сегодня мы здесь собрались, чтобы обсудить форматирование 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 представляет 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[1].

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

Ниже я использую спецификатор формата `%x`, чтобы преобразовать целочисленное значение в строковое и представить его как шестнадцатеричное число:

```
>>> '%x' % errno
'badc0ffee'
```

[1] См. документацию Python «Форматирование строк в стиле printf»: https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

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 0xbadc0ffee!'
```

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%(errno)x!' % {
...     "name": name, "errno": errno }
'Эй, Боб! Вот ошибка 0xbadc0ffee!'
```

Это облегчает поддержку ваших строк и их модификацию в будущем. Вам не нужно волноваться о том, что порядок, в котором вы передаете значения, 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 «современное» 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 поддерживается.

№ 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 оператора `%` и делает синтаксис форматирования строк более упорядоченным.

Форматирование теперь обрабатывается вызовом функции `format()` со строковым объектом[1].

Функция `format()` может применяться для выполнения простого позиционного 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 использовать их в любом порядке, в котором вы захотите. Это довольно мощное функциональное средство языка, поскольку оно позволяет изменять порядок следования отображаемых элементов, не изменяя аргументы, переданные в функцию форматирования:

```
>>> 'Эй, {name}! Вот ошибка 0x{errno:x}!'.format(
...     name=name, errno=errno)
'Эй, Боб! Вот ошибка 0xbadc0ffee!'
```

Этот пример также показывает, как изменился синтаксис форматирования целочисленной переменной в виде шестнадцатеричной строки. Теперь мы должны передавать *спецификацию формата* (format spec) путем добавления суффикса «`:x`» 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[2].

В Python 3 «современному» форматированию строк отдается предпочтение по сравнению с форматированием с использованием `%`. Однако, начиная с Python 3.6, появился еще более оптимальный способ форматирования строковых значений. И об этом способе я вам расскажу в следующем разделе.

---

[1] См. документацию Python «str.format()»: https://docs.python.org/3/library/string.html#format-string-syntax

[2] См. документацию Python «Синтаксис форматной строки»: https://docs.python.org/3/library/string.html#format-string-syntax

№ 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 использовать выражения Python, которые встраиваются в строковые константы. Ниже дан простой пример, который поможет вам проникнуться этим функциональным средством языка:

```
>>> f'Привет, {name}!'
'Привет, Боб!'
```

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

```
>>> a = 5
>>> b = 10
>>> f'Пять плюс десять равняется {a + b}, а не {2 * (a + 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 содержит `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 происходит за кадром, то мы увидим, что в этой функции f-строка преобразовывается 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 качестве оптимизации используется код операции `BUILD_STRING`[1]. Но функционально они одинаковы:

```
>>> import dis
>>> dis.dis(greet)
   2     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 самые задачи форматирования, которые мы обсудили в предыдущих двух разделах:

```
>>> f"Эй, {name}! Вот ошибка {errno:#x}!"
"Эй, Боб! Вот ошибка 0xbadc0ffee!"
```

Новые форматированные строковые литералы Python аналогичны шаблонным 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 форматированных строковых литералах вы можете узнать из официальной документации Python[2].

---

[1] См. Python 3. Спорный вопрос в трекере ошибок № 27078: https://bugs.python.org/issue27078

[2] См. документацию Python «Форматированные строковые литералы»: https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals

№ 4. Шаблонные строки

Еще один прием форматирования строк в Python представлен шаблонными строками. Этот механизм более простой и менее мощный, но в некоторых 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 0xbadc0ffee!'
```

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

Более сложные мини-языки форматирования для других приемов форматирования строк могут вносить уязвимости в ваши программы с точки зрения безопасности. Например, форматные строки могут получать доступ 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()
>>> user_input = '{error.__init__.__globals__[SECRET]}'

# 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 форматирования своих строковых значений в Python, вы можете испытывать замешательство. 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 ситуации, 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 используйте 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» форматирование 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 вариант применения будет влиять на то, какой метод вам следует использовать.
- Если вы затрудняетесь в выборе метода форматирования строк, то попробуйте применить мое *эмпирическое правило форматирования строк*.

2.6. Пасхалка «Дзен Python»

Я знаю, что далее приводится привычная картина, если говорить о книгах 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[1].

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!

[1] Язык Python создал нидерландский программист Гвидо ван Россум.

3 Эффективные функции

3.1. Функции Python — это объекты первого класса

Функции Python относятся к объектам первого класса. Их можно присваивать 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 упростит 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 примеров, 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 интерпретатора Python по мере продвижения.

Чтобы осмыслить понятия, которые мы будем здесь рассматривать, потребуется 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 прекрасный момент, когда вы будете готовы, вас внезапно осенит, и все встанет на свои места.

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

```
def yell(text):
     return text.upper() + '!'

>>> yell('привет')
'ПРИВЕТ!'
```

Функции — это объекты

Все данные в программе Python представляются объектами или связями между объектами[1]. 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 присвоить 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`:

```
>>> bark('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 лежащую в основе функцию, вы все так же можете через него вызвать эту функцию:

[1] См. документацию Python «Объекты, значения и типы»: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types

```
>>> del yell

 >>> yell('Привет?')
NameError: "name 'yell' is not defined"

>>> bark('эй')
'ЭЙ!'
```

Кстати, Python прикрепляет к каждой функции строковый идентификатор. Это делается для отладочных целей во время создания функции. К этому внутреннему идентификатору можно получить доступ посредством атрибута `__name__`[1]:

```
>>> bark.__name__
'yell'
```

Нужно сказать, что хотя атрибут `__name__` функции по-прежнему «yell», это не влияет на то, каким образом вы получаете доступ к объекту-функции 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>]
```

[1] Начиная с Python 3.3, также имеется атрибут `__qualname__`, который служит для такой же цели и обеспечивает строку с квалифицированным именем для устранения неоднозначности между именами функций и классов (см. PEP 3155: https://www.python.org/dev/peps/pep-3155/).

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

```
>>> for f in funcs:
 ...     print(f, f('всем 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 необходимости сначала присваивать его переменной. Для этого можно выполнить поиск и затем немедленно назвать результирующий «бестелесный» объект-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` передать функцию `bark`:

```
>>> greet(bark)
'ПРИВЕТ! Я — ПРОГРАММА PYTHON'
```

Разумеется, можно также определить новую функцию, чтобы сгенерировать приветствие с другим колоритом. Например, следующая функция `whisper` может сработать лучше, если вы не хотите, чтобы ваши программы 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 программах 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 функции, 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 элементом итерируемого объекта, выдавая результат по мере прохождения итерируемого объекта.

Ниже показано, как вы могли бы отформатировать всю последовательность 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 строковыми значениями.

Функции могут быть вложенными

Быть может, вы удивитесь, но Python допускает определение функций внутри других функций. Такие функции нередко называются *вложенными 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 последовательный.

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

```
>>> whisper('Йоу')
NameError:
"name 'whisper' is not defined"

>>> speak.whisper
AttributeError:
"'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 источнику 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 затем 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 *принимать линии поведения* через аргументы, но и *возвращать линии поведения*. Здорово, правда?

И знаете что? С этого места дела приобретают несколько безумный характер. 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 функции и что даже существует возможность возвращать эти (в других ситуациях скрытые) внутренние функции из родительской функции.

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

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

Чтобы это проиллюстрировать, я собираюсь немного переписать предыдущий 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`. Обратили 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 *возвращать 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 конфигурирования 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 *вызываемыми*, что во многих случаях позволяет рассматривать их в качестве функций.

Если объект является вызываемым, то это означает, что вы можете использовать с ним синтаксис вызова функций с круглыми скобками и даже передавать в него аргументы вызова функции. Все это приводится в действие дандер-методом `__call__`. Ниже приведен пример класса, определяющего 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 вызываемым или нет:

```
>>> callable(plus_3)
True
>>> callable(yell)
True
>>> callable('привет')
False
```

Ключевые выводы

- В Python абсолютно все является объектом, включая функции. Их можно присваивать переменным, хранить в структурах данных и передавать или возвращать в другие функции и возвращать из них (функции 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 позволяет рассматривать их в качестве функций.

3.2. 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 словом `def`. Они могут использоваться всякий раз, когда требуются объекты-функции.

Например, ниже показано определение простой лямбда-функции, выполняющей 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 «*функциональное 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 объявлению функции при помощи ключевого слова `def`, только записывается в одну строку. Основное отличие здесь в том, что перед его использованием 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 обычную функцию.

Прежде чем вы продолжите, возможно, вам стоит немного поэкспериментировать 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 потратите 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-единственным 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 выражения 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-функции? 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 применения лямбд состоит в написании кратких и сжатых функций для сортировки 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 бóльшую 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 контекста, 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, объявленной при помощи ключевого слова `def`, может яснее выражать намерение 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 исследованием 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 использованием лямбд, которые выглядели «круто», но на самом деле оказались помехой 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: print('Бум!')
...     crash = lambda self: print('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 удобочитаемый 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 (именованной) функции либо конструкции включения в список бóльшую ясность?

3.3. Сила декораторов

В своей основе декораторы Python позволяют расширять и изменять поведение вызываемых объектов (функций, методов и классов) *без* необратимой 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 концов, только что упомянутое мною выше выглядит довольно абстрактным, и, вполне возможно, вообще сложно увидеть, какую пользу принесут декораторы в повседневной работе разработчика. Позвольте мне несколько прояснить этот вопрос, предоставив вам вполне реальный пример.

Предположим, в вашей программе составления отчетности есть 30 функций 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. Компании 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 относительном спокойствии.

Без декораторов следующие три дня вам пришлось бы провести в попытках 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 декоратора `@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 декораторы, 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 стандартной 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 рекомендую 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 дочерняя 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 выглядит 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 воспользоваться синтаксисом Python `@` для декорирования функции:

```
@null_decorator
def greet():
    return 'Привет!'

>>> greet()
'Привет!'
```

Размещение строки `@null_decorator` перед определением функции аналогично тому, что функция сначала определяется и затем уже прогоняется через декоратор. Синтаксис `@` является всего лишь *синтаксическим сахаром* (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 функцию (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 функцию (замыкание), которая затем может быть вызвана в дальнейшем, выполняет 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 изменяется необратимо — его поведение меняется, только когда он декорирован.

Это позволяет прикреплять к существующим функциям и классам конструктивные блоки многократного использования, в частности функционал 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 абсолютно 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 получается 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 многочисленные 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 приоритет имеет тег `@strong`? Когда вы вызываете декорированную функцию, происходит вот что:

```
>>> greet()
'<strong><em>Привет!</em></strong>'
```

Этот результат ясно показывает, в каком порядке декораторы были применены: *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 поверх старых, поднимаясь все выше и выше.

Если разложить приведенный выше пример и избавиться от синтаксиса @, который применяют декораторы, то цепочка вызовов функций-декораторов выглядит так:

```
decorated_greet = strong(emphasis(greet))
```

И снова вы видите, что сначала применяется декоратор `emphasis` и затем результирующая обернутая функция снова обертывается декоратором `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* функцию `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 принимает 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[1]. Ниже приведен декоратор `proxy`, в котором задействуется их преимущество:

```
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 оригинальную 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.

[1] См. раздел 3.4 «Веселье с `*args` и `**kwargs`».

Давайте расширим прием, сформулированный декоратором `proxy`, в более полезный практический пример. Ниже приведен декоратор `trace`, который регистрирует аргументы функции и итоговые результаты, полученные во время исполнения:

```
def trace(func):
     def wrapper(*args, **kwargs):
         print(f'ТРАССИРОВКА: вызвана {func.__name__}() '
               f'с {args}, {kwargs}')

        original_result = func(*args, **kwargs)

        print(f'ТРАССИРОВКА: {func.__name__}() '
               f'вернула {original_result!r}')
        return original_result
     return wrapper
```

При декорировании функции с использованием декоратора `trace` и последующем ее вызове, будут выведены переданные в декорированную функцию 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 (недекорированной) 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 трудоемкой. К счастью, существует быстрое решение этой проблемы: декоратор `functools.wraps`, включенный в стандартную библиотеку Python[1].

Декоратор `functools.wraps` можно использовать в своих собственных декораторах для того, чтобы копировать потерянные метаданные из недекорированной 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, возвращаемому декоратором, переносит в него строку документации и другие метаданные входной функции:

[1] См. документацию Python «functools»: https://docs.python.org/3/library/functools.html

```
@uppercase def greet():
      """Вернуть дружеское приветствие."""
      return 'Привет!'

>>> greet.__name__
'greet'
>>> greet.__doc__
'Вернуть дружеское приветствие.'
```

В качестве оптимального практического приема я порекомендовал бы использовать 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 сложного раздела и многое узнали о декораторах в Python. Отличная работа!

Ключевые выводы

- Декораторы определяют структурные блоки многократного использования, которые можно применять к вызываемому объекту с целью модификации его поведения без необратимого изменения самого вызываемого объекта.
- Синтаксис `@` является всего-навсего сокращенной записью для вызова декоратора с входной функцией. Многочисленные декораторы, размещенные над одной-единственной функцией, применяются снизу-вверх (*стековая укладка декораторов*).
- В качестве оптимального практического приема отладки используйте 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 инструментов разработки программного обеспечения, декораторы не являются панацеей, и ими не стоит злоупотреблять. Важно уравновесить необходимость «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 определение 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 позволяют 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 принимать 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 аргументами:

```
>>> 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, key1='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 возможность время от времени рычать «арррг!» и «кварррррг!».)

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 которую вы хотите переадресовать аргументы[1].

[1] См. раздел 3.5 «Распаковка аргументов функции».

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):
     def __init__(self, *args, **kwargs):
         super().__init__(*args, **kwargs)
         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 означает, 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 собственными 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 осторожность (иначе вскоре у вас, возможно, появится еще одна причина воскликнуть «аррррг!»).

Еще один сценарий, где этот прием будет потенциально полезен, — написание оберточных функций, таких как декораторы. Там вы также захотите 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 программный код достаточно четким и при этом остаться в рамках принципа DRY[1]. 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.

[1] DRY (от *англ.* Don't Repeat Yourself, то есть «не повторяйся») — это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода. См. https://ru.wikipedia.org/wiki/Don't_repeat_yourself. — *Примеч. пер.*

- Переменная `*args` собирает дополнительные позиционные аргументы в кортеж. Переменная `**kwargs` собирает дополнительные именованные 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 средством 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 функцию 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. Например, 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 излишне 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 использованием 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 аргументы в вызванную функцию.

Этот прием работает для любого итерируемого объекта, включая выражения-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 имеется оператор ** для распаковки именованных аргументов, поступающих из словарей. Предположим, что наш вектор был представлен в виде следующего объекта `dict`:

```
>>> dict_vec = {'y': 0, 'z': 1, 'x': 1}
```

Этот объект-словарь можно передать в функцию `print_vector` практически 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 звездочки (*), 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 аргументов функции, предоставляет вам большую гибкость за бесплатно. Зачастую это означает, что вам не придется реализовывать класс для необходимого вашей программе типа данных. При этом вполне достаточно обойтись простыми встроенными структурами данных, подобными кортежам 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 способствовать написанию более гибких интерфейсов для ваших модулей и функций.

3.6. Здесь нечего возвращать

В конец любой функции Python добавляет неявную инструкцию `return None`. По этой причине, если в функции не указано возвращаемое значение, 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 foo1(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(foo1(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 называется *процедурой*), то я исключаю инструкцию `return`. Добавлять эту инструкцию 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 чисел 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 инструкции `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 удобного в сопровождении программного кода, такое непредсказуемое поведение 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» программный 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 стилистически.
- Это ключевое функциональное средство языка Python, однако ваш программный код может передавать свое намерение четче при помощи явной инструкции `return None`.

4 Классы и ООП

4.1. Сравнения объектов: is против ==

Когда я был мальчишкой, у наших соседей жили кошки-близняшки. Внешне они были очень похожи — одинаковая темно-серая шерсть и одинаковый пронизывающий взгляд зеленых глаз. Отбросив некоторые индивидуальные 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 `a`, 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 оператора `==`, 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 действительности 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 оператором `is`. Это сравнение подтверждает, что обе переменные в действительности 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 существующим списком в качестве аргумента, чтобы создать копию, которую мы назовем `c`:

```
>>> c = list(a)
```

И снова вы увидите, что только что созданный нами новый список выглядит идентичным объекту-списку, на который указывают `a` и `b`:

```
>>> c
[1, 2, 3]
```

А вот теперь начинается самое интересное. Давайте сравним нашу копию списка `c` с первоначальным списком `a`, использовав для этого оператор `==`. Какой ответ вы ожидаете увидеть?

```
>>> a == c
True
```

О'кей. Надеюсь, что вы как раз этого и ожидали. Данный результат говорит следующее: `c` и `a` 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`:

```
>>> a is c
False
```

О-па! И вот тут мы получаем другой результат. Python говорит, что `c` и `a` 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 переменные, равны (имеют одинаковое содержимое).

Всякий раз, когда вам придется решать, применять оператор `is` или оператор `==`, просто вспомните про кошек-близняшек (в принципе, сойдут и собаки). Если вы это будете делать, то у вас все будет в порядке.

4.2. Преобразование строк (каждому классу по __repr__)

Когда вы определяете собственный класс в Python и затем пытаетесь напечатать один из его экземпляров в консоли (или проверить его в сеансе интерпретатора), вы получаете относительно неудовлетворительный

результат. Принятое по умолчанию поведение с преобразованием в строковое 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 = Car('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 является адресом объекта в оперативной памяти). Это лучше, чем *ничего*, но не очень-то полезно.

Вы можете попытаться найти обходной путь, непосредственно распечатав атрибуты класса или даже добавив в классы собственный метод `to_string()`:

```
>>> 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-методы `__str__` и `__repr__`. Они представляют собой питоновский способ управления тем, как объекты преобразовываются в строковые значения в различных ситуациях[1].

[1] См. документацию Python «Модель данных Python»: 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* объекта 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 отдельной функции `to_string()`. Это питоновский способ управлять преобразованием строк.

Между прочим, некоторые разработчики предпочитают называть дандер-методы Python «магическими». Но эти методы никоим образом

магическими не являются. То, что имена этих методов начинаются и оканчиваются 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 заканчивается. Вы заметили, что осмотр объекта `my_car` в сеансе интерпретатора по-прежнему дает этот странный результат `<Car object at 0x109ca24e0>`?

Это произошло, потому что фактически имеется *два* дандер-метода, которые управляют тем, как объекты преобразовываются в строковые значения 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-метода 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 = Car('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 объекта в сеансе интерпретатора Python просто печатается результат выполнения 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 контейнером:

```
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 эксперимент. Мы создадим объект `datetime.date` и выясним, каким образом в нем используются методы `__repr__` и `__str__` для управления преобразованием строк:

```
>>> import datetime
>>> today = datetime.date.today()
```

Результат метода `__str__` объекта даты должен быть прежде всего *удобочитаемым*. 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 объектом 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(2017, 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 применение такой функции на практике. Она не будет стоить затраченных на нее усилий и просто создаст для вас дополнительную работу. Мое эмпирическое 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 `!r`, 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 повторили 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 реализацию метода `__repr__`, когда имя класса изменится. Это позволяет беспрепятственно придерживаться принципа DRY, то есть *«не повторяйся»*:

```
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 форматирования вы сможете сохранить свой исходный код аккуратным и соответствующим правилам PEP 8.

Во время инспектирования объекта или непосредственного вызова функции `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 реализация `__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 реализацией 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 представления текста: `str`. Он содержит символы Юникода и может представлять большинство систем письменности в мире.

В Python 2.*x* для строковых данных используется другая модель данных[1]. 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-метод 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__`. Встроенная 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 правила преобразования текста. Но есть способ все снова упростить в практическом 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

[1] См. документацию Python 2 «Модель данных»: https://docs.python.org/2/reference/datamodel.html

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 просто можете копипастить повсюду, где это необходимо (либо разместить ее в базовом классе, где это имеет смысл). Тогда весь ваш код преобразования 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 метода `__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 `__unicode__`.

4.3. 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 отладочную 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. Игрушечный пример функции валидации имени может выглядеть следующим образом:

```
def validate(name):
     if len(name) < 10:
         raise ValueError
```

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 «высокого 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 ValueError
ValueError
```

Этот отчет не очень-то и полезен. Несомненно, мы знаем, что что-то пошло не так и что проблема имела отношение к «неправильному значению» 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 встроенного в Python класса `ValueError`, но заставим его говорить за себя, дав ему более конкретное имя:

```
class NameTooShortError(ValueError):
     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 переменную `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 обратной 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 собственного 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 базового класса, от которого наследуют все конкретные ошибки:

```
class BaseValidationError(ValueError):
    pass
```

Далее, все наши «реальные» классы ошибок могут быть сделаны производными от базового класса ошибок. В результате мы получаем хорошую и чистую иерархию исключений, приложив лишь незначительные дополнительные усилия:

```
class NameTooShortError(BaseValidationError):
    pass
```

```
class NameTooLongError(BaseValidationError):
    pass

class NameTooCuteError(BaseValidationError):
    pass
```

Например, это позволяет пользователям вашего пакета писать инструкции `try-except`, которые могут обработать все ошибки, возникающие в результате 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 придется прибегать к захватыванию всех исключений при помощи всеобъемлющей 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 исключения 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 программирования «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 встроенного в Python класса `Exception` или от более конкретных классов-исключений, таких как `ValueError` или `KeyError`.

- Для определения логически сгруппированных иерархий исключений можно использовать наследование.

4.4. 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 модифицировать *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 коллекций. Встроенные в Python изменяемые коллекции, такие как списки, словари и множества, могут быть скопированы путем вызова своих фабричных 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, вдобавок ко всему, он создает только *мелкие копии*. Для составных объектов, таких как списки, словари и множества, между *мелким* и *глубоким* копированием имеется важное различие.

Мелкая копия (shallow copy) означает конструирование нового объекта-коллекции и затем его заполнение ссылками на дочерние объекты, найденные 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 затем рекурсивное его заполнение копиями дочерних объектов, найденных в оригинале. При копировании объекта таким способом выполняется обход 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, проинспектировав 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 оригинала, 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 списка, 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 списке `xs`, эта модификация также будет отражена в списке `ys` — таким образом, *оба списка совместно используют одинаковые дочерние объекты*. 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 оказывается, 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 объекта 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 копированием. 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 мелких и глубоких копий произвольных объектов Python.

Создание глубоких копий

Давайте повторим предыдущий пример с копированием списка, но с одним 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 оригинальном объекте (`xs`), то вы увидите, что эта модификация не затронет глубокую копию (`zs`).

Оба объекта, оригинал и копия, на этот раз полностью независимы. Список `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 поэкспериментировать 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 программном 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, использовав 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 своих координат использует примитивные типы (целые числа), то в данном случае нет никакой разницы между мелкой и глубокой копией. Но я расширю 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 применить 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, возможно, 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 времяпрепровождения!

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 клонируются рекурсивно. Клон полностью независим от оригинала, но на создание глубокой копии уходит больше времени.
- При помощи модуля `copy` вы можете копировать произвольные объекты (включая собственные классы).

4.5. Абстрактные базовые классы держат наследование под контролем

Абстрактные классы (АК), иногда также называемые *абстрактными базовыми 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 работе был спор о том, какой шаблон использовать для реализации удобной в сопровождении иерархии классов в Python. Точнее говоря, цель состояла в том, чтобы определить простую иерархию классов для сервисного бэкенда самым благоприятным для программиста и удобным в сопровождении способом.

---

[1] См. документацию Python «Операции мелкого и глубокого копирования»: https://docs.python.org/3/library/copy.html

У нас был класс `BaseService`, который определял общий интерфейс и несколько 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 конкретные реализации были производными от класса `BaseService`.

Чтобы сделать этот программный код максимально удобным в обслуживании и благоприятным для программиста, мы хотели удостовериться, что

- создание экземпляров базового класса невозможно,
- упущение из виду реализации методов интерфейса в одном из подклассов вызывает ошибку на ранней стадии.

Итак, почему же может возникнуть потребность в использовании модуля 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 = Concrete()
>>> c.foo()
'вызвана foo()'
>>> c.bar()
NotImplementedError
```

Эта первая реализация выглядит неплохо, но пока не идеально. Ее оборотными 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[1], мы можем добиться большего успеха и решить эти оставшиеся проблемы. Вот обновленная реализация с использованием абстрактного класса, определенного 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()...
```

[1] См. документацию Python «Модуль abc»: https://docs.python.org/3/library/abc.html

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. Подклассы `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 экземпляра является большим преимуществом. В результате написание недопустимых 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. Использование абстрактных классов позволяет программисту четче формулировать свой замысел и таким образом делает код более коммуникативным. Я рекомендую 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 классов более легкими в сопровождении.

4.6. Чем полезны именованные кортежи

Python поставляется со специализированным контейнерным типом `namedtuple`, то есть именованным кортежем. И этот тип, по всей видимости, 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 исключительность? 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 свойствам, 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 возможность 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, *именованными кортежами* (named tuples). Доступ к каждому хранящемуся в них объекту можно получить через уникальный (человекочитаемый) идентификатор. Это свойство освобождает вас от необходимости запоминать 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('Авто' , 'цвет пробег')
```

Именованные кортежи были добавлены в стандартную библиотеку Python версии 2.6. Чтобы ими воспользоваться, необходимо импортировать 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 фабричной 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 реализации 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 действительности это просто сокращенная запись для приведенных ниже двух шагов:

```
>>> 'цвет пробег'.split()
['цвет', 'пробег']
>>> Car = namedtuple('Авто', ['цвет', 'пробег'])
```

Разумеется, вы также можете непосредственно передать список со строковыми 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('Авто', [
...     '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.цвет
'красный'
>>> my_car.пробег
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 использоваться 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 набирать чуть меньше текста и сэкономит на многословности:

```
>>> my_car
Авто(цвет='красный' , пробег=3812.4)
```

Подобно кортежам, именованные кортежи не изменяются. Когда вы попытаетесь переписать одно из их полей, вы получите исключение `AttributeError`:

```
>>> my_car.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('Авто', 'цвет пробег')
class MyCarWithMethods(Car):
     def hexcolor(self):
         if self.цвет == 'красный':
             return '#ff0000'
         else:
             return '#000000'
```

Теперь можно создавать объекты `MyCarWithMethods` и, следовательно, вызывать 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 — использовать 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 также 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 публичного интерфейса класса `namedtuple`. Вспомогательные методы получают такие имена, чтобы избежать конфликтов имен с определяемыми пользователями полями кортежей. Так что можете спокойно ими пользоваться, если они вам нужны!

Хочу показать вам несколько сценариев, где могли бы пригодиться вспомогательные методы именованного кортежа. Давайте начнем со вспомогательного метода `_asdict()`. Он возвращает содержимое именованного кортежа в виде словаря:

```
>>> my_car._asdict()
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._asdict(), 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:

```
>>> Car._make(['красный', 999])
Авто(color='красный', пробег=999)
```

Когда использовать именованные кортежи

Именованные кортежи могут оказаться простым средством для приведения в порядок исходного кода, и они могут сделать код более удобочитаемым, обеспечив вашим данным наиболее совершенную структуру.

Например, на моем опыте переход от ситуативных типов данных, таких как словари с фиксированным форматом, к именованным кортежам помогает 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 «самодокументированном» 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 более 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 определения неизменяющегося класса вручную.
- Именованные кортежи помогут почистить ваш исходный код, обеспечив вашим данным более доступную для понимания структуру.

- Именованные кортежи обеспечивают несколько полезных вспомогательных методов, которые начинаются с одинарного символа подчеркивания, но при этом являются составной частью публичного интерфейса. Вполне нормально их использовать.

4.7. Переменные класса против переменных экземпляра: подводные камни

Помимо проведения различия между методами класса и методами экземпляра, объектная модель Python также проводит различие между *переменными* класса и *переменными* экземпляра.

Это различие имеет большое значение. Мне, как начинающему разработчику на Python, оно также доставляло немало хлопот. В течение длительного времени я не мог найти время, чтобы разобраться в этих понятиях 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 помощи 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 одному конкретному экземпляру класса. Вместо этого переменные класса хранят свое содержимое в самом классе, и все объекты, созданные на основе того или иного класса, предоставляют общий доступ к одинаковому набору переменных класса. Например, это означает, что модификация 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 переменной экземпляра абсолютно независимо от одного экземпляра объекта

к другому. И поэтому модификация переменной экземпляра одновременно затрагивает только один экземпляр объекта.

Ладно, все это было довольно абстрактно — самое время рассмотреть немного исходного кода! Давайте потренируемся на собачках... В обучающих 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 экземпляров `Dog` работает, как и ожидалось, и каждый из них получает переменную 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 гибкости. Доступ к переменной класса `num_legs` можно получить либо непосредственно 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 экземпляра характерны для каждого экземпляра объекта и создаются, когда выполняется 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 небольшой «песочнице» с исходным кодом, которая у нас сейчас есть?

Первая идея — просто модифицировать переменную `num_legs` в классе `Dog`:

```
>>> 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 микровселенной 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 экземпляры 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 изменение 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 обращаемся к области действия экземпляра:

```
>>> jack.num_legs, jack.__class__.num_legs
(6, 4)
```

Как вы видите, переменные класса, казалось бы, стали *несогласованными*. Это произошло потому, что внесение изменения в `jack.num_legs` создало *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 которые 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 пострадала (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` отслеживает, сколько раз он использовался для создания экземпляров на протяжении 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 инициализирует счетчик нулем, а затем оставляет его в покое.

Всякий раз, когда вы создаете новый экземпляр этого класса, он увеличивает общий счетчик на единицу во время выполнения конструктора `__init__`:

```
>>> CountedObject.num_instances
0
>>> CountedObject().num_instances
1
>>> CountedObject().num_instances
2
>>> CountedObject().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 конструктор 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) реализация 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 разделяемым 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 (несмотря на их подводные камни) могут оказаться полезными инструментами на практике. Удачи!

Ключевые выводы

- Переменные класса предназначены для данных, совместно используемых всеми экземплярами класса. Они принадлежат именно классу, а не

конкретному экземпляру и являются общими для всех экземпляров класса.

- Переменные экземпляра предназначены для данных, которые уникальны 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 (непреднамеренно) 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 сообщает свой замысел и в конечном счете будет удобнее в сопровождении.

Давайте начнем с написания класса (Python 3), который содержит простые 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 объявление `class MyClass`, вы можете объявить класс в новом стиле, с наследованием 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 бóльшую часть времени. Вы видите, что этот метод принимает 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 метода.

[1] См. документацию Python «@classmethod»: https://docs.python.org/3/library/functions.html#classmethod

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 экземплярах класса.

Статические методы

Третий метод, `MyClass.staticmethod`, был помечен декоратором `@staticmethod`[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 понимание 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 вызываем. Начнем с создания экземпляра класса, а затем вызовем три определенных в нем разных метода.

Класс `MyClass` был создан так, чтобы реализация каждого метода возвращала кортеж, содержащий информацию, которую мы можем использовать, чтобы проследить, что происходит и к каким частям класса или объекта этот метод может получить доступ.

---

[1] См. документацию Python «@staticmethod»: https://docs.python.org/3/library/functions.html#staticmethod

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 проигнорировать синтаксический сахар, предоставляемый синтаксической конструкцией вызова с точкой, `obj.method()`, и передать объект экземпляра вручную:

```
>>> MyClass.method(obj)
('вызван метод экземпляра', <MyClass instance at 0x11a2>)
```

Между прочим, методы экземпляра могут также получать доступ непосредственно 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 самом 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>`, представляющему сам класс (в Python все является объектом, даже сами классы).

Обратите внимание на то, как Python автоматически передает класс в качестве 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 экземпляра 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 точечного 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 методы 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 экземпляр объекта и попытались вызвать функцию экземпляра непосредственно на самом шаблоне класса. Иными словами, в Python нет способа заполнить аргумент `self`, и поэтому данный вызов терпит неудачу с исключением `TypeError`.

Это должно сделать различие между этими тремя типами методов чуть яснее. Но не переживайте, я не собираюсь останавливаться на этом. В следующих 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 существует 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 пользователям 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*[1] 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` непосредственно.

Вы можете использовать эту идиому, чтобы следовать принципу «Не повторяйся» (DRY). Если в какой-то момент мы решим этот класс переименовать, 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'])
```

[1] См. Википедию: «Фабрика (объектно-ориентированное программирование)»: https://en.wikipedia.org/wiki/Factory_(object-oriented_programming) и https://ru.wikipedia.org/wiki/Абстрактная_фабрика_(шаблон_проектирования)

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 — понять, что они позволяют определять для своих классов альтернативные конструкторы.

Python допускает всего один метод `__init__` в классе. Использование методов 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 продолжу 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 метод `__repr__`, и теперь они принимают дополнительный аргумент `radius`.

Я также добавил метод экземпляра `area()`, который вычисляет и возвращает 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 статическими методами.

Как мы узнали, статические методы не могут получать доступ к состоянию 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 ограничение 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 архитектуры классов для того, чтобы процесс новой разработки естественным образом направлялся в пределах этих границ. Безусловно, эти ограничения достаточно легко нарушить. Но на практике они нередко помогают избежать непреднамеренных модификаций, которые идут вразрез с первоначальным проектом.

Другими словами, использование статических методов и методов класса способствует передаче замысла разработчика, при этом достаточно подкрепляя этот замысел, чтобы избежать большинства ошибок «по недоразумению» 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 предоставить 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 протестировать.

Нам не придется переживать по поводу настройки полного экземпляра класса перед тем, как мы сможем протестировать этот метод в модульном тесте. Мы просто можем действовать подобно тому, как мы действовали бы при тестировании обычной функции. И опять-таки, это облегчает сопровождение кода в будущем и обеспечивает связь между объектно-ориентированным и процедурным стилями программирования.

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 получать 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 работают как обычные функции, но принадлежат пространству имен класса.
- Статические методы и методы класса сообщают и (до известной степени) подкрепляют замысел разработчика в отношении конструкции класса. Это может обладать определенными преимуществами в сопровождении кода.

5 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 обеспечивает 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 сосредоточиваться 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 сформулированного «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, словари, множества... м-м-м. Стеки? Разве у нас есть стеки?

Видите ли, проблема в том, что Python поставляется с обширным набором 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 science» 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 связанный 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 распространенные 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 собеседованиях 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 фундаментальным (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.

5.1. Словари, ассоциативные массивы и хеш-таблицы

В Python словари — центральная структура данных. В словарях хранится произвольное количество объектов, каждый из которых идентифицируется уникальным *ключом* словаря.

Словари также нередко называют *ассоциативными массивами* (associative arrays), *ассоциативными хеш-таблицами* (hashmaps), *поисковыми таблицами* (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* представляют 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 поисках чьего-то номера, можно почти напрямую перескочить к имени и посмотреть связанную с ним информацию.

Эта аналогия несколько рушится, когда дело доходит до того, каким образом информация организована, чтобы допускать выполнение быстрых операций поиска. Но фундаментальные характеристики производительности остаются прежними: словари позволяют быстро находить информацию, 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.

dict — ваш дежурный словарь

Из-за своей важности Python содержит надежную реализацию словаря, которая встроена непосредственно в ядро языка: тип данных `dict`[1].

Для работы со словарями в своих программах Python также предоставляет 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 тип[2]: 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-кортежи — при условии, что они сами содержат только хешируемые типы.

---

[1] См. документацию Python «Ассоциативные типы — dict»: https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

[2] См. глоссарий документации Python «hashable»: https://docs.python.org/3/glossary.html

Для большинства вариантов использования встроенная в Python реализация словаря делает все, что вам нужно. Словари хорошо оптимизированы 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 настроенной реализации хеш-таблицы, которая обеспечивает ожидаемые характеристики производительности с временной сложностью $O(1)$ для операций поиска, вставки, обновления и удаления в среднем случае.

Нет особых причин не использовать стандартную реализацию `dict`, включенную в Python. Тем не менее существуют специализированные сторонние реализации словаря, например списки с пропусками или словари на основе B-деревьев.

Помимо «обыкновенных» объектов `dict`, 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 `dict`, который запоминает порядок вставки добавляемых в него ключей: `collections.OrderedDict`[1].

Хотя в Python 3.6 и выше стандартные экземпляры `dict` сохраняют порядок вставки ключей, такое поведение является всего лишь побочным эффектом реализации в Python и не определяется спецификацией языка[2]. Поэтому, если для работы вашего алгоритма порядок следования ключей

[1] См. документацию Python «collections.OrderedDict»: https://docs.python.org/3/library/collections.html#collections.OrderedDict

[2] См. список рассылки CPython: https://mail.python.org/pipermail/python-dev/2016-September/146327.html

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`, находящегося 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'])
```

collections.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 найти[1].

Это свойство может сэкономить на наборе кода и сделать замысел программиста яснее в сравнении с использованием методов `get()` или отлавливанием 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 list():
>>> dd['собаки'].append('Руфус')
>>> dd['собаки'].append('Кэтрин')
```

[1] См. документацию Python «collections.defaultdict»: https://docs.python.org/3/library/collections.html#defaultdict-objects

```
>>> 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[1]. 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']
KeyError: 'отсутствует'
```

types.MappingProxyType — обертка для создания словарей только для чтения

`MappingProxyType` — это обертка стандартного словаря, которая предоставляет доступ только для чтения данных обернутого словаря[2]. Этот

[1] См. документацию Python «collections.ChainMap»: https://docs.python.org/3/library/collections.html#collections.ChainMap

[2] См. документацию Python «types.MappingProxyType»: https://docs.python.org/3/library/types.html

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 ассоциативный тип использовать в ваших программах, я указал бы на встроенный тип данных `dict`. Он представляет собой универсальную и оптимизированную 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 `dict` будет «вполне приемлем» в большинстве случаев.
- Специализированные реализации, такие как словари с доступом только для чтения или упорядоченные словари, имеются в стандартной библиотеке Python.

5.2. 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 которых 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 использования. Но прежде чем начать, рассмотрим некоторые основы.

Как работают массивы и для чего они применяются?

Массивы состоят из записей данных, при этом записи имеют фиксированный 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 разрешено парковать велосипеды. «Ограниченная» автостоянка соответствует структуре данных для «типизированного массива», которая допускает 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 элемента. Для данного случая надлежащая реализация массива гарантирует постоянное *O(1)* время доступа.

В своей стандартной библиотеке Python содержит несколько массивоподобных структур данных, каждая из которых обладает слегка отличающимися характеристиками. Давайте их рассмотрим.

list — изменяемые динамические массивы

Списки (lists) являются составной частью ядра языка Python[1]. Несмотря на свое имя, списки Python реализованы как *динамические массивы*. Это означает, что список допускает добавление и удаление элементов и авто-

[1] 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

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 означает, 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 Python[1]. Однако в отличие от списков, в Python объекты-кортежи

[1] См. документацию Python «tuple»: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

не изменяются. Это означает, что элементы не могут динамически добавляться или удаляться — все элементы в кортеже должны быть определены во время создания.

Точно так же, как и списки, кортежи могут содержать элементы произвольных 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 хранение 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 различия — они являются «типизированными массивами», ограниченными единственным типом данных[1].

[1] См. документацию Python «array.array»: https://docs.python.org/3/library/array.html

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 замены» 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[1]. В сущности, это означает, что тип `str` представляет собой

[1] См. документацию Python «str»: https://docs.python.org/3.6/library/stdtypes.html#text-sequence-type-str

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, модификация 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
'abcd'

# 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'))
'abcd'

# Строки — это рекурсивные структуры данных:
>>> type('abc')
"<class 'str'>"
>>> type('abc'[0])
"<class 'str'>"
```

bytes — неизменяемые массивы одиночных байтов

Объекты `bytes` представляют собой неизменяемые последовательности одиночных байтов (целых чисел в диапазоне $0 \leq x \leq 255$)[1]. В концептуальном плане они подобны объектам `str` и их также можно представить как неизменяемые массивы байтов.

Аналогично строковому типу, тип `bytes` имеет свой собственный литеральный синтаксис, предназначенный для создания объектов, и объекты этого типа пространственно эффективны. Объекты `bytes` не могут изменяться, 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 о 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))
ValueError: "bytes must be in range(0, 256)"

# Байты неизменяемы:
>>> arr[1] = 23
TypeError:
"'bytes' object does not support item assignment"

>>> del arr[1]
TypeError:
"'bytes' object doesn't support item deletion"
```

[1] См. документацию Python «bytes»: https://docs.python.org/3/library/stdtypes.html#bytes

bytearray — изменяемые массивы одиночных байтов

Тип `bytearray` представляет собой изменяемую последовательность целых чисел в диапазоне $0 \leq x \leq 255$[1]. 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 неизменяемые 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"
```

[1] См. документацию Python «bytearray»: https://docs.python.org/3/library/stdtypes.html#bytearray

```
# (целые числа в диапазоне 0 <= x <= 255)
>>> arr[1] = 'привет'
TypeError: "an integer is required"

>>> arr[1] = 300
ValueError: "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 сосредоточились 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*[1], предлагают широкий спектр массивоподобных реализаций с большим быстродействием для научных вычислений и науки о данных.

Если ограничиваться массивоподобными структурами данных, включенными 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 нужно. Кроме того, рассмотрите выход за пределы стандартной библиотеки и попробуйте такие пакеты, как *NumPy* или *Pandas*[2].

[1] См. www.numpy.org
[2] См. https://pandas.pydata.org/

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 производительность или занимаемое пространство оперативной памяти становятся проблемой. В большинстве случаев использование массивоподобной структуры 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.

5.3. 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 встроенных 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 встроенный в Python тип `tuple`, который может как считаться, так и не считаться записью в строгом смысле этого слова, потому что кортежи не обеспечивают 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 принятия 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[1]. 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 изменяться, 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.

```
car1 = {
     'цвет': 'красный',
     'пробег': 3812.4,
     'автомат': True,
}
car2 = {
     'цвет': 'синий',
     'пробег': 40231,
     'автомат': False,
}

# Словари имеют хороший метод repr:
```

[1] См. раздел «Словари, ассоциативные массивы и хеш-таблицы» настоящей главы.

```
>>> 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:
car3 = {
    'цвет': 'зеленый',
    'автомат': False,
    'лобовое стекло': 'треснутое',
}
```

tuple — неизменяемые группы объектов

Кортежи в Python представляют собой простые структуры данных, предназначенные для группирования произвольных объектов[1]. Кортежи неизменяемы — после их создания их нельзя исправить.

С точки зрения производительности кортежи занимают чуть меньше оперативной памяти, чем списки в Python[2], и к тому же быстрее создаются.

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:

---

[1] См. документацию Python «tuple»: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

[2] 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)

```
>>> 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 кортежу 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 разночтений порядка следования полей. Поэтому я рекомендую держать минимальное количество полей в кортеже.

```
# Поля: цвет, пробег, автомат
>>> car1 = ('красный', 3812.4, True)
>>> car2 = ('синий', 40231.0, False)

# Экземпляры кортежа имеют хороший метод repr:
>>> car1
('красный', 3812.4, True)
>>> car2
('синий', 40231.0, False)
```

```
# Получить пробег:
>>> car2[1]
40231.0

# Кортежи неизменяемы:
>>> 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 объект 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 реализаций. Например, добавление новых полей в конструктор `__init__` будет многословным и займет время.

Кроме того, принятое по умолчанию строковое представление объектов-экземпляров, создаваемых на основе собственных классов, не очень полезно. Чтобы это исправить, вам, вероятно, придется добавить свой собственный метод `__repr__`[1], который, как правило, довольно многословен и подлежит обновлению всякий раз, когда вы добавляете новое поле.

Хранящиеся в классах поля могут изменяться, и новые поля могут добавляться свободно, нравится вам это или нет. С помощью декоратора `@property` можно обеспечить себе большее управление и создавать поля с доступом только для чтения[2], но это требует написания большего количества связующего кода.

---

[1] См. раздел «Преобразование строк (каждому классу по __repr__)» главы 4.

[2] См. документацию Python «property»: https://docs.python.org/3/library/functions.html#property

Написание собственного класса — отличная возможность, когда в объекты-записи требуется добавить бизнес-логику и *поведение* с использованием 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

>>> car1 = Car('красный', 3812.4, True)
>>> car2 = Car('синий', 40231.0, False)

# Получить пробег:
>>> car2.mileage
40231.0

# Классы изменяемы:
>>> car2.mileage = 12
>>> car2.windshield = 'треснутое'

# Строковое представление не очень полезно
# (приходится добавлять написанный вручную метод __repr__):
>>> car1
<Car object at 0x1081e69e8>
```

collections.namedtuple — удобные объекты данных

Класс `namedtuple`, доступный в Python 2.6+, предоставляет расширение встроенного типа данных `tuple`[1]. Аналогично определению собственного класса, применение именованного кортежа `namedtuple` позволяет определять «шаблоны» многократного использования для своих записей, гарантирующие использование правильных имен полей.

Именованные кортежи неизменяемы, как и обычные кортежи. Это означает, что вы не можете добавлять новые поля или изменять существующие поля после того, как экземпляр `namedtuple` был создан.

[1] См. раздел «Чем полезны именованные кортежи» главы 4.

Помимо этого, именованные кортежи являются, скажем так, именованными кортежами (named tuples). Доступ к каждому хранящемуся в них объекту можно получить по уникальному идентификатору. Это освобождает 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 зрения 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 исходный 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 рефакторизацию, 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 «самодокументированном» виде (в известной степени).

```
>>> from collections import namedtuple
>>> Car = namedtuple('Авто' , 'цвет пробег автомат')
```

```
>>> car1 = Car('красный', 3812.4, True)

# Экземпляры имеют хороший метод repr:
>>> car1 Авто(цвет='красный', пробег=3812.4, автомат=True)

# Доступ к полям:
>>> car1.пробег
3812.4

# Поля неизменяемы:
>>> car1.пробег = 12
AttributeError: "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 `collections`[1]. 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 *mypy*[2]. Но даже без инструментальной поддержки они могут предоставлять полезные подсказки 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
```

[1] См. документацию Python «typing.NamedTuple»: https://docs.python.org/3.6/library/typing.html

[2] См. mypy-lang.org

```
>>> car1 = Car('красный', 3812.4, True)
# Экземпляры имеют хороший метод repr:
>>> car1 Car(цвет='красный', пробег=3812.4, автомат=True)
# Доступ к полям:
>>> car1.пробег 3812.4

# Поля неизменяемы:
>>> car1.пробег = 12
AttributeError: "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 туру:
>>> Car('красный', 'НЕВЕЩЕСТВЕННЫЙ', 99)
Car(цвет='красный', пробег='НЕВЕЩЕСТВЕННЫЙ', автомат=99)
```

struct.Struct — сериализованные C-структуры

Класс `struct.Struct`[1] 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 строкоподобного мини-языка, который позволяет определять расположение различных типов данных C, таких как `char`, `int` и `long`, а также их беззнаковых вариантов.

Сериализованные структуры редко используются для представления объектов 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, применяемый только программным кодом Python.

В некоторых случаях упаковка примитивных данных в структуры позволяет уменьшить объем потребляемой оперативной памяти, чем их

[1] См. документацию Python «struct.Struct»: https://docs.python.org/3/library/struct.html#module-struct

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'x17x00x00x00x00x00x00x00x00(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.SimpleNamespace`[1]. 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 ключи 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 содержательный 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',
```

[1] См. документацию Python «types.SimpleNamespace»: https://docs.python.org/3.3/library/types.html

```
...                         пробег=3812.4,
...                         автомат=True)

# Метод repr по умолчанию:
>>> car1
namespace(автомат=True, пробег=3812.4, цвет='красный')

# Экземпляры поддерживают атрибутивный доступ и могут изменяться:
>>> car1.пробег = 12
>>> car1.лобовое_стекло = 'треснутое'
>>> del car1.автомат
>>> car1
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, y, z)` в трехмерном пространстве.

Вам нужны неизменяемые поля: в данном случае обыкновенные кортежи, `collections.namedtuple` и `typing.NamedTuple`, дадут неплохие возможности 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 напоминает `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 написать 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 простой 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.

5.4. 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, множества 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 $O(1)$ время. Операции объединения, пересечения, разности и взятия подмножеств должны в среднем занимать $O(n)$ времени. В реализациях множества, включенных в стандартную библиотеку Python, данные характеристики производительности соблюдаются[1].

[1] См. https://wiki.python.org/moin/TimeComplexity

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 пустое множество, вам нужно вызвать конструктор `set()`. Использование фигурных скобок `{}` неоднозначно и вместо этого создаст пустой словарь.

Python и его стандартная библиотека предоставляют несколько реализаций множества. Давайте их рассмотрим.

set — ваше дежурное множество

Это встроенная в Python реализация множества[1]. Тип `set` изменяемый и допускает динамическую вставку и удаление элементов.

Множества Python `set` подкрепляются типом данных `dict` и обладают одинаковыми характеристиками производительности. Любой хешируемый объект может храниться в множестве `set`[2].

```
>>> vowels = {'а', 'о', 'э', 'и', 'у', 'ы', 'е', 'е', 'ю', 'я'}
>>> 'э' in vowels
True

>>> letters = set('алиса')
>>> letters.intersection(vowels)
{'а', 'и'}

>>> vowels.add('x')
```

[1] См. документацию Python «set»: https://docs.python.org/3/tutorial/datastructures.html#sets
[2] См. документацию Python «hashable»: https://docs.python.org/3/glossary.html

```
>>> 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 сконструировано[1]. Множества `frozenset` статичны и допускают только операции с запросами в отношении своих элементов (никаких вставок или удалений). 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', 'o', 'э', 'и', 'у', 'ы', 'е', 'е', 'ю',
                        'я'}) >>> vowels.add('p')
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 неоднократное появление элемента в множестве[2].

---

[1] См. документацию Python «frozenset»: https://docs.python.org/3/library/stdtypes.html#frozenset

[2] См. документацию Python «collections.Counter»: https://docs.python.org/3/library/collections.html#counter-objects

Это бывает полезно, если вам нужно вести учет не только того, принадлежит 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 соблюдать 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)
3 # 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 библиотеку.
- Используйте встроенный тип `set`, когда вы хотите получить изменяемое множество.
- Объекты `frozenset` хешируемы и могут использоваться в качестве словаря или ключей множества.
- Класс `collections.Counter` реализует структуры данных «мультимножество», или «мешок».

5.5. Стеки (с дисциплиной доступа LIFO)

Стек представляет собой коллекцию объектов, которая поддерживает быструю 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 называются *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 линейными 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 синтаксическом анализе языка и управлении рабочей памятью времени исполнения («стек вызовов»). Короткий и красивый алгоритм с использованием стека представлен поиском в глубину (DFS) на древовидной или графовой структуре данных.

Python поставляется с несколькими реализациями стека, каждая из которых имеет слегка отличающиеся характеристики. Сейчас мы их рассмотрим и сравним их характеристики.

list — простые встроенные стеки

Встроенный в Python тип `list` создает нормальную стековую структуру данных, поскольку он поддерживает операции вталкивания и выталкивания за амортизируемое $O(1)$ время[1].

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 хранящихся 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 получаете амортизируемую временную сложность $O(1)$.

Недостаток же состоит в том, что это делает показатели их производительности менее надежными, чем стабильные вставки и удаления с временной сложностью $O(1)$, которые обеспечиваются реализацией на основе связного списка (такого, как `collections.deque`, см. ниже). С другой стороны, списки реально обеспечивают быстрый (со временем $O(1)$) произвольный доступ к элементам в стеке, и это может быть дополнительным преимуществом.

Используя списки в качестве стеков, необходимо учитывать одно важное предостережение относительно производительности.

Чтобы получить производительность с амортизируемым временем $O(1)$ 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 должны расти по направлению к более высоким индексам и сжиматься к более низким.

Добавление и удаление элементов в начале списка намного медленнее и занимает $O(n)$ времени, поскольку существующие элементы должны сдвигаться, чтобы создать место для нового элемента. Такого антишаблона производительности следует избегать.

---

[1] См. документацию Python «Использование списков в качестве стеков»: https://docs.python.org/3/tutorial/datastructures.html

```
>>> s = []
>>> s.append('есть')
>>> s.append('спать')
>>> s.append('программировать')

>>> s
['есть', 'спать', 'программировать']

>>> s.pop()
'программировать'
>>> s.pop()
'спать'
>>> s.pop()
'есть'

>>> s.pop()
IndexError: "pop from empty list"
```

collections.deque — быстрые и надежные стеки

Класс `deque` реализует очередь с двусторонним доступом, которая поддерживает добавление и удаление элементов с любого конца за $O(1)$ (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[1].

Объекты Python `deque` реализованы как двунаправленные связные списки, что дает им стабильную производительность для операций вставки и удаления элементов, но при этом плохую $O(n)$ производительность для произвольного доступа к элементам в середине очереди[2].

В целом двусторонняя очередь `collections.deque` – отличный выбор, если вы ищете стековую структуру данных в стандартной библиотеке Python, которая обладает характеристиками производительности, аналогичными реализации на основе связного списка.

---

[1] См. документацию Python «collections.deque»: https://docs.python.org/3.6/library/collections.html#collections.deque

[2] См. CPython «_collectionsmodule.c»: https://github.com/python/cpython/blob/master/Modules/_collectionsmodule.c

```
>>> from collections import deque
>>> s = deque()
>>> s.append('есть')
>>> s.append('спать')

>>> s.append('программировать')
>>> s
deque(['есть', 'спать', 'программировать'])

>>> s.pop()
'программировать'

>>> s.pop()
'спать'

>>> s.pop()
'есть'

>>> s.pop()
IndexError: "pop from an empty deque"
```

deque.LifoQueue — семантика блокирования для параллельных вычислений

Данная реализация стека в стандартной библиотеке Python синхронизирована и обеспечивает семантику блокирования с целью поддержки многочисленных параллельных производителей и потребителей[1].

Помимо `LifoQueue`, модуль `queue` содержит несколько других классов, которые реализуют очереди с мультипроизводителями/мультипотребителями, широко используемые в параллельных вычислениях.

В зависимости от вашего варианта использования семантика блокирования может оказаться полезной, а может накладывать ненужные издержки. В этом случае в качестве стека общего назначения лучше всего использовать список `list` или двустороннюю очередь `deque`.

```
>>> from queue import LifoQueue
>>> s = LifoQueue()
>>> s.put('есть')
```

[1] См. документацию Python «queue.LifoQueue»: https://docs.python.org/3.6/library/queue.html

```
>>> s.put('спать')
>>> 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 производительности 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 использования:

- 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 требовала изменения размеров, и для этих операций вы получаете амор-

тизируемую временную сложность $O(1)$. Однако вам следует быть внимательными и стараться выполнять вставку и удаление элементов «с правильной стороны», используя методы `append()` и `pop()`. В противном случае производительность замедлится до $O(n)$.

- Двусторонняя очередь `collections.deque` поддерживается двунаправленным связным списком, который оптимизирует добавления и удаления с обоих концов и обеспечивает для этих операций стабильную производительность $O(1)$. Производительность класса `deque` не только стабильнее, но его также легче использовать, потому что вам не приходится 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 *LIFO*) на Python.

Ключевые выводы

- Python поставляется с несколькими реализациями стека, которые обладают слегка различающимися характеристиками производительности 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 производительности.

5.6. 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 получения 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* состоит 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 металлической трубе), вы не можете до них добраться. Единственный способ 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*); однако 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 $O(1)$ времени на операции вставки и удаления. Эти две выполняемые с очередью операции являются глав-

ными, и при правильной реализации они обеспечивают высокое быстродействие.

Очереди находят широкое применение в алгоритмах и нередко помогают решать задачи планирования и параллельного программирования. Короткий и красивый алгоритм с использованием очереди представлен поиском в ширину (breadth-first search, BFS) на древовидной или графовой структуре данных.

В алгоритмах планирования выполнения задач во внутреннем представлении нередко используются очереди с приоритетом. Они представляют собой специализированные очереди: вместо получения следующего элемента по времени вставки очередь с приоритетом получает элемент *с самым высоким приоритетом*. Приоритет отдельных элементов определяется очередью, основанной на примененном к их ключам упорядочении. В следующем разделе мы обратимся к очередям с приоритетом и рассмотрим ближе, как они реализуются в Python.

Однако в обычной очереди содержащиеся в ней элементы не переупорядочиваются. 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 которых обладает несколько различающимися характеристиками. Давайте их рассмотрим.

list — ужасно мееееедленная очередь

В качестве очереди можно использовать обычный список, но с точки зрения производительности такое решение не идеально[1]. 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 импровизированной очереди в Python (если только вы не имеете дело с небольшим количеством элементов).

[1] См. документацию Python «Применение списков в качестве очередей»: https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-queues

```
>>> q = []
>>> q.append('есть')
>>> q.append('спать')
>>> 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 поддерживает добавление и удаление элементов с любого конца за $O(1)$ (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[1].

Объекты Python `deque` реализованы как двунаправленные связные списки (doubly-linked lists)[2]. Это придает им превосходную и стабильную производительность для операций вставки и удаления элементов, но при этом плохую $O(n)$ 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 библиотеке Python.

```
>>> from collections import deque
>>> q = deque()
>>> q.append('есть')
>>> q.append('спать')
>>> q.append('программировать')
```

[1] См. документацию Python «collections.deque»: https://docs.python.org/3.6/library/collections.html#collections.deque

[2] См. CPython «_collectionsmodule.c»: https://github.com/python/cpython/blob/master/Modules/_collectionsmodule.c

```
>>> q
deque(['есть', 'спать', 'программировать'])

>>> q.popleft()
'есть'

>>> q.popleft()
'спать'

>>> q.popleft()
'программировать'

>>> q.popleft()
IndexError: "pop from an empty deque"
```

queue.Queue — семантика блокирования для параллельных вычислений

Данная реализация очереди в стандартной библиотеке Python синхронизирована и обеспечивает семантику блокирования с целью поддержки многочисленных параллельных производителей и потребителей[1].

Модуль `queue` содержит несколько других классов, которые реализуют очереди с мультипроизводителями/мультипотребителями, которые широко используются в параллельных вычислениях.

В зависимости от вашего варианта использования семантика блокирования может оказаться полезной, а может накладывать ненужные издержки. В этом случае в качестве очереди общего назначения лучше всего использовать двустороннюю очередь `collections.deque`.

```
>>> from queue import Queue
>>> q = Queue()
>>> q.put('есть')
>>> q.put('спать')
>>> q.put('программировать')

>>> q
```

[1] См. документацию Python «queue.Queue»: https://docs.python.org/3.6/library/queue.html#module-queue

```
<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 параллельную обработку находящихся в очереди элементов многочисленными параллельными рабочими процессами[1]. Процессно-ориентированное распараллеливание популярно в Python из-за глобальной блокировки интерпретатора (GIL), которая препятствует некоторым формам параллельного 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 передавать любой консервируемый (модулем `pickle`) объект через границы процессов.

```
>>> from multiprocessing import Queue
>>> q = Queue()
>>> q.put('есть')
>>> q.put('спать')
```

[1] См. документацию Python «multiprocessing.Queue»: https://docs.python.org/3.6/library/multiprocessing.html#multiprocessing.Queue

```
>>> 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 производительности.
- 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 характеристики производительности, которые можно ожидать от хорошей реализации очереди, а также может применяться в качестве стека (очереди с дисциплиной доступа LIFO).

5.7. Очереди с приоритетом

Очередь с приоритетом представляет собой контейнерную структуру данных, которая управляет набором записей с полностью упорядоченными

ключами[1] (например, числовым значением *веса*) с целью обеспечения быстрого 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 планирования, 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 предпочтение перед задачами с более низким приоритетом (например, скачивание обновлений в фоновом режиме). Организовывая предстоящие задачи в очередь с приоритетом, которая использует актуальность задачи в качестве ключа, планировщик задач может быстро выбирать задачи 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 библиотекой Python. Каждая реализация будет иметь свои собственные преимущества 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 элемент. Недостатком является то, что вставка новых элементов в список является медленной $O(n)$ операцией.

---

[1] См. Википедию «Полная упорядоченность»: https://en.wikipedia.org/wiki/Total_order и https://ru.wikipedia.org/wiki/Линейно_упорядоченное_множество

Несмотря на то что точка вставки может быть найдена за *O(log n)* время с помощью алгоритма `bisect.insort`[1] стандартной библиотеки, это решение всегда находится во власти медленного шага вставки.

Поддержание упорядоченности путем добавления в конец списка и пересортировки также занимает минимум *O(n log n)* времени. Еще один недостаток — 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[2].

[1] См. документацию Python «bisect.insort»: https://docs.python.org/3.6/library/bisect.html
[2] См. документацию Python «heapq»: https://docs.python.org/3.6/library/heapq.html

Этот модуль — хороший выбор для реализации очередей с приоритетом в Python. Поскольку двоичная куча `heapq` технически обеспечивает только реализацию *min-heap* (то есть кучи, где значение в любой вершине не больше, чем значения ее потомков), должны быть предприняты дополнительные шаги, которые обеспечат стабильность сортировки и другие функциональные возможности, которые, как правило, ожидают от «практической версии» очереди с приоритетом[1].

```
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 пространственную вычислительную сложность[2].

Разница состоит в том, что очередь с приоритетом `PriorityQueue` синхронизирована и обеспечивает семантику блокирования с целью поддержки многочисленных параллельных производителей и потребителей.

В зависимости от вашего варианта использования она либо станет полезной, либо слегка замедлит вашу программу. В любом случае вы мо-

[1] См. документацию Python «Примечания к реализации очереди с приоритетом»: там же.

[2] См. документацию Python «queue.PriorityQueue»: https://docs.python.org/3.6/library/queue.html

жете предпочесть интерфейс на основе класса, предлагаемый классом `PriorityQueue`, использованию интерфейса на основе функций, предлагаемого 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, которые вы можете использовать в своих программах.
- Реализация `queue.PriorityQueue` выбивается из общего ряда, отличаясь хорошим объектно-ориентированным интерфейсом и именем, которое четко указывает на ее направленность. Такая реализация должна быть предпочтительным вариантом.
- Если требуется избежать издержек, связанных с блокировкой очереди `queue.PriorityQueue`, то непосредственное использование модуля `heapq` также будет хорошим выбором.

6 Циклы и итерации

6.1. Написание питоновских циклов

Один из самых легких способов отличить разработчика с опытом работы на C-подобных языках, который совсем недавно перешел на Python, — посмотреть, 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 размер контейнера `my_items`, чтобы определить количество итераций.

В Python можно писать циклы, которые справляются с этими двумя задачами автоматически. И будет просто замечательно, если вы возьмете это на вооружение. Например, если вашему коду не придется отслеживать нарастающий 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-диапазоны в действительности не хранят отдельные значения, представляющие числовую последовательность, вместо этого они функционируют как итераторы и вычисляют значения последовательности на ходу[1].

Поэтому, вместо того чтобы на каждой итерации цикла вручную увеличивать 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-

[1] 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.

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 перебор 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 программированию. Обратите внимание, что в этом цикле больше не отслеживается 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 возвращать 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 поддерживают 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 значения. 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 использовать, 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 индексами цикла и условиями остановки.

- Циклы `for` в Python в действительности являются циклами «for each», которые могут напрямую перебирать элементы контейнера или последовательности.

6.2. Осмысление включений

Одно из моих любимых функциональных средств языка Python — включения в список[1]. 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, — небольшая 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 обыкновенный цикл `for`, то вы, вероятно, написали бы что-то типа этого:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x * x)
```

[1] 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.*

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 представленному 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 произвольного 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 наоборот. 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 собираемся 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 помощью 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 программирующие на Python разработчики должны уметь применять.

Прежде чем мы пойдем дальше, хочу подчеркнуть, что Python поддерживает 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 включениями 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 одним уровнем вложенности. Я обнаружил, что за этими границами в большинстве 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 применение сделают ваш программный код намного более питоновским.
- Конструкции включения попросту являются причудливым синтаксическим сахаром для шаблона с простым циклом `for`. Как только вы разберетесь 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 средство, 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 размеру подсписков.

Приведу пример. В операции нарезки используется знакомый синтаксис индексации «`[]`» со следующим шаблоном "`[начало:конец:шаг]`»:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst
[1, 2, 3, 4, 5]

# lst[начало:конец:шаг]
>>> lst[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:

```
>>> lst[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:

```
>>> lst[::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 хитрость: 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:

```
>>> lst = [1, 2, 3, 4, 5]
>>> del lst[:]
>>> lst
[]
```

Как видите, этот фрагмент удаляет все элементы из `lst`, но оставляет сам объект-список неповрежденным. В Python 3 для выполнения такой же работы также можно применить метод `lst.clear()`, который в зависимости от обстоятельств, возможно, будет более удобочитаемым шаблоном. Однако имейте в виду, что метод `clear()` 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 заполнения вручную:

```
>>> original_lst = lst
>>> lst[:] = [7, 8, 9]
>>> lst
[7, 8, 9]
>>> original_lst
[7, 8, 9]
>>> original_lst is lst
True
```

Приведенный выше пример кода заменил все элементы в `lst`, но не уничтожил 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_lst
[7, 8, 9]
>>> copied_lst is lst
False
```

Создание *мелкой* копии означает, что копируется только структура элементов, 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 элементов 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 остальных коллег в вашей команде.

6.4. Красивые итераторы

Мне нравится то, как синтаксис Python отличается своей красотой и ясностью 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, итерации 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: объекты, которые поддерживают дандер-методы `__iter__` и `__next__`, автоматически 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, которые поддерживают протокол итератора. Они послужат в качестве «немагических» 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 увидели 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, существующим 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 отличается от примеров, которые вы видели в других пособиях по итераторам, 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, который 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 итеративном 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 `RepeaterIterator` мы создаем и возвращаем из дандер-метода `__iter__`? Это вспомогательный класс, который нам нужно определить, чтобы заработал наш пример итераций в цикле `for…in`:

```
class RepeaterIterator:
     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 `RepeaterIterator` с объектом `Repeater`, который его создал. Благодаря этому мы можем держаться за «исходный» объект, итерации по которому выполняются.

2. В `RepeaterIterator.__next__` мы залезаем назад в «исходный» экземпляр класса `Repeater` и возвращаем связанное с ним значение.

В этом примере кода `Repeater` и `RepeaterIterator` работают *вместе*, чтобы 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 действительно 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('Привет')
```

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 работающий 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 методы `__init__` и `__next__` работают вместе, делая объект Python итерируемым.

Профессиональный совет: если вы выполнили предыдущий пример в сеансе Python REPL или в терминале и хотите его остановить, нажмите сочетание 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, поддерживает 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 результат:

```
repeater = Repeater('Привет')
iterator = repeater.__iter__()
while True:
     item = iterator.__next__()
     print(item)
```

Как видите, конструкция `for…in` была всего лишь синтаксическим сахаром для простого цикла `while`:

- Этот фрагмент кода сначала подготовил объект `repeater` к итерации, вызвав его метод `__iter__`. Он вернул фактический *объект-итератор*.
- После этого цикл неоднократно вызывал метод `__next__` объекта-итератора, 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 инициализируем 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 потребляемой 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, оставаясь полностью изолированным от внутренней структуры последнего.

Имеете ли вы дело со списком элементов, словарем, бесконечной последовательностью, 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 образом 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 «эмулировать» 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 выдает то же самое приветствие.

Между прочим, здесь я воспользовался возможностью замены вызовов `__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 «фасад».

Python предлагает эти фасады также и для другой функциональности. Например, `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 исходного кода.

Более простой класс-итератор

До этого момента наш пример итератора состоял из двух отдельных классов, `Repeater` и `RepeaterIterator`. Они соответствовали непосредственно двум фазам, используемым в протоколе итератора Python: сначала подготовке 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 взвалить на один-единственный класс. Это позволит сократить объем программного 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 `RepeaterIterator`? Он был нужен, чтобы принять метод `__next__` для доставки новых значений из итератора. Но *место* определения метода `__next__` вовсе не имеет никакого значения. В протоколе итератора имеет значение только то, что метод `__iter__` возвращает любой объект с определенным на нем методом `__next__`.

Поэтому идея такая: `RepeaterIterator` 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 `RepeaterIterator` и реализовать итерируемый объект при помощи одного-единственного класса Python. Давайте попробуем! Наш пример с новым и упрощенным итератором 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 протокола 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 *бесконечно*.

Очевидно, бесконечное повторение не является главным вариантом использования 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 остановится. И вероятно, вы *не* ожидаете, что он захламит окно вашего терминала, без устали выводя «`3`», пока вы в дикой панике не начнете жать на Ctrl+C...

Пора узнать, как написать итератор, который в итоге *прекращает* генерировать 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 ограниченным повторителем `BoundedRepeater`. Он будет похож на наш предыдущий 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 качестве 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 итераторе значения, он вызывает исключение `StopIteration`.

Все верно: итераторы используют исключения для структуризации потока управления. Чтобы подать сигнал о завершении итераций, итератор Python просто вызывает встроенное исключение `StopIteration`.

Если я продолжу запрашивать значения из итератора, он продолжит вызывать 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 каждом 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 количества 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 расширенный 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 проверку на исключение `StopIteration` и при необходимости выходим из цикла `while`.

Возможность написать трехстрочный цикл `for…in` вместо восьмистрочного цикла `while` представляет собой вполне хорошее улучшение. И в результате 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 подчеркивания).

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 оригинальный метод `__next__` и пересылает возвращаемое из него значение. По существу, 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 фактические 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 последовательности, 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 написания итерируемых объектов в Python. Следует также рассмотреть генераторы и выражения-генераторы.

6.5. 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 большого объема шаблонного кода. И если говорить по правде, то как «ленивому» 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 последовательности 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 копирку 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 используется 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 repeater('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 нормальные функции, но их поведение очень различается. Начнем с того, что вызов функции-генератора вообще не выполняет функцию. Он просто создает и возвращает *объект-генератор*:

```
>>> repeater('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 сохраняет ее локальное состояние. На практике это означает, что локальные переменные и состояние исполнения функции-генератора лишь откладываются 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 синтаксический 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* генератор. 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 управления 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('Всем 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('Всем 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 достигаем 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 громоздким. Я хотел продемонстрировать, как вызов инструкции `return` из генератора 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 порождать 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 неявную инструкцию `return None`. И вот как будет выглядеть наша окончательная 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 написания 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`.

6.6. Выражения-генераторы

По мере того как я все больше узнавал о протоколе итератора Python и различных способах его реализации в собственном коде, я стал понимать, что синтаксический сахар является повторяющейся темой.

Дело в том, что итераторы на основе класса и функции-генераторы выражают 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 незначительному 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 получаем от этого выгоду. Мы приступаем к работе со все более и более мощными структурными блоками, которые сокращают бесполезную рутину и позволяют достигать большего за меньшее время.

Ранее в этой книге вы увидели, как генераторы предлагают синтаксический сахар для написания итераторов на основе класса. *Выражения-генераторы* (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 сжатому 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 делает 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, определенный 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 выражения-генератора мы, похоже, получили те же самые результаты, которые мы получали из функции-генератора `bounded_repeater`.

Правда, есть одно маленькое предостережение: после того как выражение-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 включения 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 итерируемый «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()` с выражением-генератором, в результате чего вы сконструируете объект-список, содержащий 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-список прямо на месте, то в большинстве случаев вы с самого начала просто пишете включение в список.

Давайте рассмотрим синтаксическую структуру этого простого выражения-генератора поближе. Шаблон, который вы должны увидеть, выглядит следующим образом:

```
genexpr = (expression for item in collection)
```

Приведенный выше «образец» выражения-генератора соответствует следующей 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 более 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 инструкциями. 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 выражение-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. Поскольку 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 cond1
       for y in ys if cond2
```

```
    ...
    for z in zs if condN)
```

Образец выше переводится в следующую ниже логику функции-генератора:

```
for x in xs:
     if cond1:
         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 инструментом 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 составные 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 написать функцию-генератор или итератор на основе класса.

6.7. 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 выражений-генераторов Python, то вы в мгновение ока будете строить сжатые 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 представляют собой синтаксический сахар для написания итераторов на Python. Они абстрагируются от большей части шаблонного кода, необходимого во время написания итераторов на основе класса.

В то время как обычная функция производит одно-единственное возвращаемое значение, генераторы производят последовательность результатов. Можно сказать, что они *генерируют поток значений* на протяжении своего жизненного цикла.

Например, я могу определить следующий ниже генератор, который производит серию целочисленных значений от одного до восьми, поддерживая нарастающий счетчик и выдавая новое значение всякий раз, когда с ним вызывается функция `next()`:

```
def integers():
     for i in range(1, 9):
         yield i
```

Вы можете подтвердить такое поведение, выполнив данный ниже фрагмент кода в интерпретаторе REPL Python:

```
>>> chain = integers()
>>> 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 принимает 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 обработки 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 инвертирует 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 конец, 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*. Буферизация 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 $3 \times 3 = 9$.
3. Квадрат целого числа, выданный генератором `squared`, немедленно передается в генератор `negated`, который модифицирует его в –9 и выдает его снова.

Вы можете продолжать расширять эту цепочку генераторов, чтобы отстроить 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 довольно сжатая. С помощью небольшой уловки мы можем сжать определение 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 является то, что их не получится сконфигурировать с использованием аргументов 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 комбинировать выражения-генераторы и обычные генераторы на свой вкус. В случае с составными конвейерами это поможет улучшить удобочитаемость.

Ключевые выводы

- Генераторы могут состыковываться в цепочки, формируя очень эффективные 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 Трюки со словарем

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. Предположим, 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 написать функцию `greeting()`, которая будет возвращать пользователю приветствие 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 реализация технически работает — но только если идентификатор пользователя

является допустимым ключом в словаре `name_for_userid`. Если в функцию `greeting` передать *недопустимый* идентификатор пользователя, то она вызовет исключение:

```
>>> greeting(382)
'Привет, Элис!'

>>> greeting(33333333)
KeyError: 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 несколько претензий:

- он *неэффективен*, потому что он опрашивает словарь дважды;
- он *многословен*, поскольку, например, часть строки с приветствием повторяется;
- он не является *питоновским* — официальная документация Python, в частности, для таких ситуаций рекомендует использовать стиль программирования «легче попросить прощения, чем разрешения» (EAFP):

> Этот общепринятый стиль программирования на Python исходно предполагает существование допустимых ключей или атрибутов и отлавливает исключения, если предположение оказывается ложным[1].

Более эффективная реализация, которая следует принципам EAFP, могла бы вместо выполнения явной проверки на принадлежность ключа словарю 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 первоначальных 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 умолчанию», который можно использовать в качестве запасного значения[2]:

```
def greeting(userid):
     return 'Привет, %s!' % name_for_userid.get(
         userid, 'всем')
```

---

[1] См. глоссарий Python «EAFP»: https://docs.python.org/3.6/glossary.html?highlight=glossary

[2] См. документацию Python «dict.get()»: https://docs.python.org/3/tutorial/datastructures.html#dictionaries

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 является 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.

7.2. 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 порядке 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 словаре, 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 последовательностей.

Чтобы сравнить два кортежа, Python сначала сравнивает элементы, хранящиеся в индексной позиции `0`. Если они различаются, то он определяет исход сравнения. Если они равны, то сравниваются следующие два элемента 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 проблемы 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 упорядочиваются элементы. Вы можете управлять порядком их следования путем передачи *функции ключа* во встроенную функцию `sorted()`, которая изменит то, как будут сравниваться элементы словаря.

Функция ключа — это просто обычная функция Python, которая будет вызываться с каждым элементом перед тем, как делать сравнения. Функция ключа на входе получает элемент словаря, а на выходе возвращает требуемый «ключ» для сравнения порядка следования элементов.

К сожалению, слово «ключ» здесь используется в двух контекстах одновременно: функция ключа не касается ключей словаря, она просто отмечает каждый входной элемент произвольным *ключом сравнения*.

Теперь, возможно, нам стоит взглянуть на пример. Поверьте, понять функции 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 использовать 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 отсортирован 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 структурных 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, простое 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 функции 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, единственная цель которого — преобразовать одну структуру данных в другую.

Ключевые выводы

- Создавая сортированные «представления» словарей и другие коллекции, 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 обнаружите, 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 применять для имитации инструкций выбора `switch-case` в Python при помощи 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 ответвлений `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, возвращать 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 интуитивно ожидаете: мы просто обращаемся к списку по индексу и исполь-

зуем вызывной синтаксис «()», чтобы вызвать функцию и передать ей аргументы:

```
>>> funcs[0](2, 3)
5
```

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 поиска 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 используем 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 ознакомления с приведенным ниже примером вы сможете увидеть шаблон, необходимый для сведения определенных видов инструкций if к диспетчеризации 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 соответствующие математические расчеты на операндах x и y:

```
>>> def dispatch_if(operator, x, y):
...     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 применять 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, потому 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': lambda: x + y,
...         'sub': lambda: x – y,
...         'mul': lambda: x * y,
...         'div': 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 функции 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 зрения производительности это не идеально. В случае, если программный код нуждается 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 примере 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 инструкций `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 случаях 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, которая способна многому научить, если хорошенько над ней поразмыслить. Такой фрагмент код — это как коан в дзен-буддизме: вопрос или утверждение, используемое в практике дзен, чтобы вызвать сомнение и проверить достижения ученика.

Крошечный фрагмент кода, который мы обсудим в этом разделе, является 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 CPython.

От этого однострочника я получаю такой кайф, что как-то раз я даже напечатал 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 паузу, 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 приведенного 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, которые 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 рассматривает 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'[1].

И разумеется, это означает, что в Python булевы значения технически можно использовать в качестве индексов списка или кортежа:

```
>>> ['нет', 'да'][True]
'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` — представляют *одинаковый ключ словаря*. Когда интерпретатор вычисляет выражение-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 выражение-словарь:

```
>>> {True: 'да', 1: 'нет', 1.0: 'возможно'}
{True: 'возможно'}
```

[1] См. документацию Python «Иерархия стандартных типов»: https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy

Почему здесь в качестве ключа мы по-прежнему получаем `True`? Разве не должен ключ из-за повторных присваиваний в самом конце тоже поменяться 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 производительности: 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 результирующем 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 имеющиеся 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 быстрее 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 идеальны. 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 называется *хеш-конфликтом* и является особым случаем, с которым должны 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 выражения-словаря. Поэтому давайте выясним, играют ли хеш-значения ключей здесь тоже какую-то определенную роль.

Я определяю приведенный ниже класс как небольшой сыскной инструмент:

```
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 притворяются, что они являются эквивалентными любому другому объекту, но при этом с ними будет связано уникальное хеш-значение. Это позволит 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 эквивалентные 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» вызывается 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 источника 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 переопределениями в один-единственный словарь с окончательными значениями конфигурации.

Или, обобщая: иногда вам нужен способ объединить два или более словаря в один, чтобы результирующий словарь содержал комбинацию ключей и значений исходных словарей.

В этом разделе я покажу несколько способов сделать это. Сначала посмотрим на простой пример, чтобы можно было что-то обсуждать. Предположим, 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 значения `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 разрешения конфликтов для повторяющихся ключей.

В Python классическое решение задачи «слияния многочисленных словарей» 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()`, определяет 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()` настолько, 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[1]. Поэтому в Python 3.5+ есть еще один — и, пожалуй, более приятный — способ объединения произвольного количества словарей:

```
>>> zs = {**xs, **ys}
```

У этого выражения в точности такой же результат, что и у цепочки вызовов `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 понятным, 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 сопровождении.

В данном случае я склоняюсь к использованию нового синтаксиса при условии, что работаю с Python 3. Более того, при использовании опера-

[1] См. PEP 448 «Дополнительные обобщения распаковки»: https://www.python.org/dev/peps/pep-0448/

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 исполнения? Или, возможно, вам приходилось генерировать диагностическое сообщение, чтобы выводить некоторые параметры конфигурации...

Я был разочарован, и часто, тем, насколько трудно в Python читать некоторые структуры данных, когда они печатаются как текстовые строки. 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 неразборчивому 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 приятным форматированием, можно применить функцию `json.dumps()`:

```
>>> import json
>>> json.dumps(mapping, indent=4, sort_keys=True)

{
     "a": 23,
     "b": 42,
     "c": 12648430
}
```

Эти настройки конфигурации в результате получают хорошее и выделенное 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 модуля `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 `json.dumps` и скопипастить его в сеансе интерпретатора 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 словарей, здесь мы получаем то, что воспринимается значительно легче.

Вместе с тем, по сравнению с `json.dumps()`, она не представляет вложенные структуры визуально столь же хорошо. В зависимости от обстоятельств 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 непримитивных ключей и значений, поскольку это вызовет исключение `TypeError`.

8 Питоновские методы повышения производительности

8.1. Исследование модулей и объектов Python

Вы можете в интерактивном режиме исследовать модули и объекты непосредственно из интерпретатора Python. Это недооцененное функциональное 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 определений интерфейсов наизусть.

В Python дела обстоят по-другому — эффективный разработчик будет проводить массу времени в сеансах интерпретатора REPL, работая интерактивно 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 интерактивно, находясь внутри интерпретатора.

Эти приемы будут работать с любой версией Python — надо лишь запустить 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 использует 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 поисковой системе или заглянуть в официальную документацию Python в Сети. Однако встроенная в Python функция `dir()` позволяет вам получать 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 расположенный 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 продолжать 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 атрибутов к тем, которыми вы интересуетесь:

```
>>> [_ for _ in dir(datetime) if 'date' in _.lower()]
['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 подробные описания функций и классов, экспортируемых модулем `datetime`?

В этом случае вам поможет встроенная в Python функция `help()`. С ее помощью 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. Чтобы выйти из режима интерактивной справки, нужно нажать клавишу `q`. Это вернет вас назад к командной строке интерпретатора. Неплохая возможность, 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 документацию на основе атрибутов, определенных в объекте, и его строки документации 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 отформатированную 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 интерпретатора Python. Они также доступны вне Сети и работают без подключения к интернету, что может оказаться очень полезным в случае крайней необходимости.

Ключевые выводы

- Используйте встроенную функцию `dir()`, чтобы интерактивно исследовать 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` является 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 библиотеки?

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, некоторые 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 модификации глобальной среды нередко требуется, чтобы вы выполняли команды `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 доверия, 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 иного 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 программные средства, от которых они зависят, в виде библиотек с нативным (платформенно-ориентированным) кодом и средой выполнения интерпретатора, 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 небольшую 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, чтобы держать их в безупречном виде и отделенными от остальных. Но вы также 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
```

Эта команда за одну минуту создаст новую папку venv в текущем каталоге, а также заполнит ее базовой средой Python 3:

```
$ ls venv/
bin        Include    Lib        pyvenv.cfg
```

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 указывали 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[1].

Обратите внимание на то, как это изменило вид подсказки в строке командной оболочки, и теперь она содержит название активной виртуальной среды в круглых скобках: (venv). Давайте проверим, какой исполняемый 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

[1] В Windows команда activate выполняется напрямую, то есть ее не нужно загружать вместе с источником.

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 администратора. И во-вторых, инсталляция или обновление пакета с активной виртуальной 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 выполнить 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? Аналогично команде `activate`, существует команда `deactivate`, которая возвращает 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 должны 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 управления зависимостями, таких как определение зависимостей проекта при помощи файлов `requirements.txt`.

Если вы ищете материал с глубоким изложением этой темы и с дополнительными советами по поводу производительности, обратитесь к моему *Курсу управления зависимостями Python*[1], 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.

[1] См. https://dbader.org/products/managing-python-dependencies/?utm_source=python-tricks-book&utm_medium=pdf&utm_campaign=pytricks-book

- 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 CPython 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 исходный код, в Python используются компактные цифровые коды, константы и ссылки, которые представляют результат лексического и семантического 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 виртуальная машина Python работает с байткодом. На самом деле формат байткода считается деталью реализации, и не гарантируется, что он останется стабильным 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 использовать, 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 исходный код в промежуточный язык, прежде чем он его «выполнит». Так вот, если это правда, то мы должны увидеть результаты этого шага компиляции. И мы можем.

Каждая функция имеет атрибут `__code__` (в Python 3), который мы можем использовать, чтобы получить инструкции виртуальной машины, константы и переменные, используемые нашей функцией `greet`:

```
>>> greet.__code__.co_code
b'dx01|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 сэкономить пространство памяти. Константы... как бы сказать... константны, то есть они не подлежат изменению и используются попеременно в разных местах.

Поэтому вместо того, чтобы повторять фактические постоянные величины в потоке команд `co_code`, Python хранит константы отдельно в поисковой 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 рассмотрение 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 исходный 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 является составной частью стандартной библиотеки. Поэтому мы можем его просто импортировать и вызвать `dis.dis()` с функцией `greet` в качестве аргумента, чтобы получить более удобочитаемое представление о ее байткоде:

```
>>> import dis
>>> dis.dis(greet)
   2           0 LOAD_CONST           1 ('Привет, ')
               2 LOAD_FAST            0 (name)
               4 BINARY_ADD
               6 LOAD_CONST           2 ('!')
               8 BINARY_ADD
              10 RETURN_VALUE
```

Главное, что сделал дизассемблер, было разбиение потока команд и назначение каждому находящемуся в нем *коду операции* человекочитаемого имени, как, например, `LOAD_CONST`.

Вы также видите, как ссылки на константы и переменные теперь чередуются 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 исходной функции `greet()`.

Сначала он извлекает константу в индексе 1 `('Привет, ')` и помещает ее в *стек*. Затем он загружает содержимое переменной `name` и также помещает 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 машиной*. 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 операции: *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 получить 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 поэтому 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 посвящена 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 экспериментов с использованием виртуальной машины. По этой теме я порекомендовал бы книгу *Проектирование компиляторов: виртуальные машины* (Compiler Design: Virtual Machines, Wilhelm and Seidl).

Ключевые выводы

- CPython исполняет программы, сначала транслируя их в промежуточный 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. Возьмите новые навыки и идиомы, которые вы узнали из этой книги, и начните их использовать на практике. Не позвольте этой книге стать просто очередной прочитанной книгой по программированию.

Что, если с этого момента вы начнете усеивать свой программный код расширенными 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 хорошему 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 использовании их только там, где они имеют смысл и помогают делать программный код выразительнее.

И поверьте, через некоторое время ваши коллеги подхватят тренд. Если они задают вам вопросы, делитесь с ними полезными знаниями. Подтягивайте 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 поэтому 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[1]. Кроме того, непременно получите бесплатный экземпляр *Трюки Python: цифровой комплект инструментов*, доступный по адресу dbader.org/python-tricks-toolkit.

9.1. Бесплатные еженедельные советы для разработчиков на Python

Хотите еженедельную порцию советов для разработчика на Python, чтобы улучшить свою производительность и оптимизировать свой рабочий процесс? Есть хорошие новости! Я веду бесплатную электронную рассылку для таких, как вы, разработчиков на Python.

Электронные письма, которые я рассылаю, не являются обычными сообщениями в стиле «а вот и список популярных статей». Вместо этого

[1] См. https://dbader.org/youtube

я стремлюсь делиться по крайней мере одной оригинальной мыслью в неделю в формате короткого эссе.

Если вы хотели бы увидеть, из-за чего же такой ажиотаж, то отправляйтесь 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. Сложно развивать свой опыт программирования на Python в полном одиночестве.

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 программирования.

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 попросить 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 поддержку. 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 существенного. Кроме того, если вы не будете постоянно в Сети, то пропустите большую часть разговоров. А если вы *постоянно* в Сети, то ваша производительность пострадает от бесконечного потока уведомлений. Слабые чат-группы страдают теми же самыми недостатками.

- Социальный новостной сайт *Hacker News* предназначен для обсуждения и комментирования технических новостей. Он не способствует установлению долговременных отношений между комментаторами. Кроме того, на сегодня это одно из самых агрессивных сообществ в технологической 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, набрасывающиеся 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.

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 зарегистрированного 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 подходят 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 платформе, где можно задавать вопросы, получать ответы и делиться успехом. 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

Перевел с английского А. Логунов

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*Д. Дрошнев*
Художественный редактор	*С. Заматевская*
Корректоры	*Н. Викторова, И. Тимофеева*
Верстка	*Л. Егорова*

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 печатные профессиональные, технические и научные.

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 Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
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. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: 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,
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

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: 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; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **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).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

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.**

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
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 высокого 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 собственную сеть дистрибуции по 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 обязательно проследим, чтобы ваша книга постоянно 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 формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*