Секреты РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Скотт Мейерс, редактор серии

Секреты Python

59 рекомендаций по написанию эффективного кода

ВИЛЬЯМС

Бретт Слаткин

Секреты Python

59 рекомендаций по написанию
эффективного кода

Effective Python

59 Specific Ways to Write Better Python

Brett Slatkin

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Секреты Python

59 рекомендаций по написанию эффективного кода

Бретт Слаткин

Москва • Санкт-Петербург • Киев
2016

ББК 32.973.26-018.2.75
С47
УДК 681.3.07

Издательский дом "Вильямс"

Главный редактор *С.Н. Тригуб*

Зав. редакцией *В.Р. Гинзбург*

Перевод с английского и редакция канд. хим. наук *А.Г. Гузикевича*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Слаткин, Бретт.

С47 Секреты Python: 59 рекомендаций по написанию эффективного кода. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 272 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2078-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Бретт Слаткин

Секреты Python: 59 рекомендаций по написанию эффективного кода

Литературный редактор *И.А Попова*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 08.06.2016. Формат 70x100/16
Гарнитура Times. Усл. печ. л. 21,93. Уч.-изд. л. 11,2
Тираж 500 экз. Заказ № 3420

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

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2078-2 (рус.)
ISBN 978-0-13-403428-7 (англ.)

Оглавление

Содержание

Отзывы о книге Секреты Python

"Каждая рекомендация в книге Слаткина *Секреты Python* — это самостоятельный урок с наглядными примерами исходного кода. Благодаря этому вы можете читать главы в любом порядке, обращаясь непосредственно к тому разделу, который вас в данный момент больше всего интересует. Я буду рекомендовать эту книгу студентам в качестве компактного сборника ценных наставлений по широкому кругу вопросов для программистов среднего и высокого уровня, работающих с языком Python".

Брэндон Роудс, инженер-разработчик компании Dropbox Inc., председатель конференции PyCon 2016-2017

"Я программирую на языке Python уже не один год и всегда считала, что знаю его довольно неплохо. Благодаря этой сокровищнице ценных советов и методических рекомендаций я поняла, какие еще существуют резервы для того, чтобы заставить мой код на Python работать быстрее (например, за счет использования встроенных структур данных), облегчить его чтение (например, за счет использования именованных аргументов) и применять подходы, более соответствующие духу языка Python (например, использовать функцию `zip` для параллельного итерирования списков)".

Памела Фокс, преподаватель, Академия Хана

"Будь у меня под рукой эта книга тогда, когда я решился перейти с Java на Python, это помогло бы мне сэкономить массу времени, которое я потратил на многократное переписывание кода, что происходило всякий раз, когда я осознавал, что делаю что-то "не в духе Python". В книге собраны крупицы тех основных сведений, которые обязан знать каждый программист на языке Python, чтобы не спотыкаться о них на протяжении многих месяцев и даже лет работы, учась на собственных ошибках. Широта охватываемых тем просто поражает. Важность следования принципам PEP8, основные идиомы Python, проектирование функций, методов и классов, эффективное использование стандартной библиотеки, проектирование качественного API, тестирование программ и измерение производительности кода — обо всем этом вы сможете прочитать в книге. Фантастический материал, представляющий действительную ценность для любого программиста на языке Python, независимо от того, новичок он или опытный разработчик".

Майк Бейер, создатель фреймворка SQLAlchemy

"Книга *Секреты Python* с четко сформулированными в ней рекомендациями по улучшению стиля и функциональности программного кода на

языке Python позволит вам поднять свой профессиональный уровень еще на одну ступеньку".

Ли Калвер, разработчик-консультант, Dropbox Inc.

"Эта книга — полезнейший ресурс для опытных разработчиков на других языках программирования, которые хотят быстро освоить язык Python, не ограничиваясь при этом лишь базовыми языковыми конструкциями. Книга поможет в кратчайшие сроки приступить к написанию программ в стиле, максимально соответствующем парадигмам этого языка. Она отличается четкой структуризацией содержащейся в ней информации, а также ясностью изложения материала, что облегчает его усвоение, причем в каждом разделе или главе рассматривается отдельная тема. Эта книга охватывает большой спектр конструкций "чистого" языка Python, не сбивая читателя с толку обсуждением сложных аспектов более широкой экосистемы Python. Более опытным разработчикам предлагаются углубленные примеры языковых конструкций, с которыми они до этого просто не сталкивались, а также примеры редко используемых возможностей языка. Бросается в глаза та исключительная легкость, с которой автор оперирует различными категориями Python. Он использует свое профессиональное мастерство для того, чтобы предупредить читателя о возможных подводных камнях и типичных ситуациях, которые могут приводить к сбоям в работе программ. Помимо этого, в книге содержатся ценные указания, касающиеся тонких различий между версиями Python 2.*x* и 3.*x*, что пригодится тем, кому приходится иметь дело с различными вариантами Python".

Кэтрин Скотт, руководитель группы разработчиков, Tempo Automation

"Это замечательная книга как для новичков, так и для опытных программистов. Примеры кода тщательно продуманы, а соответствующие пояснения отличаются краткостью и ясностью".

Чарльз Титус Браун, адъюнкт-профессор, Калифорнийский университет в Дейвисе

"Это необычайно полезный источник сведений по эффективному использованию языка Python и созданию на его основе надежных и простых в сопровождении программ. Практическое применение изложенных в книге рекомендаций позволит программистам на языке Python повысить уровень своего профессионального мастерства".

Уэс МакКинни, создатель библиотеки pandas, автор книги "Python for Data Analysts", инженер-программист компании Cloudera

Об авторе

Бретт Слаткин — ведущий инженер-разработчик компании Google, а также руководитель группы разработчиков и соучредитель компании Google Consumer Surveys. До этого разрабатывал инфраструктуру Python для сервиса хостинга Google App Engine и участвовал в создании протокола PubSubHubbub. На протяжении последних 9 лет применяет Python в системе управления огромной сетью серверов Google.

В свободное от основной работы время занимается разработкой инструментальных средств с открытым исходным кодом и написанием статей, которые публикует у себя в блоге (`http://onebigfluke.com`). Степень бакалавра в области компьютерной инженерии получил в Колумбийском университете в Нью-Йорке. В настоящее время проживает в Сан-Франциско.

Введение

Язык программирования Python обладает уникальными возможностями, осознать которые поначалу не так-то просто. Предыдущий опыт многих программистов, знакомых с другими языками, часто мешает им в полной мере оценить выразительность средств Python и эффективно использовать предлагаемые возможности. Нередко встречаются программисты, которые впадают в другую крайность, проявляя чрезмерное усердие в попытках выжать из Python все, что только возможно. Однако при отсутствии надлежащего опыта такой подход может порождать серьезные проблемы, с которыми впоследствии приходится бороться.

В книге тщательно анализируется стиль написания программ, который принято характеризовать фразой *в духе Python* (Pythonic way) и который позволяет максимально эффективно задействовать все возможности этого языка. Автор предполагает, что основы Python читателю уже известны. Новички ознакомятся здесь с наилучшими методиками применения Python. Опытные программисты научатся уверенно применять новые для них и поэтому непривычные на первых порах инструменты этого языка.

Моя задача заключается в том, чтобы подготовить вас к эффективному использованию языка Python.

О чем эта книга

Каждая глава представляет собой сборник рекомендаций, объединенных в рамках одной темы. Можете читать их в произвольном порядке, выбирая то, что в данный момент вас больше всего интересует. Рекомендации включают советы относительно того, как решить ту или иную задачу, избежать типичных ошибок или обеспечить разумный компромисс в спорных моментах, а также позволяют осознанно подойти к выбору вариантов решений, являющихся наилучшими в тех или иных ситуациях.

Приведенные в книге рекомендации адресованы программистам, работающим с версиями Python 3 и Python 2. Большинство рекомендаций подойдет также программистам, использующим альтернативные варианты реализации Python, такие как Jython, IronPython, PyPy и др.

Глава 1. Мыслим категориями языка Python

В Python-сообществе бытует выражение "в духе Python". Так говорят о программах, выдержанных в стиле, максимально эффективно задействующем особенности этого языка. Идиомы Python выработаны на основе живого практического опыта программистов, использующих этот и другие языки. В этой главе описаны наилучшие подходы к решению задач, с которыми часто сталкиваются программисты, работающие с языком Python.

Глава 2. Функции

Функции в языке Python обладают целым рядом дополнительных особенностей, облегчающих жизнь программистам. Некоторые из них имеют аналоги в других языках, но многие свойственны только Python. Эта глава посвящена использованию функций для раскрытия намерений программиста, содействия повторному использованию кода и уменьшения вероятности того, что в него вкрадутся ошибки.

Глава 3. Классы и наследование

Python — объектно-ориентированный язык. Чтобы что-то сделать в Python, часто требуется написать классы и определить способы их взаимодействия через интерфейсы и иерархию. В этой главе рассказывается о том, как использовать классы и наследование для создания объектов с определенным поведением.

Глава 4. Метаклассы и атрибуты

Метаклассы и динамические атрибуты — мощные средства Python. В то же время они позволяют реализовывать чрезвычайно причудливые и неожиданные варианты поведения. В этой главе рассмотрены наиболее распространенные идиомы для использования этих механизмов, гарантирующие соблюдение принципа *наименьшей неожиданности*.

Глава 5. Одновременность и параллелизм

Язык Python упрощает написание программ, обеспечивающих одновременное выполнение нескольких вычислительных задач. Кроме того, Python может быть использован для организации параллельных вычислений посредством системных вызовов, подпроцессов и C-расширений.

В этой главе обсуждается наилучшая практика применения Python в подобных ситуациях.

Глава 6. Встроенные модули

Пакет Python включает много важных модулей, которые понадобятся вам при написании программ. Стандартные модули настолько тесно переплетены с идиоматикой Python, что могли бы выступать в качестве составной части спецификации языка. В этой главе рассмотрены наиболее важные из встроенных модулей.

Глава 7. Совместная работа

Совместная работа над программами на языке Python требует особенно внимательного отношения к написанию кода. Но даже если вы работаете один, вам все равно надо знать, как использовать модули, написанные другими разработчиками. Эта глава посвящена описанию стандартных инструментальных средств и наилучшей практики организации совместной разработки программ на языке Python.

Глава 8. Производство

В языке Python предусмотрены средства адаптации к широкому кругу сред развертывания. Он также включает встроенные модули, нацеленные на повышение надежности и защищенности программ. В этой главе обсуждаются предлагаемые в Python средства отладки, оптимизации и тестирования, предназначенные для улучшения качества и производительности программ.

Соглашения, принятые в книге

Исходный код примеров, а также имена файлов, классов, объектов, методов и переменных выделяются в тексте моноширинным шрифтом. В необходимых случаях для разбивки длинных строк кода или текста используется символ ↳. Пропускам в коде примеров, обозначенным символами комментария (`#...`), соответствуют фрагменты кода, несущественные для текущего обсуждения. С целью сокращения размера приводимого кода описание примеров средствами встроенного документирования не применялось. Однако я настоятельно рекомендую вам использовать эту возможность при разработке собственных проектов, следуя общепринятым рекомендациям по стилю (раздел "Рекомендация 2. Руководствуйтесь правилами стилевого оформления программ, изложенными в документе PEP8") и обязательно дополняя свой

код строками документирования (раздел "Рекомендация 49. Снабжайте строками документирования каждую функцию, класс и модуль").

Для большинства примеров книги показан соответствующий вывод. Говоря о "выводе", я подразумеваю выводимую на консоль или терминал информацию, которую вы видите при выполнении программы в окне интерпретатора команд. Разделы вывода выделяются моноширинным шрифтом, и им предшествует строка >>> (приглашение интерпретатора Python). Идея состоит в том, чтобы вы могли вводить код примеров в интерактивной оболочке Python и сравнивать полученные результаты с приведенными в книге.

Наконец, перед некоторыми разделами, выделенными моноширинным шрифтом, отсутствуют строки >>>. Такие разделы представляют вывод, полученный при выполнении программы вне интерактивной оболочки Python. Примеры этого типа часто начинаются символом $, указывающим на то, что данная программа запускалась в командной строке оболочки типа Bash.

Где найти код примеров

Некоторые примеры вам будет полезно просмотреть целиком, а не в виде фрагментов кода, перемежающихся с текстом. Это также даст вам возможность самостоятельно поработать с кодом, чтобы понять, почему программа работает именно так, а не иначе. Исходный код всех примеров доступен на сайте http://www.effectivepython.com. На этом же сайте вы найдете информацию об ошибках и опечатках, обнаруженных в книге.

Ждем ваших отзывов!

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

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

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

Наши электронные адреса:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152

1 Мыслим категориями языка Python

Идиомы языка программирования определяются его пользователями. В Python-сообществе прижилось выражение "в духе Python" (Pythonic way), подразумевающее особый стиль программирования, свойственный только этому языку. Этот стиль никем не регламентирован и не навязывается компилятором. Он является продуктом своей эпохи, зародившимся на основе индивидуального и коллективного опыта использования Python программистами. Последние придерживаются философии, в соответствии с которой явное лучше, чем неявное, простое лучше, чем сложное, а удобочитаемость имеет большое значение (для ознакомления с основными принципами *дзен-философии Python* достаточно выполнить команду `import this`).

Программисты, знакомые с другими языками, могут пытаться писать программы на Python так, как они писали бы их на C++, Java или любом другом языке, который они знают лучше всего. Новичкам же, вероятно, придется долго привыкать к широкому диапазону понятий, выражаемых на языке Python. Важно, чтобы представители обеих категорий были знакомы с наилучшими — *в духе Python* — подходами к решению типичных задач. Владение этими шаблонами программирования положительно скажется на любой из создаваемых вами программ.

Рекомендация 1. Следите за тем, какую версию Python вы используете

В большинстве примеров кода, приведенных в книге, используется синтаксис версии Python 3.4 (выпуск 17 марта 2014 г.). В некоторых примерах специально для того, чтобы обратить внимание на важные различия между версиями, используется синтаксис версии Python 2.7 (выпуск 3 марта 2010 г.). Большинство моих советов применимо ко

всем популярным средам выполнения: CPython, Jython, IronPython, PyPy и др.[1]

На компьютере может быть несколько предустановленных версий стандартной среды выполнения, но какая именно из них будет вызвана командой python, введенной в командной строке, вам не всегда может быть известно. Обычно python является псевдонимом python2.7, но в некоторых случаях эта команда может вызывать более ранние версии: python2.6 или даже python2.5. Вы сможете определить, какую версию используете, воспользовавшись флагом --version.

```
$ python --version
Python 2.7.8
```

Обычно версия Python 3 доступна под именем python3.

```
$ python3 --version
Python 3.4.2
```

Номер используемой версии Python можно определить и на этапе выполнения, просмотрев значения, хранящиеся во встроенном модуле sys.

```
import sys
print(sys.version_info)
print(sys.version)

>>>
sys.version_info(major=3, minor=4, micro=2,
↳ releaselevel='final', serial=0)
3.4.2 (default, Oct 19 2014, 17:52:17)
↳ [GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)]
```

Python-сообществом активно поддерживаются обе линейки — Python 2 и Python 3. Разработка Python 2 прекращена, за исключением исправления ошибок, улучшения системы безопасности и ретроподдержки, облегчающей переход от Python 2 к Python 3. Для упрощения адаптации к последующим версиям линейки Python 3 существуют такие вспомогательные инструменты, как 2to3 и six.

В Python 3 постоянно добавляются новые возможности и улучшения, которые никогда не будут добавлены в Python 2. На момент написания этой книги большинство наиболее распространенных библиотек Python с открытым исходным кодом было совместимо с Python 3. Я настоятельно рекомендую вам использовать Python 3 в своем следующем проекте.

---

[1] Исходная реализация — CPython — является стандартом. Альтернативные реализации ориентированы на интеграцию с другими языками (Jython, Iron Python), параллельные вычисления (Stackless Python), реализацию языков программирования (PyPy) и пр. — *Примеч. ред.*

Что следует запомнить

- В настоящее время активно используются две основные версии Python: Python 2 и Python 3.
- Существует целый ряд популярных сред выполнения для Python: CPython, Jython, IronPython, PyPy и др.
- Убедитесь в том, что в командной строке вашей системы вызывается именно та версия Python, на которую вы рассчитываете.
- Используйте в своем следующем проекте версию Python 3, поскольку именно на ней сосредоточено основное внимание Python-сообщества.

Рекомендация 2. Руководствуйтесь правилами стилевого оформления программ, изложенными в документе PEP 8

Документ *Python Enhancement Proposal #8* (сокр. *PEP 8*) содержит предложения по стилевому оформлению кода программ на языке Python. Вообще говоря, вы вправе форматировать свой код так, как считаете нужным, коль скоро в нем соблюдены правила синтаксиса. Однако применение единообразного стиля облегчит изучение кода другими людьми и улучшит его удобочитаемость. Совместное использование общего стиля с другими Python-программистами в рамках большого сообщества способствует улучшению качества программ при коллективной работе над проектами. Но даже если единственный человек, который когда-либо будет читать ваш код, — это вы, соблюдение рекомендаций PEP 8 облегчит внесение последующих изменений в код, если в этом возникнет необходимость.

Документ PEP 8 содержит детализированные правила написания корректно оформленного кода на Python. По мере развития языка этот документ постоянно обновляется. Было бы неплохо, если бы вы прочитали целиком все руководство (`http://www.python.org/dev/peps/pep-0008`). Ниже приведены некоторые правила, которых следует обязательно придерживаться.

Пробелы. В языке Python пробелы имеют синтаксическое значение. Особое значение Python-программисты придают влиянию пробелов на удобочитаемость кода.

- Используйте пробелы, а не символы табуляции, для создания отступов.
- Используйте по 4 пробела для каждого уровня синтаксически значимых отступов.

- Длина строк не должна превышать 79 символов.
- Дополнительные строки, являющиеся продолжением длинных выражений, должны выделяться четырьмя дополнительными пробелами сверх обычного их количества для отступов данного уровня.
- Между определениями функций и классов в файлах следует вставлять две пустые строки.
- Между определениями методов в классах следует вставлять одну пустую строку.
- Не окружайте пробелами индексы элементов списков, вызовы функций и операторы присваивания значений именованным аргументам.
- Помещайте по одному и только одному пробелу до и после оператора присваивания.

Имена. В документе PEP 8 для различных элементов языка предлагается свой стиль имен. Благодаря этому можно легко определить в процессе чтения кода, какому типу соответствует то или иное имя.

- Имена функций, переменных и атрибутов должны следовать формату `lowercase_underscore` (нижний регистр букв, разделение слов символами подчеркивания).
- Имена защищенных атрибутов экземпляра должны следовать формату `_leading_underscore` (один символ подчеркивания в начале).
- Имена закрытых атрибутов экземпляра должны следовать формату `__leading_underscore` (два символа подчеркивания в начале).
- Имена классов и исключений должны следовать формату `CapitalizedWord` (каждое слово начинается с прописной буквы).
- Константы уровня модуля должны записываться в формате `ALL_CAPS` (все буквы прописные, в качестве разделителя используется символ подчеркивания).
- В определениях методов экземпляров классов в качестве имени первого параметра следует всегда указывать `self` (это имя ссылается на текущий объект).
- В определениях методов классов в качестве имени первого параметра следует всегда указывать `cls` (это имя ссылается на текущий класс).

Выражения и инструкции. Одно из положений дзен-философии Python гласит: "Должен существовать один — и предпочтительно только один — очевидный способ сделать это". В рекомендациях документа

PEP 8 предпринимается попытка кодифицировать такой стиль написания выражений и предложений.

- Используйте встроенные отрицания (`if a is not b`), а не отрицание утвердительных выражений (`if not a is b`).
- Не тестируйте пустые значения (такие, как `[]` или `''`), проверяя их длину (`if len(somelist) == 0`). Вместо этого используйте проверку `if not somelist`, исходя из того, что результат вычисления пустого выражения неявно трактуется как `False`.
- То же самое касается и непустых значений (таких, как `[1]` или `'hi'`): в инструкции `if somelist` результат вычисления непустого значения неявно трактуется как `True`.
- Избегайте записи инструкций `if`, циклов `for` и `while`, а также сложных инструкций `except` в одной строке. Размещайте их на нескольких строках, чтобы сделать код более понятным.
- Всегда помещайте инструкции `import` в самом начале файла.
- Для импорта модулей всегда используйте их абсолютные имена, а не имена, заданные относительно пути к текущему модулю. Например, чтобы импортировать модуль `foo` из пакета `bar`, следует использовать инструкцию `from bar import foo`, а не просто `import foo`.
- Если требуется выполнить относительный импорт, то используйте явный синтаксис: `from . import foo`.
- Импортируемые модули должны располагаться в разделах, указываемых в следующем порядке: 1) модули стандартных библиотек; 2) модули сторонних разработчиков; 3) ваши собственные модули. В каждом подразделе модули должны располагаться в алфавитном порядке.

Примечание

Средство Pylint (`http://www.pylint.org/`) — это популярный статический анализатор исходного кода на языке Python. Pylint обеспечивает автоматическое обнаружение отклонений стиля оформления кода от стандартов PEP 8 и выявляет многие другие типы распространенных ошибок в программах на языке Python.

Что следует запомнить

- Всегда следуйте рекомендациям по оформлению кода Python, изложенным в документе PEP 8.

- Использование общего с многочисленными членами Python-сообщества стиля оформления кода облегчает коллективную разработку.
- Использование единообразного стиля упрощает изменение кода впоследствии.

Рекомендация 3. Знайте о различиях между типами строк bytes, str и unicode

В Python 3 существуют два типа данных, представляющих последовательности символов: `bytes` и `str`. Экземпляры `bytes` содержат так называемые "сырые" (необработанные) битовые значения. Экземпляры `str` содержат символы Unicode.

В Python 2 для представления последовательности символов также предусмотрены два типа: `str` и `unicode`. В отличие от Python 3 экземпляры `str` содержат "сырые" 8-битовые значения. Экземпляры `unicode` содержат символы Unicode.

Символы Unicode можно представлять в виде двоичных данных ("сырых" 8-битовых значений) многими способами. Наиболее распространенной кодировкой является *UTF-8*. Важно отметить, что экземпляры `str` в Python 3 и `unicode` в Python 2 не имеют ассоциированных с ними двоичных кодировок. Для преобразования символов Unicode в двоичные данные необходимо использовать метод `encode()`. Для преобразования двоичных данных в символы Unicode необходимо использовать метод `decode()`.

При написании программ на Python важно выполнять кодирование и декодирование символов Unicode на наиболее удаленных границах интерфейсов. В основной части программы должны использоваться символы Unicode (`str` в Python 3 и `unicode` в Python 2), и никакие предположения относительно кодировки символов делаться не должны. Такой подход обеспечивает достаточную терпимость к использованию альтернативных кодировок входных данных (таких, как *Latin-1*, *Shift JIS* и *Big5*) и в то же время достаточно строг в отношении кодировки выходного текста (в идеальном случае — UTF-8).

Описанное разделение символьных типов приводит к возникновению двух распространенных ситуаций в коде Python:

- вы хотите работать с "сырыми" 8-битовыми значениями, которые представляют собой символы в кодировке UTF-8 (или какой-либо другой);
- вы хотите работать с символами Unicode, не имеющими какой-либо определенной кодировки.

Вам часто будут требоваться две вспомогательные функции: одна из них будет выполнять преобразования между этими двумя случаями, а вторая — гарантировать, что тип входных значений соответствует тому, который ожидается в коде.

В Python 3 вам потребуется метод, который принимает тип `str` или `bytes` и всегда возвращает тип `str`.

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value # Экземпляр str
```

Вам потребуется еще один метод, который принимает тип `str` или `bytes` и всегда возвращает тип `bytes`.

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # Экземпляр bytes
```

В Python 2 вам потребуется метод, который принимает тип `str` или `unicode` и всегда возвращает тип `unicode`.

```
# Python 2
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value # Экземпляр unicode
```

Вам потребуется еще один метод, который принимает тип `str` или `unicode` и всегда возвращает тип `str`.

```
# Python 2
def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value # Экземпляр str
```

При обработке “сырых” 8-битовых значений и символов Unicode в Python возникают два тонких момента.

Один из них связан с тем, что в Python 2 строки `unicode` и `str` внешне ведут себя одинаково, если строка `str` содержит лишь 7-битовые символы ASCII.

- Вы можете объединять такие строки `str` со строками `unicode` с помощью оператора `+`.
- Вы можете сравнивать экземпляры таких строк `str` и `unicode`, используя операторы равенства и неравенства.
- Вы можете использовать экземпляры `unicode` для форматирования строк с помощью оператора форматирования (`'%s'`).

Подобное сходство поведения означает, что во многих случаях строки `str` и `unicode` при передаче их в функцию, ожидающую какой-либо один из этих двух типов, являются взаимозаменяемыми (если вы имеете дело только с 7-битовыми ASCII-символами). В Python 3 строки `bytes` и `str` никогда не являются эквивалентными, даже если они пустые, и поэтому вы должны более тщательно контролировать типы символьных последовательностей при передаче их функциям.

Второй момент заключается в том, что в Python 3 операции, включающие дескрипторы файлов (возвращаемые встроенной функцией `open()`), выполняются с использованием кодировки UTF-8. В Python 2 файловые операции по умолчанию выполняются с использованием двоичной кодировки. Это может приводить к появлению ошибок, кажущихся неожиданными, особенно программистам, привыкшим работать с Python 2.

Например, предположим, что вы хотите записать в файл некоторые двоичные данные. Приведенный ниже код в Python 2 сработает, а в Python 3 — нет.

```
with open('random.bin', 'w') as f:
    f.write(os.urandom(10))

>>>
TypeError: must be str, not bytes
```

Данное исключение возникло по той причине, что в Python 3 у функции `open()` появился новый параметр: `encoding`. По умолчанию этот параметр имеет значение `'utf-8'`. Поэтому методы `read()` и `write()` ожидают экземпляры `str`, содержащие символы Unicode, а не экземпляры `bytes`, содержащие двоичные данные.

Чтобы все работало так, как надо, вы должны указать, что файл открывается в режиме записи двоичных (`'wb'`), а не текстовых (`'w'`) данных. Ниже я использую этот метод так, чтобы он корректно работал как в Python 2, так и в Python 3.

```
with open('random.bin', 'wb') as f:
    f.write(os.urandom(10))
```

Аналогичная проблема возникает и при чтении данных из файлов. Решение остается тем же: при открытии файла следует указать двоичный режим (`'rb'`), а не текстовый (`'r'`).

Что следует запомнить

- В Python 3 экземпляры `bytes` содержат последовательности 8-битовых значений, а экземпляры `str` — последовательности символов Unicode. Совместное использование экземпляров `bytes` и `str` в операциях (таких, как `>` или `+`) не допускается.
- В Python 2 экземпляры `str` содержат 8-битовые значения, а экземпляры `unicode` — символы Unicode. Совместное использование экземпляров `str` и `unicode` допускается в том случае, если экземпляры `str` содержат 7-битовые символы ASCII.
- Если вам необходимо выполнить файловые операции чтения или записи двоичных данных, всегда открывайте файл в режиме чтения и записи двоичных данных (`'rb'` и `'wb'`).

Рекомендация 4. Заменяйте сложные выражения вспомогательными функциями

Лаконичный синтаксис языка Python упрощает написание однострочных выражений, реализующих достаточно сложную логику. Предположим, вам нужно декодировать строку запроса, которая содержится в URL-адресе. В приведенном ниже примере каждый параметр представляет целое число.

```
from urllib.parse import parse_qs
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
print(repr(my_values))

>>>
{'red': ['5'], 'green': [''], 'blue': ['0']}
```

Одни параметры строки запроса могут представлять сразу несколько значений, другие — только одно. Кроме того, некоторые параметры могут содержать пустые значения или вообще отсутствовать в запросе. Применение метода `get()` к результирующему словарю позволяет получить значения в любой из этих ситуаций.

```
print('Красный:        ', my_values.get('red'))
print('Зеленый:        ', my_values.get('green'))
print('Непрозрачность: ', my_values.get('opacity'))

>>>
Красный:        ['5']
Зеленый:        ['']
Непрозрачность: None
```

Было бы неплохо, если бы отсутствующим или пустым параметрам по умолчанию присваивались нулевые значения. Вы вполне можете сделать это с помощью булевых выражений, поскольку логика в данном случае настолько проста, что, на первый взгляд, не заслуживает применения полной инструкции `if` или вспомогательной функции.

Синтаксис Python позволяет чрезвычайно просто реализовать подобный вариант. Суть трюка состоит в том, что и пустой строке, и пустому списку, и нулю неявно соответствуют ложные значения. Таким образом, вычисление каждого из приведенных ниже выражений сводится к вычислению подвыражения, стоящего за оператором `or`, если вычисление первого подвыражения дает ложное значение.

```
# Для строки запроса 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print('Красный:        %r' % red)
print('Зеленый:        %r' % green)
print('Непрозрачность: %r' % opacity)

>>>
Красный:        '5'
Зеленый:        0
Непрозрачность: 0
```

Параметр `red` работает, поскольку в словаре `my_values` представлен соответствующий ключ. Его значением является список с единственным элементом — строкой `'5'`. Эта строка неявно эквивалентна истинному значению, поэтому переменной `red` присваивается значение первой части выражения `or`.

Параметр `green` работает, поскольку в словаре `my_values` для него имеется значение в виде списка с одним элементом — пустой строкой. Пустая строка интерпретируется как ложное значение, в результате чего вычисление выражения `or` дает `0`.

Параметр `opacity` работает, поскольку в словаре `my_values` он вообще не представлен. В ситуациях, когда запрашиваемый ключ отсутствует

в словаре, метод `get()` возвращает второй аргумент. В данном случае значением по умолчанию является список с одним элементом — пустой строкой. Обнаружив, что `opacity` отсутствует в словаре, этот код делает в точности то же, что и в случае параметра `green`.

Однако данное выражение трудно читать, и к тому же оно не в полной мере выполняет все, что нам нужно. Ведь нам желательно иметь гарантии того, что все параметры имеют целочисленные значения, чтобы впоследствии их можно было использовать в математических выражениях. Для преобразования строки в целое число можно воспользоваться встроенной функцией `int()`:

```
red = int(my_values.get('red', [''])[0] or 0)
```

После таких манипуляций с кодом читать данное выражение стало еще труднее. В нем слишком много визуальных помех. Разбирать такой код весьма непросто. Тому, кто читает его в первый раз, придется потратить много времени для того, чтобы понять его смысл. Поэтому, несмотря на привлекательность краткой записи, не стоит пытаться втискивать все в одну строку.

Чтобы сделать подобные фрагменты кода более понятными, но одновременно обеспечить их компактность, в версии Python 2.5 были введены условные выражения `if/else` (их также называют *тернарными условными выражениями*), которые позволяют улучшить читаемость кода, одновременно сохраняя его компактный вид.

```
red = my_values.get('red', [''])
red = int(red[0]) if red[0] else 0
```

Вот это уже лучше. В менее сложных ситуациях условные выражения `if/else` могут сделать код значительно более понятным. Однако в отношении ясности кода последний вариант все еще проигрывает альтернативе в виде полного оператора `if/else`, охватывающего несколько строк. Понять рассредоточенную логику, представленную ниже, значительно легче, чем ее компактный вариант.

```
green = my_values.get('green', [''])
if green[0]:
    green = int(green[0])
else:
    green = 0
```

Можно продвинуться немного дальше в нужном направлении, используя вспомогательную функцию, особенно если предполагается многократное применение кода с данной логикой.

```
def get_first_int(values, key, default=0):
    found = values.get(key, [''])
```

```
    if found[0]:
        found = int(found[0])
    else:
        found = default
    return found
```

Вызывающий код гораздо более понятен, чем сложное выражение с оператором or или его двухстрочная версия с выражением if/else.

```
green = get_first_int(my_values, 'green')
```

Как только вы видите, что выражение начинает усложняться, разбейте его на меньшие части или вынесите логику во вспомогательные функции. Выигрыш в ясности кода всегда перевесит те преимущества, которые обеспечиваются его краткостью. Не дайте лаконичному синтаксису языка Python завлечь вас в дебри запутанного кода.

Что следует запомнить

- Синтаксис языка Python позволяет с чрезвычайной легкостью писать однострочные выражения, с которыми, однако, связан риск неоправданного усложнения кода, становящегося трудными для восприятия.
- Перемещайте сложные выражения во вспомогательные функции, особенно если предполагается многократное использование одной и той же логики.
- Выражение if/else обеспечивает более удобочитаемую альтернативу использованию булевых операторов наподобие or или and в выражениях.

Рекомендация 5. Умейте работать со срезами последовательностей

Язык Python включает синтаксис для извлечения целых разделов последовательностей, так называемых *срезов* (slices), а не отдельных элементов. Срезы чрезвычайно упрощают доступ к подмножествам элементов последовательностей. Особенно просто применять срезы в случае встроенных типов, таких как list, str и bytes, однако область их применения может быть расширена на любой класс Python, реализующий специальные методы __getitem__() и __setitem__() (раздел "Рекомендация 28. Используйте наследование от классов из модуля collections.abc для создания пользовательских контейнерных типов").

Базовая форма синтаксиса срезов представляет собой запись вида somelist[start:end], где начало диапазона (элемент с индексом start)

включается в срез, а конец диапазона (элемент с индексом end) не включается.

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print('Первые четыре:    ', a[:4])
print('Последние четыре: ', a[-4:])
print('Средние два:      ', a[3:-3])

>>>
Первые четыре:    ['a', 'b', 'c', 'd']
Последние четыре: ['e', 'f', 'g', 'h']
Средние два:      ['d', 'e']
```

Если начало среза совпадает с началом списка, то нулевой индекс следует опускать, чтобы снизить уровень визуальных помех.

```
assert a[:5] == a[0:5]
```

Если диапазон среза распространяется до конца списка, то конечный индекс также следует опускать ввиду его избыточности в данном случае.

```
assert a[5:] == a[5:len(a)]
```

Отрицательные значения индексов удобны для определения смещений относительно конца списка. Любая из приведенных ниже форм записи срезов будет понятна тому, кто впервые читает ваш код. Здесь вас не подстерегают никакие сюрпризы, и я настоятельно рекомендую использовать эти разновидности срезов.

```
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #                     ['e', 'f', 'g', 'h']
a[-3:]    #                         ['f', 'g', 'h']
a[2:5]    #           ['c', 'd', 'e']
a[2:-1]   #           ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  #                         ['f', 'g']
```

При извлечении срезов обеспечивается корректная обработка выхода начального или конечного индекса за границы списка. Это упрощает задание максимальной длины обрабатываемого среза для входных последовательностей.

```
first_twenty_items = a[:20]
last_twenty_items = a[-20:]
```

В противоположность этому непосредственный доступ к элементу списка по тому же индексу вызывает исключение.

```
a[20]

>>>
IndexError: list index out of range
```

Примечание

Имейте в виду, что индексирование списка отрицательными значениями переменной — один из немногих случаев, в которых извлечение среза может приводить к неожиданным результатам. Например, выражение `somelist[-n:]` будет отлично работать, если n больше единицы (например, `somelist[-3:]`). Но при нулевом значении n выражение `somelist[-0:]` создает копию исходного списка.

Результатом извлечения среза из списка является совершенно новый список. Ссылки на объекты исходного списка сохраняются. Изменение результата извлечения среза никак не сказывается на исходном списке.

```
b = a[4:]
print('До:           ', b)
b[1] = 99
print('После:        ', b)
print('Без изменений:', a)

>>>
До:            ['e', 'f', 'g', 'h']
После:         ['e', 99, 'g', 'h']
Без изменений: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

При присвоении срезу нового значения оно заменяет собой элементы исходного списка, попадающие в указанный диапазон. В отличие от выполнения операции присваивания в кортежах (например: `a, b = c[:2]`), операнды не обязаны иметь одинаковую длину. Значения, занимающие позиции до и после границ диапазона среза, сохраняются. Список будет расти или сокращаться в соответствии с присваиваемыми значениями.

```
print('До:    ', a)
a[2:7] = [99, 22, 14]
print('После: ', a)

>>>
До:    ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
После: ['a', 'b', 99, 22, 14, 'h']
```

Если при извлечении среза опустить начальный и конечный индексы, то результат будет представлять собой копию исходного списка.

```
b = a[:]
assert b == a and b is not a
```

Если операция присваивания выполняется без указания начального и конечного индексов среза, то все содержимое списка заменяется (без создания нового объекта списка) последовательностью, указанной справа от оператора присваивания.

```
b = a
print('До:    ', a)
a[:] = [101, 102, 103]
assert a is b      # Все тот же объект списка
print('После: ', a) # Теперь список имеет другое содержимое

>>>
До:    ['a', 'b', 99, 22, 14, 'h']
После: [101, 102, 103]
```

Что следует запомнить

- Избегайте многословия: не указывайте `0` в качестве начального индекса и длину последовательности в качестве конечного индекса среза.
- При извлечении срезов допускается использование начального и конечного индексов, выходящих за границы последовательности, что упрощает включение границ в определение среза (например, `a[:20]` или `a[-20:]`).
- При присвоении значений срезам списков последовательность, указанная справа от оператора присваивания, заменяет указанный в срезе диапазон исходной последовательности, даже если их длины не совпадают.

Рекомендация 6. Избегайте совместного использования индексов начала, конца и шага в одном срезе

Кроме базового синтаксиса срезов (см. предыдущую рекомендацию), в языке Python предусмотрен специальный синтаксис: `[start:end:stride]`, с помощью которого можно задавать не только начальный (`start`) и конечный (`end`) индексы, но и индекс шага выборки элементов последовательности (`stride`). Это позволяет выбирать каждый n-й элемент при извлечении среза. Например, использование шага упрощает группирование элементов списка по признаку четности.

```
a = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = a[::2]
evens = a[1::2]
```

```
print(odds)
print(evens)

>>>
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

Проблема в том, что использование этого синтаксиса часто приводит к получению неожиданных результатов или ошибкам. Например, один из обычных трюков в Python — это обращение байтовых строк с помощью срезов с шагом, равным -1.

```
x = b'mongoose'
y = x[::-1]
print(y)

>>>
b'esoognom'
```

Этот прием отлично подходит для байтовых строк и ASCII-символов, но терпит фиаско при попытке его применения к символам Unicode в виде байтовых строк в кодировке UTF-8.

```
w = '謝謝'
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')

>>>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x9d in
↳ position 0: invalid start byte
```

Полезны ли другие индексы шага, кроме -1? Рассмотрим следующие примеры.

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[::2]  # ['a', 'c', 'e', 'g']
a[::-2] # ['h', 'f', 'd', 'b']
```

Здесь запись ::2 означает извлечение каждого второго элемента при отсчете от начала списка, тогда как запись ::-2 означает то же самое, но при отсчете от конца списка.

А что, по вашему мнению, означает запись 2::2? И что вы можете сказать насчет записей -2::-2, -2:2:-2, 2:2:-2?

```
a[2::2]    # ['c', 'e', 'g']
a[-2::-2]  # ['g', 'e', 'c', 'a']
a[-2:2:-2] # ['g', 'e']
a[2:2:-2]  # []
```

Из всего вышесказанного напрашивается вывод, что смысл выражений, в которых используется синтаксис срезов с шагом, может быть крайне запутанным. Три числа, втиснутые в квадратные скобки, создают слишком большую информационную плотность, затрудняющую восприятие смысла выражения. Вследствие этого характер взаимодействия позиционных индексов с индексом шага становится далеко не очевидным, особенно в случае отрицательных значений шага.

Чтобы не создавать себе лишних проблем, избегайте использования индекса шага совместно с начальным и конечным индексами. Если без этого никак нельзя обойтись, организуйте код таким образом, чтобы начальный и конечный индексы можно было опустить, а индекс шага принимал лишь положительные значения. Если же вам все-таки приходится использовать все три индекса, то постарайтесь разбить операцию присваивания на две отдельных операции, исключающие одновременное использование трех индексов.

```
b = a[::2]  # ['a', 'c', 'e', 'g']
c = b[1:-1] # ['c', 'e']
```

В результате извлечения среза последовательности и последующего извлечения среза с заданным шагом создается дополнительная *поверхностная* (или *мелкая*) копия данных. Целью первой операции является минимизация размера результирующего среза. Если организовать такую двухэтапную обработку данных невозможно в силу того, что это приводит к значительному увеличению времени выполнения программы или недопустимому расходу памяти, то посмотрите, нельзя ли воспользоваться методом `islice()`, входящим в состав встроенного модуля `itertools` (раздел "Рекомендация 46. "Используйте встроенные алгоритмы и структуры данных"), который не позволяет использовать отрицательные значения для индексов начала, конца и шага среза.

Что следует запомнить

- Указание значения индекса шага одновременно с указанием значений начального и конечного индексов среза может приводить к созданию чрезвычайно запутанного кода.
- Старайтесь использовать в срезах положительные значения индекса шага без указания значений начального или конечного индекса. По возможности следует избегать использования отрицательных значений шага.
- Избегайте использования индекса шага совместно с начальным и конечным индексами в одном срезе. Если же это действительно необходимо, разбейте операцию присваивания на две отдель-

ные операции (чтобы разнести индексы) или используйте метод `islice()` из встроенного модуля `itertools`.

Рекомендация 7. Используйте генераторы списков вместо функций `map()` и `filter()`

Язык Python предлагает компактный синтаксис для получения одного списка из другого. Соответствующее выражение носит название *генератор списков*[2] (list comprehension). Предположим, требуется вычислить квадрат каждого числа, входящего в список. Это можно сделать, предоставив выражение, которое должно вычисляться, и список, к которому будет применяться итерационный цикл.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x**2 for x in a]
print(squares)

>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Если только речь не идет о функциях с одним аргументом, генераторы списков в простых случаях выглядят более понятными по сравнению со встроенной функцией `map()`. Последняя требует создания *лямбда-функции*, определяющей вычислительную часть работы, что вносит дополнительные визуальные помехи.

```
squares = map(lambda x: x ** 2, a)
```

В отличие от функции `map()` генераторы списков позволяют легко фильтровать элементы входного списка, удаляя из результата соответствующие выходные данные. Предположим, например, что требуется вычислить лишь квадраты четных чисел. Ниже это достигается за счет добавления условного выражения в генератор списка.

---

[2] Используемый в данном переводе термин *генератор списков* не совсем адекватен, поскольку в Python понятие *генератор* означает объект, возвращающий по требованию по одному элементу коллекции за один раз, а не целую коллекцию. Более того, в Python существуют *выражения-генераторы*, которые могут выступать в качестве подлинных генераторов списков (а также словарей и множеств) в только что описанном смысле. В связи с этим обсуждаются другие варианты перевода термина, например *списковое включение* (`https://ru.wikipedia.org/wiki/Списковое_включение`), однако в настоящее время преобладающим в русскоязычной литературе является термин *генератор списков*. С учетом сделанных оговорок использование именно этого варианта в данном переводе не должно вызывать никаких недоразумений. — *Примеч. ред.*

```
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)

>>>
[4, 16, 36, 64, 100]
```

Тот же результат можно получить, используя встроенную функцию `filter()` вместе с функцией `map()`, однако читать такой код гораздо труднее.

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

Словари и множества имеют собственные эквиваленты генераторов. Это упрощает создание производных структур данных при написании алгоритмов.

```
chile_ranks = {'ghost': 1, 'habanero': 2, 'cayenne': 3}
rank_dict = {rank: name for name, rank in chile_ranks.items()}
chile_len_set = {len(name) for name in rank_dict.values()}
print(rank_dict)
print(chile_len_set)

>>>
{1: 'ghost', 2: 'habanero', 3: 'cayenne'}
{8, 5, 7}
```

Что следует запомнить

- Генераторы списков выглядят гораздо понятнее, чем код, в котором используются встроенные функции `map()` и `filter()`, поскольку не требуют использования дополнительных лямбда-выражений.
- Генераторы списков позволяют легко игнорировать ненужные элементы входного списка, в то время как в случае использования функции `map()` для обеспечения такого поведения требуется привлекать функцию `filter()`.
- В Python аналогичные генераторы имеются также для словарей и множеств.

Рекомендация 8. Избегайте использования более двух выражений в генераторах списков

Помимо базового варианта использования (см. предыдущую рекомендацию), генераторы списков поддерживают многоуровневое вложение циклов. Предположим, например, что вы хотите упростить матри-

цу (список, содержащий другие списки), представив ее в виде простого списка элементов. Ниже это сделано с помощью генератора списков, включающего два цикла for. Эти циклы выполняются в порядке их следования слева направо.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)

>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Этот пример прост, легко читается и содержит приемлемое количество вложенных циклов. Другим разумным применением вложенных циклов является репликация двухуровневой структуры входного списка. Предположим, например, что вам необходимо возвести в квадрат каждый элемент матрицы, сохранив ее двумерную структуру. Ниже приведено используемое для этого выражение, которое содержит чуть больше визуальных помех в виде дополнительных символов [], но все равно легко читается.

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)

>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

Если бы это выражение включало еще один цикл, то длина генератора списка возросла бы настолько, что его пришлось бы разбить на несколько строк.

```
my lists = [
    [[1, 2, 3], [4, 5, 6]],
    # ...
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

В таком виде многострочный генератор списка не намного короче своей альтернативной версии. Ниже тот же результат достигается с помощью обычных операторов цикла. В этой версии выделение отступами делает циклы более понятными, чем генератор списка.

```
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

Кроме того, генераторы списков обеспечивают поддержку нескольких условных операторов `if`. Несколько условий на одном и том же уровне цикла неявно имеют смысл выражения с логическими операторами `И`. Предположим, например, что вы хотите отфильтровать список, выбрав из него только четные числа, значения которых больше `4`. Следующие два генератора списков эквивалентны.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

Условия можно определять на каждом уровне цикла вслед за выражением `for`. Предположим, например, что вы хотите отфильтровать элементы матрицы таким образом, чтобы оставить только те из них, которые делятся на три и находятся в строках, сумма элементов которых не меньше `10`. Это можно реализовать с помощью генератора списка, но код, получаемый при таком подходе, очень трудно читается.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)

>>>
[[6], [9]]
```

Хоть этот пример немного надуман, на практике вам могут встречаться ситуации, в которых подобные выражения будут казаться вполне уместными. Я настоятельно рекомендую избегать использования генераторов списков, похожих на этот. Другим людям будет очень непросто разобраться в результирующем коде. Экономия нескольких строк кода не стоит тех трудностей, к которым это может привести впоследствии.

В качестве мнемонического правила руководствуйтесь тем, что следует избегать использования более чем двух выражений в одном генераторе списка. Это могут быть два условия, два цикла или одно условие и один цикл. Как только код становится более сложным, чем в представленном примере, используйте обычные операторы `if` и `for` и пишите вспомогательные функции (раздел "Рекомендация 16. Не упускайте возможность использовать генераторы вместо возврата списков").

Что следует запомнить

- ✦ Генераторы списков поддерживают вложенные циклы и несколько условий на один уровень цикла.

- Генераторы списков с более чем двумя выражениями очень трудны для восприятия, и их использования следует избегать.

Рекомендация 9. По возможности используйте выражения-генераторы вместо генераторов длинных списков

Возможные проблемы с использованием генераторов списков (см. раздел "Рекомендация 7. Используйте генераторы списков вместо функций `map()` и `filter()`") связаны с тем, что они создают сразу весь новый список целиком. Это вполне допустимо для входных данных небольшого объема, но в случае больших объемов данных это может привести к нехватке памяти и краху программы.

Предположим, например, что вы хотите прочитать файл и возвратить количество символов, содержащихся в каждой строке. В процессе выполнения этой операции генератору списка потребуется хранить в памяти длину каждой строки файла. Если вы имеете дело с файлом огромного размера или с сетевым сокетом, через который нескончаемым потоком поступают все новые и новые данные, применение генераторов списков становится проблематичным. Использованный ниже способ годится лишь для обработки небольших объемов входных данных.

```
value = [len(x) for x in open('my_file.txt')]
print(value)

>>>
[100, 57, 15, 1, 12, 75, 5, 86, 89, 11]
```

В качестве решения, позволяющего преодолеть указанные проблемы, Python предлагает *выражения-генераторы*, представляющие собой обобщение генераторов списков. Выражения-генераторы не загружают в память выходную последовательность целиком. Вместо этого они вычисляют итератор, который по требованию возвращает из выражения по одному элементу за один раз.

Выражение-генератор внешне отличается от генератора списка тем, что вместо квадратных скобок в нем используются круглые. Ниже я использую выражение-генератор, эквивалентное приведенному выше коду. В результате мы получаем итератор, который изначально не выполняет никаких действий.

```
it = (len(x) for x in open('my_file.txt'))
print(it)
```

```
>>>
<generator object <genexpr> at 0x101b81480>
```

Возвращенный итератор обеспечивает получение очередной порции вывода из выражения-генератора по мере необходимости, продвигаясь на один шаг за один раз (с помощью встроенной функции `next()`). Ваш код может обрабатывать выходные данные выражения-генератора в приемлемом для вас объеме без риска переполнения памяти.

```
print(next(it))
print(next(it))

>>>
100
57
```

Еще одним большим преимуществом выражений-генераторов является то, что их можно объединять. Ниже итератор, возвращенный выражением-генератором, используется в качестве источника входных данных для другого выражения-генератора.

```
roots = ((x, x**0.5) for x in it)
```

Каждый раз, когда мы продвигаем этот итератор, он будет продвигать внутренний итератор, тем самым создавая эффект домино и инициируя процессы выполнения циклов, вычисления условных выражений и перемещения входных и выходных данных.

```
print(next(roots))

>>>
(15, 3.872983346207417)
```

Генераторы, объединенные в цепочки наподобие этой, выполняются в Python очень быстро. Если вы ищете способ, с помощью которого могли бы организовать обработку массивных потоков данных, то лучшего средства для выполнения этой работы, чем выражения-генераторы, вам не найти. Единственный неприятный момент состоит в том, что итераторы, возвращаемые выражениями-генераторами, запоминают состояние, и поэтому приходится следить за тем, чтобы они использовались не более чем однократно (раздел "Рекомендация 17. Не забывайте о мерах предосторожности при итерировании аргументов").

Что следует запомнить

- ✦ В случае больших объемов входных данных использование генераторов списков чревато проблемами, обусловленными интенсивным потреблением памяти.

- Выражения-генераторы позволяют избежать проблем, связанных с потреблением памяти, возвращая по одной порции выходных данных за один раз, как итераторы.
- Выражения-генераторы можно объединять в цепочки, передавая итератор из одного выражения-генератора в подвыражение `for` другого.
- Объединенные в цепочки выражения-генераторы выполняются очень быстро.

Рекомендация 10. По возможности используйте функцию `enumerate()` вместо функции `range()`

Встроенную функцию `range()` удобно использовать в циклах, выполняющих итерации по множеству целых чисел.

```
random_bits = 0
for i in range(64):
    if randint(0, 1):
        random_bits |= 1 << i
```

Имея структуру данных для итерирования, такую как список строк, вы сможете организовать цикл непосредственно для этой последовательности.

```
flavor_list = ['ваниль', 'шоколад', 'пекан', 'земляника']
for flavor in flavor_list:
    print('%s имеет чудесный вкус' % flavor)
```

При итерировании по списку вам часто надо будет знать также индекс текущего элемента. Предположим, например, что вы хотите вывести рейтинг ваших любимых сортов мороженого. Один из возможных способов сделать это заключается в использовании функции `range()`.

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print('%d: %s' % (i + 1, flavor))
```

По сравнению с другими примерами итерирования по `flavor_list` или `range()` этот код выглядит топорно. Вам нужно определить длину списка, проиндексировать массив и т.д. Все это затрудняет чтение кода.

Python предоставляет встроенную функцию `enumerate()`, которая позволяет справиться с этой ситуацией. Функция `enumerate()` обертывает любой итератор *отложенным генератором* (lazy generator), который генерирует пары значений: индекс цикла и следующее значение, полу-

чаемое от итератора. Результирующий код становится гораздо более понятным.

```
for i, flavor in enumerate(flavor_list):
print('%d: %s' % (i + 1, flavor))

>>>
1: ваниль
2: шоколад
3: пекан
4: земляника
```

Этот код можно сократить еще больше, указав значение, с которого должна начинать отсчет функция `enumerate()` (в данном случае — `1`).

```
for i, flavor in enumerate(flavor_list, 1):
    print('%d: %s' % (i, flavor))
```

Что следует запомнить

- Функция `enumerate()` обеспечивает лаконичный синтаксис для организации цикла по итератору и получения индекса каждого из элементов, предоставляемых итератором.
- Старайтесь использовать преимущественно функцию `enumerate()` вместо организации цикла по диапазону и индексирования последовательности.
- Функции `enumerate()` можно предоставить второй параметр, задающий начало отсчета (по умолчанию — `0`).

Рекомендация 11. Используйте функцию `zip()` для параллельной обработки итераторов

В Python часто возникают ситуации, когда приходится иметь дело с множеством списков родственных объектов. Генераторы списков упрощают получение производных списков из исходного с помощью выражений (см. раздел "Рекомендация 7. Используйте генераторы списков вместо функций `map()` и `filter()`").

```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]
```

Элементы обоих списков — производного и исходного — связаны своими индексами. Можно организовать параллельное итерирование по обоим спискам, выполняя итерации только по исходному списку `names`.

```
longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count

print(longest_name)

>>>
Cecilia
```

Проблема в том, что вся эта конструкция цикла в целом выглядит довольно громоздко. Индексы в `names` и `letters` затрудняют чтение кода. Индексирование массивов с помощью индекса цикла `i` происходит дважды. Использование функции `enumerate()` (см. предыдущую рекомендацию) немного улучшает ситуацию, но она все еще остается неидеальной.

```
for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = name
        max_letters = count
```

Можно сделать этот код более понятным, воспользовавшись предоставляемой в Python встроенной функцией `zip()`. В Python 3 `zip()` обертывает два или более итератора отложенным генератором. Этот `zip`-генератор возвращает кортежи, которые содержат следующие значения каждого итератора, что дает гораздо более понятный код, чем индексирование нескольких списков.

```
for name, count in zip(names, letters):
    if count > max_letters:
        longest_name = name
        max_letters = count
```

Со встроенной функцией `zip()` связаны две проблемы.

Первая из них — это то, что в Python 2 функция `zip()` не является генератором; она до конца отрабатывает предоставленные ей итераторы и возвращает список всех создаваемых ею кортежей. Следствием этого становится большой перерасход памяти, чреватый возможностью краха программы. Если в Python 2 возникает необходимость в обработке очень крупных итераторов с помощью функции `zip()`, то гораздо лучше использовать вместо нее функцию `izip()` из встроенного модуля

itertools (раздел "Рекомендация 46. Используйте встроенные алгоритмы и структуры данных").

Суть второй проблемы в том, что функция zip() ведет себя не совсем обычно, если входные итераторы имеют разную длину. Предположим, например, что вы добавляете еще одно имя в указанный выше список, но забываете обновить счетчик letters. Применение функции zip() даст результат, отличающийся от ожидаемого.

```
names.append('Rosalind')
for name, count in zip(names, letters):
    print(name)

>>>
Cecilia
Lise
Marie
```

Новый элемент для 'Rosalind' здесь отсутствует. Это произошло просто потому, что так работает функция zip(). Она возвращает кортежи до тех пор, пока обернутый ею итератор не будет исчерпан. Такой подход отлично работает, если вам известно, что все итераторы имеют одну и ту же длину, как это часто имеет место в случае производных списков, создаваемых генераторами. Во многих других случаях вы можете столкнуться с неожиданным и нежелательным поведением этой функции, когда она урезает вывод. Если у вас нет уверенности в том, что длина списков, которые вы планируете обработать с помощью функции zip(), одинакова, попробуйте использовать вместе нее функции zip_longest() (izip_longest() в Python 2) из встроенного модуля itertools.

Что следует запомнить

- Для параллельного итерирования по нескольким итераторам можно использовать встроенную функцию zip().
- В Python 3 функция zip() — это встроенный генератор, который по требованию возвращает кортежи. В Python 2 эта функция возвращает сразу весь результат в виде списка кортежей.
- Функция zip() без всякого уведомления урезает вывод, если предоставленные ей итераторы имеют разную длину.
- Функция zip_longest() из встроенного модуля itertools позволяет итерировать параллельно по нескольким итераторам, независимо от их длины (раздел "Рекомендация 46. Используйте встроенные алгоритмы и структуры данных").

Рекомендация 12. Избегайте использования блоков else после циклов for и while

В языке Python циклы предлагают дополнительную возможность, недоступную в большинстве других языков программирования: сразу после повторно выполняемого внутреннего блока цикла можно поместить блок else.

```
for i in range(3):
    print('Цикл %d' % i)
else:
    print('Блок Else!')

>>>
Цикл 0
Цикл 1
Цикл 2
Блок Else!
```

Как это ни удивительно, но блоки else выполняются сразу же после того, как заканчивается выполнение цикла. Тогда почему используется именно слово "else"? Почему не "and"? В инструкции if/else часть else означает: "Выполнить этот блок, если предыдущий блок не выполнялся". В инструкции try/except часть except имеет сходный смысл: "Выполнить этот блок, если попытка выполнения предыдущего блока оказалась неудачной".

Точно такому же шаблону следует и часть else в инструкции try/except/else (раздел "Рекомендация 13. Старайтесь использовать возможности каждого из блоков конструкции try/except/else/finally"), поскольку она означает: "Выполнить этот блок, если предыдущий блок не завершился неудачно". Инструкция try/finally также интуитивно понятна, поскольку она означает: "Всегда выполнять этот блок после попытки выполнения предыдущего блока".

С учетом всех этих случаев использования ключевых слов else, except и finally в Python программист-новичок мог бы предположить, что слово else в инструкции for/else означает: "Выполнить этот блок, если цикл не был успешно выполнен". Действительный смысл прямо противоположен. Использование инструкции break в цикле приведет к фактическому пропуску блока else.

```
for i in range(3):
    print('Цикл %d' % i)
    if i == 1:
        break
    else:
```

```
        print('Блок Else!')

>>>
Цикл 0
Цикл 1
```

Еще одним сюрпризом является немедленное выполнение блока `else`, если цикл применяется к пустому списку.

```
for x in []:
    print('Этот блок никогда не будет выполнен')
else:
    print('Блок For Else!')

>>>
Блок For Else!
```

Блок `else` выполняется и тогда, когда условие цикла `while` изначально возвращает значение `false`.

```
while False:
    print('Этот блок никогда не будет выполнен')
else:
    print('Блок While Else!')

>>>
Блок While Else!
```

Основанием для организации такого поведения конструкции `if/else` послужило то, что выполнение блоков `else` после циклов полезно в тех случаях, когда циклы используются для поиска каких-то объектов. Предположим, требуется определить, являются ли два заданных числа взаимно простыми (т.е. является ли число 1 их единственным общим делителем). Ниже приведен код, в котором выполняются итерации по всем возможным делителям тестируемых чисел. Цикл завершается после перебора всех возможных вариантов. Блок `else` выполняется в том случае, когда числа являются взаимно простыми, поскольку цикл не доходит до оператора `break`.

```
a = 4
b = 9
for i in range(2, min(a, b) + 1):
    print('Тестируется', i)
    if a % i == 0 and b % i == 0:
        print('Не взаимно простые')
        break
else:
    print('Взаимно простые')
```

```
>>>
Тестируется 2
Тестируется 3
Тестируется 4
Взаимно простые
```

Никогда не используйте такой подход на практике. Вместо этого лучше написать вспомогательную функцию, которая выполнит все вычисления. Получили распространение два стиля написания подобных функций.

Суть первого подхода заключается в преждевременном выходе из цикла и возврате из функции, как только обнаруживается, что условие выполняется. В случае нормального завершения цикла возвращается результат, предусмотренный по умолчанию.

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True
```

Второй подход предполагает возврат результата в виде переменной, указывающей на то, был ли найден искомый объект при выполнении цикла. Как только удается найти то, что ищется, осуществляется выход из цикла.

```
def coprime2(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

Оба подхода улучшают читаемость кода. Выразительность кода, обеспечиваемая блоком `else`, не перевешивает тех неудобств, которые он доставит другим людям (да и вам самому), если впоследствии придется просматривать этот код. Простые конструкции типа циклов должны быть самоочевидными в Python. А использования блоков `else` после циклов следует вообще избегать.

Что следует запомнить

- В языке Python предусмотрен специальный синтаксис, предусматривающий блоки `else`, расположенные сразу за внутренними блоками циклов `for` и `while`.

- Блок else, следующий за циклом, выполняется лишь в том случае, если в теле цикла не выполнялся оператор break.
- Избегайте использования блоков else после циклов, поскольку их поведение не является интуитивно понятным и может вносить путаницу.

Рекомендация 13. Старайтесь использовать возможности каждого из блоков конструкции try/except/else/finally

Ваше решение предпринять какие-либо действия по обработке исключений в Python может быть реализовано в четырех структурно выделенных участках кода. Речь идет о блоках try, except, else и finally. Каждый из них выполняет строго определенные функции в рамках одной сложной инструкции, и в зависимости от ситуации вы можете использовать их в различных комбинациях (раздел "Рекомендация 51. Изолируйте вызывающий код от API, определяя базовое исключение Exception").

Блоки finally

Используйте инструкцию try/finally, если хотите предоставить исключению возможность распространяться, но при этом намерены освободить ресурсы, даже если возникло исключение. Одно из обычных применений инструкции try/finally — надежное закрытие дескрипторов файлов (раздел "Рекомендация 43. Обеспечивайте возможность повторного использования блоков try/finally с помощью инструкций contextlib и with").

```
handle = open('random_data.txt') # Может генерировать
                                 # исключение IOError
try:
    data = handle.read()         # Может генерировать
                                 # исключение UnicodeDecodeError
finally:
    handle.close()               # Всегда выполняется после
                                 # блока try:
```

Любое исключение, сгенерированное методом read(), всегда будет распространяться до уровня вызывающего кода, но при этом гарантируется, что будет выполнен также метод close() объекта handle. Метод open() следует вызывать вне блока try, поскольку в случае исключений, которые могут возникать при открытии файла (например, IOError, если такого файла не существует), блок finally не должен выполняться.

Блоки else

Используйте инструкцию try/except/else для того, чтобы четко указать, какие исключения будут обрабатываться вашим кодом, а какие — распространяться. Блок else будет выполняться только в том случае, если в блоке try не возникли исключения. Блок else позволяет минимизировать объем кода блока try и улучшить его удобочитаемость. Предположим, например, что вам необходимо загрузить данные словаря в формате JSON из строки и возвратить значения содержащихся в ней ключей.

```
def load_json_key(data, key):
    try:
        result_dict = json.loads(data) # Может генерировать
                                       # исключение ValueError
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]        # Может генерировать
                                       # исключение KeyError
```

В случае предоставления данных, не соответствующих формату JSON, при попытке их декодирования с помощью метода json.loads() возникнет исключение ValueError, которое перехватывается и обрабатывается блоком except. В случае успешного декодирования данных в блоке else будет выполняться поиск ключа. Если при поиске ключа возникнут исключения, то они будут распространяться до уровня вызывающего кода, поскольку генерируются вне блока try. Часть else гарантирует визуальное отделение кода, следующего за конструкцией try/except, от блока except. Благодаря этому процесс распространения исключений становится более понятным.

Комбинация всех блоков

Используйте полную конструкцию try/exception/else/finally в тех случаях, когда хотите выполнить все задачи в рамках одной сложной инструкции. Предположим, например, что вы хотите прочитать из файла описание работы, которую предстоит сделать, а затем обработать файл и обновить его на месте. В данном случае для чтения файла и обработки данных используется блок try. Блок except используется для обработки тех исключений, возникновения которых вы ожидаете в блоке try. Блок else используется для обновления файла на месте, а кроме того, разрешает распространение родственных исключений. Блок finally освобождает дескриптор файла.

```
UNDEFINED = object()

def divide_json(path):
    handle = open(path, 'r+')  # Может генерировать исключение
                               # IOError
    try:
        data = handle.read()   # Может генерировать исключение
                               # UnicodeDecodeError
        op = json.loads(data)  # Может генерировать исключение
                               # ValueError
        value = (
            op['numerator'] /
            op['denominator']) # Может генерировать исключение
                               # ZeroDivisionError
    except ZeroDivisionError as e:
        return UNDEFINED
    else:
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)
        handle.write(result)   # Может генерировать исключение
                               # IOError
        return value
    finally:
        handle.close()         # Всегда выполняется
```

Такая конфигурация особенно полезна, поскольку все блоки работают интуитивно понятным образом. Например, если исключение возникает в блоке `else` при перезаписи результирующих данных, то блок `finally` будет в любом случае выполнен и закроет дескриптор файла.

Что следует запомнить

- Сложная инструкция `try/finally` позволяет выполнять код, освобождающий ресурсы, независимо от того, возникали ли исключения в блоке `try`.
- Блок `else` позволяет минимизировать объем кода в блоках `try` и визуально отделить участки кода, выполняющиеся в отсутствие исключений, от блоков `try/except`.
- Блок `else` может использоваться для выполнения дополнительных действий после успешного выполнения блока `try`, но перед выполнением действий по освобождению ресурсов в блоке `finally`.

2 Функции

Первый из организационных инструментов, который программисты используют в Python, — это *функции*. Как и в других языках программирования, функции позволяют разбивать программы на более простые части меньшего размера. Они улучшают читаемость кода и делают его более понятным, а также обеспечивают повторное использование и рефакторинг кода.

Функции в языке Python обладают рядом дополнительных возможностей, облегчающих жизнь программиста. Некоторые из них имеют аналоги в других языках программирования, но многие уникальны для Python. Эти дополнительные возможности могут прояснять и делать более очевидным предназначение функции, а также способствуют устранению визуального шума и более отчетливой демонстрации намерений вызывающего объекта. Кроме того, они могут значительно снижать вероятность возникновения малозаметных ошибок, обнаруживать которые бывает не так-то легко.

Рекомендация 14. Использование исключений предпочтительнее возврата значения None

При написании вспомогательных функций программисты на языке Python часто придают специальный смысл возвращаемому значению None. В некоторых ситуациях такое решение можно считать вполне обоснованным. Предположим, например, что вам нужна вспомогательная функция, которая делит одно число на другое. При делении на нуль возврат значения None представляется естественным, поскольку в данном случае результат не определен.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Возврат значения None может соответствующим образом интерпретироваться кодом, в котором используется данная функция.

```
result = divide(x, y)
if result is None:
    print('Недопустимые входные данные')
```

А что произойдет, если числитель равен нулю? При ненулевом знаменателе результат будет равен 0. Проблемы могут возникнуть, если по случайному недосмотру в качестве индикатора ошибок вы используете не значение None, а любое значение, эквивалентное значению False (похожая ситуация была описана в разделе "Рекомендация 4. Заменяйте сложные выражения вспомогательными функциями").

```
x, y = 0, 5
result = divide(x, y)
if not result:
    print('Недопустимые входные данные') # Так неправильно!
```

Это распространенная ошибка в коде на Python в тех случаях, когда None имеет специальный смысл. Именно поэтому возврат функцией значения None может приводить к ошибкам. Для уменьшения вероятности подобных ошибок существуют два метода.

Суть первого способа заключается в том, чтобы представлять возвращаемое значение в виде кортежа из двух элементов. Первая часть кортежа указывает на успешность или неуспешность выполнения операции, а вторая — это фактически вычисленный результат.

```
def divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None
```

Код, вызывающий данную функцию, должен распаковать кортеж. По этой причине он вынужден принимать к рассмотрению часть кортежа, указывающую на характер завершения операции, а не просто получать результат деления.

```
success, result = divide(x, y)
if not success:
    print('Недопустимые входные данные')
```

Проблема в том, что вызывающий код может легко игнорировать первую часть кортежа, указав в качестве имени переменной символ подчеркивания, чему, по принятому в Python соглашению, соответствует неиспользуемая переменная. На первый взгляд, в этом коде нет ни-

чего плохого. Однако он грешит теми же недостатками, что и возврат значения None.

```
_, result = divide(x, y)
if not result:
    print('Недопустимые входные данные')
```

Второй и более эффективный способ снижения вероятности появления ошибок подобного рода состоит в том, чтобы вообще отказаться от возврата значения None в качестве индикатора ошибки. Вместо этого сгенерируйте исключение и передайте его на обработку вызывающему коду. Ниже мы преобразуем исключение ZeroDivisionError в исключение ValueError, которое указывает вызывающему коду на недопустимость входных значений.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Недопустимые входные данные') from e
```

Теперь вызывающий код должен обработать исключение, возникшее из-за некорректности входных данных (это поведение должно быть задокументировано, как описывается в разделе "Рекомендация 49. Снабжайте строками документирования каждую функцию, класс и модуль"). Вызывающему коду больше не нужно проверять значение, возвращаемое функцией. Если функция не сгенерировала исключение, то это означает, что она возвращает нормальное значение. Если же возникает исключение, то исход его обработки совершенно понятен.

```
x, y = 5, 2
try:
    result = divide(x, y)
except ValueError:
    print('Недопустимые входные данные')
else:
    print('Результат: %.1f' % result)

>>>
Результат: 2.5
```

Что следует запомнить

- Использование функций, возвращающих значение None для индикации особых ситуаций, может приводить к ошибкам, поскольку None и другие значения (например, 0 или пустая строка) в условных выражениях эквивалентны значению False.

- Сигнализируйте о возникновении особых ситуаций с помощью исключений, а не путем возврата значения None. При этом ожидается, что документированные исключения будут корректно обработаны вызывающим кодом.

Рекомендация 15. Знайте, как замыкания взаимодействуют с областью видимости переменных

Предположим, вам необходимо отсортировать список чисел, сделав при этом приоритетной одну группу чисел, которая должна следовать первой. Такой шаблон может быть полезным при выводе журнала сообщений, когда вы хотите, чтобы важные сообщения или информация об исключениях отображались в первую очередь.

Обычный подход к решению этой задачи предполагает передачу вспомогательной функции в качестве аргумента key метода sort() списка. Возвращаемые вспомогательной функцией значения используются для сортировки списка. Вспомогательная функция может проверить, относится ли данный элемент к важной группе, и соответствующим образом изменить ключ сортировки.

```
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
        return (1, x)
    values.sort(key=helper)
```

Эта функция работает для простых входных данных.

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)

>>>
[2, 3, 5, 7, 1, 4, 6, 8]
```

Есть три причины, по которым эта функция работает так, как ожидается.

- Язык Python поддерживает *замыкания* — функции, ссылающиеся на переменные из области видимости, в которой они были определены. Вот почему функция helper() способна получать доступ к аргументу group функции sort_priority().

- В языке Python функции являются *объектами первого класса*, т.е. допускается ссылаться непосредственно на них, присваивать их переменным, передавать в качестве аргументов другим функциям, сравнивать в выражениях и инструкциях `if` и т.п. Именно поэтому метод `sort()` может принимать функцию-замыкание в качестве аргумента `key`.
- В языке Python предусмотрены специальные правила для сравнения кортежей. Сначала сравниваются элементы с индексом `0`, затем с индексом `1`, `2`, `3` и т.д. Вот почему возврат значения из замыкания `helper()` позволяет установить для порядка сортировки две различные группы.

Было бы неплохо, если бы эта функция возвращала значение, позволяющее судить о том, встретились ли вообще высокоприоритетные элементы, чтобы код пользовательского интерфейса мог выполнять соответствующие действия. Может создаться впечатление, что с добавлением такого поведения не должны быть связаны никакие трудности. У нас уже есть функция-замыкание, позволяющая определить, к какой именно группе относится каждое из чисел. Почему бы также не использовать функцию-замыкание для изменения состояния флага, когда встречается элемент с высоким приоритетом? Тогда функция могла бы возвращать значение флага после того, как оно было изменено замыканием.

Ниже делается попытка реализовать это способом, который, казалось бы, очевиден.

```
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True # На первый взгляд, все просто
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

Эту функцию можно выполнить с теми же входными данными, что и до этого.

```
found = sort_priority2(numbers, group)
print('Найдено:', found)
print(numbers)

>>>
Найдено: False
[2, 3, 5, 7, 1, 4, 6, 8]
```

Отсортированные результаты корректны, чего нельзя сказать о значении флага found. Элементы кортежа group были определенно обнаружены в списке numbers, и тем не менее функция вернула значение False. Как это могло произойти?

Когда вы ссылаетесь на переменную в выражении, Python разрешает эту ссылку, просматривая области видимости в указанной ниже последовательности.

1. Область видимости текущей функции.
2. Охватывающие области видимости (например, область видимости функции, содержащей текущую функцию).
3. Область видимости модуля, в котором содержится код (*глобальная область видимости*).
4. Область видимости встроенных функций (таких, как len() или str()).

Если ни в одной из перечисленных областей имя переменной, на которую имеется ссылка, не определено, то генерируется исключение NameError.

Присваивание значения переменной работает иначе. Если переменная уже определена в текущей области видимости, то она просто принимает новое значение. Если же в текущей области видимости переменной с таким именем не существует, Python трактует присваивание значения как определение переменной. Областью видимости вновь определенной переменной является функция, в которой осуществляется данная операция присваивания.

Такое поведение операции присваивания объясняет, почему функция sort_priority2() возвращает не то значение, которое ожидается. Присваивание переменной found значения True выполняется в замыкании helper(). Такое присваивание трактуется как определение новой переменной в замыкании helper(), а не как присваивание в области видимости sort_priority2.

```
def sort_priority2(numbers, group):
    found = False         # Область видимости: 'sort_priority2()'
    def helper(x):
        if x in group:
            found = True  # Область видимости: 'helper()' - Плохо!
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

Подобные проблемы часто называют *ошибками области видимости*, поскольку у новичков они вызывают удивление. Однако такое поведе-

ние было создано намеренно. Оно не позволяет локальным переменным функций засорять модуль, в котором содержатся эти функции. В противном случае любая операция присваивания внутри функции заполняла бы глобальную область видимости модуля "мусором". Это не только создавало бы визуальные помехи, но и приводило бы к скрытым ошибкам, вызванным взаимодействием результирующих глобальных переменных.

Извлечение данных из замыкания

В Python 3 предусмотрен специальный синтаксис, позволяющий извлекать данные из замыкания. Для того чтобы указать, что переменная, которой присваивается значение, должна быть доступна за пределами текущей области видимости, используется ключевое слово `nonlocal`. Единственная граница, которую не может пересечь такая нелокальная переменная, — это пределы области видимости модуля (данное ограничение предотвращает замусоривание глобальной области видимости).

Ниже приведено определение той же функции, в котором используется нелокальная переменная.

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

Ключевое слово `nonlocal` совершенно отчетливо показывает, когда данные из замыкания присваиваются переменной, принадлежащей другой области видимости. Оно дополняет ключевое слово `global`, которое указывает на то, что значение присваивается переменной, принадлежащей непосредственно области видимости модуля.

Однако, как и в случае глобальных переменных, я бы предостерег вас от использования ключевого слова `nonlocal` где-нибудь еще, кроме самых простых функций, поскольку отслеживать такие переменные нелегко. Особенно это относится к длинным функциям, в которых объявления переменных, используемых как нелокальные, и операции присваивания им значений разделены большим количеством строк кода.

Как только вы начинаете замечать, что использовать нелокальные переменные становится все сложнее, поместите интересующее вас состояние во вспомогательный класс. Ниже определен класс, позволяю-

щий получить тот же результат, который перед этим был получен с помощью нелокальной переменной. Размер кода при этом немного увеличивается, но читать его легче (более подробно о специальном методе `__call__()` говорится в разделе "Рекомендация 23. Принимайте функции вместо классов в случае простых интерфейсов").

```
class Sorter(object):
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

Области видимости в Python 2

К сожалению, ключевое слово `nonlocal` не поддерживается в Python 2. Для получения аналогичного поведения необходимо использовать обходной путь, воспользовавшись особенностями правил Python, которые регламентируют определение областей видимости. Этот подход выглядит не очень элегантно, но является общепринятой идиомой в Python.

```
# Python 2
def sort_priority(numbers, group):
    found = [False]
    def helper(x):
        if x in group:
            found[0] = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found[0]
```

Как уже отмечалось, чтобы определить текущее значение переменной `found`, Python пересечет границу области видимости, в которой присутствует ссылка на эту переменную. Суть трюка состоит в том, что значением `found` является список, а списки — это изменяемый тип данных. Отыскав переменную `found`, замыкание может изменить ее состояние для передачи данных за пределы внутренней области видимости (с помощью присваивания `found[0] = True`).

Описанный подход работает и в тех случаях, когда переменной, которая используется для выхода за пределы области видимости, является словарь, множество или экземпляр вашего класса.

Что следует запомнить

- Функции-замыкания могут ссылаться на локальные переменные из любой области видимости, в которой они были определены.
- По умолчанию замыкания не могут воздействовать на охватывающие их области видимости путем присваивания значений объявленным в них переменным.
- В Python 3 для указания того, что замыкание может изменять переменную в охватывающей его области видимости, используйте ключевое слово `nonlocal`.
- В Python 2 для обхода ограничений, связанных с отсутствием ключевого слова `nonlocal`, используйте изменяемые значения (например, список, включающий одно значение).
- Избегайте использования ключевого слова `nonlocal` для сколь-нибудь сложных функций, кроме простейших случаев.

Рекомендация 16. Не упускайте возможность использовать генераторы вместо возврата списков

В качестве простейшего решения для функций, вырабатывающих последовательность результирующих значений, напрашивается возврат списка элементов. Предположим, вы хотите найти индекс каждого слова в строке. Ниже приведен код, в котором результаты собираются в один список с помощью метода `append()` и возвращаются по завершении работы функции.

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

Для тестовых входных данных эта функция работает так, как и следовало ожидать.

```
address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:3])
>>>
[0, 5, 11]
```

Тем не менее с использованием функции `index_words()` связаны две проблемы.

Одна из них — это то, что код несколько перегружен и содержит визуальный шум. Каждый раз, когда находится очередной результат, вызывается метод `append()`. При этом добавляемое значение (`index + 1`) визуально теряется в выражении вызова метода. Одна строка кода предназначена для создания результата, а вторая — для его возврата. В то время как тело функции включает приблизительно 130 символов (без пробелов), лишь 75 из них отвечают за существенно важную часть кода.

Для написания этой функции существует лучший способ — использование *генератора*. Генераторы — это функции, использующие выражение `yield`. Вызов такой функции еще не означает ее фактического выполнения; вместо этого она сразу же вызывает итератор. С каждым вызовом встроенной функции `nest()` итератор продвигает генератор к его следующему выражению `yield`. Каждое значение, переданное выражению `yield` генератором, итератор возвращает вызывающему коду.

Ниже определена функция генератора, которая обеспечивает получение тех же результатов.

```
def index_words_iter(text):
    if text:
      yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

Читать этот код значительно легче, поскольку из него устранено любое взаимодействие со списком результатов. Вместо этого результаты передаются выражениям `yield`. Итератор, возвращенный вызовом генератора, может быть легко преобразован в список путем передачи его встроенному методу `list()` (более подробно о том, как это работает, см. в разделе "Рекомендация 9. По возможности используйте выражения-генераторы вместо генераторов длинных списков").

```
result = list(index_words_iter(address))
```

Другая проблема, связанная с функцией `index_words()`, заключается в том, что она требует предварительного сохранения всех результатов в списке, прежде чем их можно будет вернуть. При больших объемах входных данных это может привести к перерасходу памяти и краху про-

граммы. В противоположность этому версию функции в виде генератора можно легко адаптировать к входным данным любого объема.

Ниже определен генератор, который принимает входные данные из файла по одной строке за один раз и выдает результаты также по одному слову за один раз. Рабочая память, необходимая для этой функции, ограничена максимальной длиной одной входной строки.

```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
        for letter in line:
            offset += 1
            if letter == ' ':
                yield offset
```

С помощью генератора мы получаем те же результаты.

```
with open('address.txt', 'r') as f:
    it = index_file(f)
    results = islice(it, 0, 3)
    print(list(results))

>>>
[0, 5, 11]
```

Единственным недостатком определения генераторов, подобных этому, является то, что вызывающему коду должно быть известно, что возвращаемые генераторы сохраняют состояние и не могут использоваться повторно (раздел "Рекомендация 17. Не забывайте о мерах предосторожности при итерировании аргументов").

Что следует запомнить

- Использование генераторов обеспечивает создание более понятного кода по сравнению с вариантом, предусматривающим возврат списков накопленных результатов.
- Итератор, возвращенный генератором, создает набор значений, передаваемых выражениям `yield` в теле функции генератора.
- Генераторы могут создавать последовательность выходных результатов для сколь угодно больших объемов входных данных, поскольку в их рабочую память не загружаются целиком все входные или выходные данные.

Рекомендация 17. Не забывайте о мерах предосторожности при итерировании аргументов

Во многих случаях, когда функция принимает список объектов в качестве параметра, возникает необходимость в многократном итерировании по этому списку. Предположим, например, что вы хотите проанализировать количество туристов, приезжающих в штат Техас. Пусть набор данных представляет число туристов, посетивших каждый из городов (в миллионах человек за год). Вам необходимо определить для каждого города долю туристов в процентном отношении к их общему количеству.

Для решения этой задачи нужна нормирующая функция, которая суммировала бы входные значения для определения общего количества туристов на протяжении года. Отношение количества туристов, посетивших определенный город, к общему количеству туристов дает искомый процентный показатель.

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Для работы этой функции ей необходимо предоставить список соответствующих количественных показателей посещаемости городов.

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

Чтобы масштабировать эту задачу, мы должны читать данные из файла, содержащего информацию по всем городам Техаса. Для этого определяем генератор, что даст возможность повторно использовать ту же функцию впоследствии, когда потребуется вычислить посещаемость туристами городов по всему миру, а это будет набор данных намного большего размера (см. предыдущую рекомендацию).

```
def read_visits(data_path):
    with open(data_path) as f:
```

```
        for line in f:
            yield int(line)
```

Неожиданно обнаруживается, что вызов функции `normalize()` не дает никаких результатов.

```
it = read_visits('my_numbers.txt')
percentages = normalize(it)
print(percentages)

>>>
[]
```

Причиной такого поведения послужило то, что итератор выдает свои результаты только один раз. Если вы итерируете по итератору или генератору, который уже сгенерировал исключение `StopIteration`, то получить результаты два раза подряд вам не удастся.

```
it = read_visits('my_numbers.txt')
print(list(it))
print(list(it)) # Итератор уже исчерпан

>>>
[15, 35, 80]
[]
```

Путаницу вносит также то, что вы не получите никаких сообщений об ошибке, если будете итерировать по уже исчерпанному итератору. Циклы `for`, конструктор `list()` и многие другие функции стандартной библиотеки Python ожидают, что исключение `StopIteration` будет сгенерировано в процессе нормальной работы. Эти функции не могут отличить итератор, в котором отсутствуют выходные данные, от итератора, в котором выходные данные были, но уже исчерпаны.

Чтобы разрешить эту проблему, можно явным образом исчерпать входной итератор и сохранить копию всего содержимого в списке. После этого вы сможете итерировать по списковой версии данных сколько угодно раз. Ниже приведена та же функция, но входной итератор в ней предусмотрительно копируется.

```
def normalize_copy(numbers):
    numbers = list(numbers) # Копировать итератор
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Теперь функция корректно работает с возвращаемым значением генератора.

```
it = read_visits('my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

При таком подходе проблемой остается то, что размер копий содержимого входного итератора может быть очень большим. Копирование итератора способно привести к перерасходу памяти и краху программы. Один из способов обойти эту трудность заключается в том, чтобы принимать функцию, которая возвращает новый итератор всякий раз, когда она вызывается.

```
def normalize_func(get_iter):
    total = sum(get_iter()) # Новый итератор
    result = []
    for value in get_iter(): # Новый итератор
        percent = 100 * value / total
        result.append(percent)
    return result
```

Чтобы использовать функцию `normalize_func()`, можно передать ей лямбда-выражение, которое вызывает генератор и каждый раз создает новый итератор.

```
percentages = normalize_func(lambda: read_visits(path))
```

Несмотря на то что это работает, передача лямбда-функции выглядит довольно неуклюже. Лучший способ получения того же результата — предоставить новый контейнерный класс, реализующий *протокол итератора*.

Протокол итератора — это процедура, в соответствии с которой циклы `for` и другие подобные выражения Python выполняют обход содержимого контейнерного типа. Когда Python встречает выражение вида `for x in foo`, он фактически осуществляет вызов `iter(foo)`. В свою очередь, встроенная функция `iter()` вызывает специальный метод `foo.__iter__()`. Метод `__iter__()` должен возвратить объект итератора (который сам реализует специальный метод `__next__()`). Затем цикл `for` повторно вызывает встроенную функцию `next()` для объекта итератора до его исчерпания (после чего генерируется исключение `StopIteration`).

Все это выглядит немного сложно, но с практической точки зрения вы сможете полностью обеспечить это поведение для своих классов, реализуя метод `__iter__()` как генератор. Ниже мы определим итериру-

емый контейнерный класс, который читает содержимое файлов с данными по туризму.

```
class ReadVisits(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)
```

Будучи переданным исходной функции, этот тип данных работает корректно без внесения каких-либо изменений.

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)


>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

Этот код работает, поскольку метод `sum()` в функции `normalize()` будет вызывать метод `ReadVisits.__iter__()` для размещения нового объекта итератора. Чтобы нормировать числовые данные, цикл `for` также будет вызывать функцию `__iter__()` для размещения второго объекта итератора. Продвижение и исчерпание этих итераторов будет происходить независимо, гарантируя, что каждая уникальная итерация будет видеть все значения входных данных. Единственным недостатком такого подхода является то, что он требует многократного чтения входных данных.

Теперь, когда вы знаете, как работают контейнеры наподобие `ReadVisits`, вы сможете писать собственные функции, гарантирующие, что параметры не являются только итераторами. Протокол предписывает, что при передаче итератора встроенной функции `iter()` она возвратит сам итератор. В противоположность этому, когда функции `iter()` передается контейнерный тип, каждый раз будет возвращаться новый объект итератора. Таким образом, можно тестировать входные данные на соответствие такому поведению и генерировать исключение `TypeError` для отбраковки итераторов.

```
def normalize_defensive(numbers):
    if iter(numbers) is iter(numbers): # Итератор -- плохо!
        raise TypeError('Должен предоставляться контейнер')
    total = sum(numbers)
    result = []
```

```
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Все это хорошо, если вы не хотите копировать итератор полного ввода наподобие приведенной выше функции `normalize_copy()`, но вам также необходимо многократное итерирование по входным данным. Эта функция работает так, как ожидается, для входных данных `list` и `ReadVisits`, поскольку они являются контейнерами. Она будет работать для контейнера любого типа, который следует протоколу итератора.

```
visits = [15, 35, 80]
normalize_defensive(visits) # Нет ошибки
visits = ReadVisits(path)
normalize_defensive(visits) # Нет ошибки
```

Эта функция сгенерирует исключение, если входной объект итерируемый, но не является контейнером.

```
it = iter(visits)
normalize_defensive(it)


>>>
TypeError: Должен предоставляться контейнер
```

Что следует запомнить

- Проявляйте осторожность, работая с функциями, которые требуют многократного итерирования по входным аргументам. Если этими аргументами являются итераторы, то можно столкнуться с необычным поведением программы или недостающими значениями.
- Протокол итераторов Python определяет, каким образом контейнеры и итераторы должны взаимодействовать со встроенными функциями `iter()` и `next()`, циклами `for` и другими выражениями подобного рода.
- Вы можете легко определить собственный тип итерируемого контейнера, реализовав метод `__iter__()` как генератор.
- Вы сможете обнаружить, что значение является итератором (а не генератором), если два вызова функции `iter()` для него дают один и тот же результат, по которому далее можно продвигаться с помощью встроенной функции `next()`.

Рекомендация 18. Снижайте визуальный шум с помощью переменного количества позиционных аргументов

Создавая функцию с необязательными позиционными аргументами (*args), можно сделать код более понятным и устранить *визуальный шум*.

Предположим, например, что вы хотите записывать в журнал некую отладочную информацию. При фиксированном количестве аргументов вам понадобится функция, принимающая сообщение и список значений.

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('Мои числа', [1, 2])
log('Привет', [])
>>>
Мои числа: 1, 2
Привет
```

Передача пустого списка в отсутствие значений для протоколирования выглядит громоздко и вносит визуальный шум. Гораздо лучше вообще опустить второй аргумент. Вы можете сделать это в Python, предварив имя последнего позиционного параметра символом *. Первый параметр требуется для задания текста сообщения, тогда как последующие позиционные аргументы, количество которых может быть произвольным, являются необязательными. Тело функции при этом не изменяется, изменяются лишь вызовы.

```
def log(message, *values): # Единственное отличие
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('Мои числа', 1, 2)
log('Привет') # Намного лучше
>>>
Мои числа: 1, 2
Числа
```

Если у вас уже есть список и вы хотите вызвать функцию с переменным количеством аргументов, такую как функция `log()`, то это можно сделать, используя оператор `*`. Данный оператор указывает Python на то, что элементы последовательности необходимо передать функции как позиционные аргументы.

```
favorites = [7, 33, 99]
log('Любимые цвета', *favorites)

>>>
Любимые цвета: 7, 33, 99
```

С получением переменного количества аргументов связаны две проблемы.

Одна из них заключается в том, что такие аргументы до передачи их функции всегда собираются в кортеж. Это означает, что если код, вызывающий вашу функцию, использует генератор со "звездочкой", то он будет итерироваться до полного исчерпания. В результирующий кортеж войдут все значения из генератора, что может привести к перерасходу памяти и краху программы.

```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)

>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Функции, которые принимают аргументы в виде `*args`, больше всего подходят для ситуаций, относительно которых вам известно, что количество входных данных в списке аргументов будет находиться в разумных пределах. Такой подход идеален для вызовов функций, при которых сразу передается много литеральных значений или имен переменных. Его главные достоинства в том, что он облегчает жизнь программисту и улучшает читаемость кода.

Вторая проблема, связанная с использованием аргументов `*args`, заключается в том, что в будущем вы не сможете добавить новые позиционные аргументы в свою функцию, не внеся изменения во все ее вызовы. Попытка добавления позиционного аргумента перед списком аргументов приведет к тому, что необновленные вызовы функции не будут

давать желаемых результатов, и при этом никаких предупредительных сообщений выведено не будет.

```
def log(sequence, message, *values):
    if not values:
        print('%s: %s' % (sequence, message))
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s: %s' % (sequence, message, values_str))

log(1, 'Любимые', 7, 33) # Обновленный вызов работает нормально
log('Любимые числа', 7, 33) # Старый вызов не работает

>>>
1: Любимые: 7, 33
Любимые числа: 7: 33
```

Проблема здесь заключается в том, что во втором вызове функции `log()` в качестве параметра `message` использовалось значение 7, поскольку аргумент `sequence` не был предоставлен. Обнаруживать ошибки, подобные этой, очень трудно, ведь код по-прежнему выполняется, не генерируя никаких исключений. Чтобы полностью исключить такую возможность в случаях, когда вы хотите расширить функцию, принимающую переменное количество аргументов, следует использовать именованные аргументы (раздел "Рекомендация 21. Повышайте ясность кода, используя именованные аргументы").

Что следует запомнить

- Функции могут принимать переменное количество позиционных аргументов, используя аргумент вида `*args` в объявлении функции.
- Оператор * позволяет использовать элементы последовательности в качестве позиционных параметров при вызове функции.
- Использование оператора * с генератором может приводить к перерасходу памяти и краху программы.
- Добавление новых позиционных параметров в функции, принимающие переменное количество аргументов, может вносить в код трудно обнаруживаемые ошибки.

Рекомендация 19. Обеспечивайте опциональное поведение с помощью именованных аргументов

Как и большинство других языков программирования, Python предоставляет возможность вызова функции с передачей позиционных ар-

гументов, когда значения аргументов сопоставляются с параметрами функции в порядке их следования слева направо.

```
def remainder(number, divisor):
    return number % divisor

assert remainder(20, 7) == 6
```

Кроме того, любой позиционный аргумент может быть задан в виде именованного аргумента, по ключу. В этом случае имя аргумента должно совпадать с именем соответствующего параметра, указанного в определении функции, а значение аргумента присваивается ему внутри скобок в выражении вызова функции. Именованные аргументы могут передаваться в любом порядке, коль скоро указаны все требуемые позиционные аргументы. Допускается использовать любую комбинацию позиционных и именованных аргументов. Все приведенные ниже вызовы эквивалентны.

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

Все позиционные аргументы должны указываться перед именованными.

```
remainder(number=20, 7

>>>

SyntaxError: non-keyword arg after keyword arg
```

Каждый аргумент может быть задан только один раз.

```
remainder(20, number=7)

>>>
TypeError: remainder() got multiple values for argument
↳ 'number'
```

Гибкость, предоставляемая именованными аргументами, обеспечивает три весомых преимущества.

Одно из них заключается в том, что именованные аргументы делают вызов функции более понятным для тех, кто читает код. По виду вызова `remainder(20, 7)` нельзя абсолютно ничего сказать о том, какое из чисел является делимым, а какое делителем, не заглянув в реализацию метода `remainder()`. В случае же вызова, в котором используются именованные

аргументы number=20 и divisor=7, сами имена аргументов сообщают, какую роль играет каждый из них.

Другим достоинством именованных аргументов является то, что они могут иметь значения по умолчанию, указанные в объявлении функции. Это позволяет функции предложить вам дополнительные возможности, а вы можете либо воспользоваться ими, если они вам нужны, либо оставить поведение, заданное по умолчанию. Это способствует уменьшению объема повторяющегося кода и снижению визуального шума.

Предположим, например, что вам нужно вычислить, с какой скоростью вода заполняет емкость. Если емкость находится на весах, то, для того чтобы найти интересующую вас характеристику, можно воспользоваться разностью показаний весов, полученных в разные моменты времени.

```
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff

weight_diff = 0.5
time_diff = 3
flow = flow_rate(weight_diff, time_diff)
print('%.3f кг в секунду' % flow)

>>>
0.167 кг в секунду
```

В типичных случаях требуется вычислить скорость расхода воды, выраженную в килограммах в секунду. В случае больших резервуаров и больших промежутков времени, исчисляемых днями или часами, для этой цели лучше использовать показания счетчика. Вы можете обеспечить это поведение в той же функции, добавив аргумент в виде множителя, регулирующего масштабирование времени.

```
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

Проблема в том, что теперь вам приходится всегда указывать аргумент period при вызове функции даже в том простом случае, когда длительность периода измеряется в секундах (т.е. значение period равно 1).

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

С целью уменьшения визуального шума можно задать для аргумента period значение по умолчанию.

```
def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period
```

Теперь аргумент period стал необязательным.

```
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

Этот метод хорошо работает для простых значений по умолчанию (о том, что может быть не так для сложных значений, читайте в разделе "Рекомендация 20. Используйте значение None и средство Docstrings при задании динамических значений по умолчанию для аргументов").

Третья причина, по которой следует использовать именованные аргументы, состоит в том, что они предлагают мощный способ добавления параметров в функцию с сохранением обратной совместимости с имеющимся вызывающим кодом. Это дает возможность предоставлять дополнительную функциональность, не внося многочисленных изменений в вызывающий код, что чревато появлением в нем новых ошибок.

Предположим, например, что вы хотите расширить рассмотренную выше функцию для расчета скорости потока в других весовых единицах, а не только в килограммах. Это можно сделать, добавив новый необязательный параметр — коэффициент перехода к другой предпочтительной единице измерения.

```
def flow_rate(weight_diff, time_diff,
              period=1, units_per_kg=1):
    return ((weight_diff * units_per_kg) / time_diff) * period
```

Заданное по умолчанию значение аргумента units_per_kg равно 1, что соответствует использованию килограмма в качестве единицы веса. Это означает, что все имеющиеся вызовы не почувствуют никакого изменения в поведении функции. В новых вызовах функции flow_rate() может указываться новый именованный аргумент, позволяющий использовать новое поведение.

```
pounds_per_hour = flow_rate(weight_diff, time_diff,
                            period=3600, units_per_kg=2.2)
```

Единственной проблемой при таком подходе остается то, что необязательные именованные аргументы наподобие period и units_per_kg по-прежнему могут задаваться как позиционные аргументы.

```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

Предоставление необязательных аргументов в виде позиционных может вносить некоторую путаницу, поскольку при этом неясно, чему именно соответствуют значения 3600 и 2.2. Лучше всегда указывать необязательные аргументы по именам и никогда не передавать их в виде позиционных параметров.

Примечание

Обеспечение обратной совместимости за счет использования необязательных именованных параметров имеет критически важное значение для функций, принимающих переменное количество аргументов (см. предыдущую рекомендацию). Но еще лучше всегда использовать аргументы, которые должны указываться только как именованные (раздел "Рекомендация 21. Повышайте ясность кода, используя именованные аргументы").

Что следует запомнить

- Аргументы функций могут задаваться как в виде позиционных, так и именованных аргументов.
- Именованные аргументы позволяют более точно судить об их назначении в тех случаях, когда использование только позиционных аргументов может вносить путаницу.
- Именованные аргументы с заданными значениями по умолчанию упрощают добавление в функцию нового поведения, особенно в тех случаях, когда для функции уже существует вызывающий код.
- Необязательные именованные аргументы всегда должны передаваться по имени, а не как позиционные параметры.

Рекомендация 20. Используйте значение None и средство Docstrings при задании динамических значений по умолчанию для аргументов

Иногда возникает необходимость в использовании нестатического типа в качестве значения по умолчанию для именованного аргумента. Предположим, например, что вы хотите вывести на печать журнальные сообщения, помеченные отметками времени регистрируемых событий. Вам желательно, чтобы по умолчанию отметка соответствовала времени вызова функция. Вы могли бы попытаться применить приведенный ниже подход, предполагая, что аргументы по умолчанию вычисляются заново каждый раз при вызове функции.

```
def log(message, when=datetime.now()):
    print('%s: %s' % (when, message))

log('Привет!')
sleep(0.1)
log('Еще раз привет!')
```

```
>>>
2015-11-15 21:10:10.371432: Привет!
2015-11-15 21:10:10.371432: Еще раз привет!
```

Временные метки имеют одинаковые значения, поскольку `datetime.now()` выполняется только один раз — во время определения функции. Значения аргументов по умолчанию вычисляются только один раз при загрузке модуля; обычно это происходит при запуске программы. После того как модуль, содержащий этот код, загрузится, аргумент по умолчанию `datetime.now()` больше не будет вычисляться.

В соответствии с принятым в Python соглашением, для получения желаемого результата следует предоставить в качестве значения по умолчанию `None` и документировать фактическое поведение средствами `Docstrings` (раздел "Рекомендация 49. Снабжайте строками документирования каждую функцию, класс и модуль"). Когда в вашем коде встречается значение аргумента по умолчанию, равное `None`, вы соответствующим образом выбираете значение по умолчанию.

```
def log(message, when=None):
    """Протоколирование сообщения с меткой времени.

    Args:
        message: Выводимое сообщение.
        when: Дата и время генерации сообщения.
          По умолчанию - текущие дата и время.
    """
    when = datetime.now() if when is None else when
    print('%s: %s' % (when, message))
```

Теперь временные метки будут разными.

```
log('Привет!')
sleep(0.1)
log('Привет еще раз!')

>>>
2015-11-15 21:10:10.472303: Привет!
2015-11-15 21:10:10.573395: Привет еще раз!
```

Особенно важно использовать `None` в качестве значений аргументов по умолчанию в тех случаях, когда аргументы являются *изменяемыми* (mutable) типами. Предположим, например, что вам необходимо загрузить значение, закодированное в формате JSON. Вы хотите, чтобы по умолчанию в случае неудачного декодирования данных возвращался пустой словарь. Для этого можно было бы попытаться применить следующий подход.

```
def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

Здесь возникает та же проблема, что и в приведенном выше примере с `datetime.now()`. Словарь, указанный для `default`, будет совместно использоваться всеми вызовами `decode()`, поскольку значения аргументов по умолчанию вычисляются только один раз (при загрузке модуля). Это может приводить к совершенно неожиданному поведению программы.

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

Вы ожидали увидеть два различных словаря, каждый со своим ключом и значением. Однако изменение одного из них, как нетрудно заметить, изменяет другой. Виной всему тот факт, что переменные `foo` и `bar` обе равны параметру `default`. Они являются одним и тем же объектом словаря.

```
assert foo is bar
```

Лекарство от этого — установить для именованного аргумента по умолчанию значение `None` и документировать это поведение функции средствами встроенного документирования `docstring`.

```
def decode(data, default=None):
    """Загрузка JSON-данных из строки.

    Args:
        data: JSON-данные, подлежащие декодированию.
        default: Возвращаемое значение при неудачном
                 декодировании.
              По умолчанию - пустой словарь.
    """
    if default is None:
        default = {}
    try:
```

```
        return json.loads(data)
    except ValueError:
        return default
```

Теперь выполнение того же тестового кода, что и перед этим, приводит к ожидаемому результату.

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

Что следует запомнить

- Аргументы, заданные по умолчанию, вычисляются только однажды — при определении функции во время загрузки модуля. Это может стать причиной странного поведения динамических значений (таких, как {} или []).
- Используйте None в качестве значения по умолчанию для именованных аргументов, имеющих динамические значения. Документируйте фактическое поведение функции по умолчанию средствами встроенного документирования docstring.

Рекомендация 21. Повышайте ясность кода, используя именованные аргументы

Передача аргументов по ключу (именованные аргументы) — это мощное средство функций Python (см. раздел "Рекомендация 19. Обеспечивайте опциональное поведение с помощью именованных аргументов"). Гибкость именованных аргументов позволяет писать намного более понятный код.

Предположим, например, что вам нужно делить одно число на другое, но при этом отслеживать специальные ситуации. Иногда может быть желательным игнорировать исключения ZeroDivisionError и вместо этого возвращать значение infinity (бесконечность). В других случаях вы захотите игнорировать исключения OverflowError и вместо этого возвращать нуль.

```
def safe_division(number, divisor, ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

Использовать эту функцию не составляет труда. Приведенный ниже вызов будет игнорировать переполнение арифметического устройства при делении и возвратит нуль.

```
result = safe_division(1, 10**500, True, False)
print(result)
```

```
>>>
0.0
```

Следующий вызов будет игнорировать ошибку деления на нуль и возвратит значение `infinity`.

```
result = safe_division(1, 0, False, True)
print(result)
```

```
>>>
inf
```

Проблема заключается в том, что позиции двух булевых переменных, управляющих игнорированием исключений, можно легко перепутать, что приведет к возникновению трудно обнаруживаемых ошибок. Один из способов улучшения читаемости подобного кода — использовать именованные аргументы. По умолчанию такая функция может проявлять повышенную бдительность и всегда генерировать повторные исключения.

```
def safe_division_b(number, divisor,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    # ...
```

После этого вызывающий код может использовать именованные аргументы для того, чтобы указать, какой из флагов, управляющих игно-

рированием исключений, требуется изменить для конкретных операций, перекрывая поведение, заданное по умолчанию.

```
safe_division_b(1, 10**500, ignore_overflow=True)
safe_division_b(1, 0, ignore_zero_division=True)
```

В данном случае проблема заключается в том, что, поскольку эти именованные аргументы определяют опциональное поведение, ничто не заставляет вызывающий код вносить ясность за счет использования именованных аргументов. Даже после переопределения функции `safe_division_b()` вы по-прежнему можете использовать старую форму вызова с позиционными аргументами.

```
safe_division_b(1, 10**500, True, False)
```

Имея дело с такими сложными функциями, как эта, лучше требовать, чтобы вызывающий код отчетливо демонстрировал свои намерения. В Python 3 вы можете потребовать ясности, определяя свою функцию с именованными аргументами, которые задаются только по ключам, а не по позициям.

Ниже функция `safe_division` будет переопределена таким образом, чтобы она могла принимать только именованные аргументы. Символ * в списке аргументов указывает, где заканчиваются позиционные аргументы и начинаются именованные.

```
def safe_division_c(number, divisor, *,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    # ...
```

Теперь вызов функции путем задания позиционных аргументов не будет работать.

```
safe_division_c(1, 10**500, True, False)
```

```
>>>
TypeError: safe_division_c() takes 2 positional arguments but
↳ 4 were given
```

В то же время именованные аргументы и их значения по умолчанию работают так, как ожидается.

```
safe_division_c(1, 0, ignore_zero_division=True) # OK
```

```
try:
    safe_division_c(1, 0)
except ZeroDivisionError:
    pass # Как и ожидалось
```

Задание аргументов, которые могут быть только именованными, в Python 2

К сожалению, в Python 2 отсутствует явный синтаксис для указания аргументов, которые обязаны быть именованными, как это возможно в Python 3. Однако вы можете добиться того же поведения в виде генерирования исключения `TypeErrors` для недействительных вызовов функции, используя оператор `**` в списке аргументов. Оператор `**` аналогичен оператору `*` (см. раздел "Рекомендация 18. Снижайте визуальный шум с помощью переменного количества позиционных аргументов"), за исключением того, что вместо получения переменного числа позиционных аргументов он обеспечивает получение любого количества именованных аргументов, даже если они не определены.

```
# Python 2
def print_args(*args, **kwargs):
    print 'Позиционные:', args
    print Именованные: ', kwargs

print_args(1, 2, foo='bar', stuff='meep')

>>>
Позиционные: (1, 2)
Именованные: {'foo': 'bar', 'stuff': 'meep'}
```

Чтобы обеспечить получение функцией `safe_division()` заведомо именованных аргументов в Python 2, вы указываете в ее вызове аргумент `**kwargs`. Затем вы можете извлечь ожидаемые именованные аргументы из словаря `kwargs`, используя второй аргумент метода `pop()` для указания значения по умолчанию на тот случай, если соответствующий ключ отсутствует. Наконец, вы убеждаетесь в том, что в `kwargs` больше не осталось именованных аргументов, чтобы исключить возможность предоставления вызывающим кодом недопустимых аргументов.

```
# Python 2
def safe_division_d(number, divisor, **kwargs):
    ignore_overflow = kwargs.pop('ignore_overflow', False)
    ignore_zero_div = kwargs.pop('ignore_zero_division', False)
    if kwargs:
        raise TypeError('Неожиданные **kwargs: %r' % kwargs)
    # ...
```

Теперь вы можете вызывать функцию как с предоставлением, так и без предоставления именованных аргументов.

```
safe_division_d(1.0, 10)
safe_division_d(1.0, 0, ignore_zero_division=True)
safe_division_d(1.0, 10**500, ignore_overflow=True)
```

Попытка передачи именованных аргументов как позиционных не сработает, точно так же, как и в Python 3.

```
safe_division_d(1.0, 0, False, True)

>>>
TypeError: safe_division_d() takes exactly 2 arguments
(4 given)
```

Попытка передачи аргументов, которые не ожидаются, также не сработает.

```
safe_division_d(0.0, 0, unexpected=True)

>>>
TypeError: Неожиданные **kwargs: {'unexpected': True}
```

Что следует запомнить

- Использование именованных аргументов проясняет смысл вызова функции.
- Используйте аргументы, которые обязаны быть именованными, для того чтобы вынудить вызывающий код предоставлять именно такие аргументы в тех случаях, когда неправильное использование аргументов может внести путаницу, особенно если функция принимает в качестве аргументов несколько булевых флагов.
- Python 3 поддерживает явный синтаксис для указания тех аргументов функций, которые обязаны задаваться по ключу.
- Python 2 позволяет эмулировать аргументы функций, которые обязаны быть именованными, за счет использования аргумента вида `**kwargs` и генерирования исключений `TypeError` вручную.

3 Классы и наследование

Поскольку Python — объектно-ориентированный язык программирования, он полностью поддерживает наследование, полиморфизм и инкапсуляцию. Решение многих задач в Python часто требует создания новых классов и определения способов их взаимодействия со своими интерфейсами и иерархиями.

Классы и наследование в Python значительно облегчают написание программ, реализующих запланированное поведение в отношении объектов. Использование классов упрощает дальнейшее улучшение и расширение функциональности программ в будущем по мере необходимости. Классы обеспечивают нужную гибкость в условиях изменяющихся требований. Умение применять эти возможности позволит вам писать простой в сопровождении код.

Рекомендация 22. Отдавайте предпочтение структуризации данных с помощью классов, а не словарей или кортежей

Встроенный тип словаря Python отлично подходит для поддержки динамического внутреннего состояния объекта на протяжении его времени жизни. Термином *динамическое состояние* мы описываем ситуации, в которых потребность в хранении некоторых данных может возникать лишь в процессе эксплуатации программы. Предположим, требуется организовать ведение журнала, фиксирующего оценочные баллы (`grades`) для группы студентов, имена которых заранее не известны. Вместо того чтобы задавать имена студентов в виде предопределенных атрибутов, можно использовать для их хранения класс.

```
class SimpleGradebook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
```

```
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)
```

Использование класса не вызывает затруднений.

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
# ...
print(book.average_grade('Isaac Newton'))

>>>
90.0
```

В силу простоты работы со словарями возникает риск их чрезмерного использования, приводящего к созданию хрупкого кода. Предположим, например, что вы хотите расширить класс `SimpleGradebook` для того, чтобы он поддерживал список баллов по предметам, а не только общее количество баллов, заработанных студентом. Вы могли бы это сделать, изменив словарь `_grades` таким образом, чтобы он отображал имена студентов (ключи) в еще один словарь (значения). Тогда внутренний словарь устанавливал бы соответствие между предметами (ключи) и оценками (значения).

```
class BySubjectGradebook(object):
    def __init__(self):
        self._grades = {}
    def add_student(self, name):
        self._grades[name] = {}
```

Задача кажется довольно тривиальной. Методам `report_grade()` и `average_grade()` будет несколько сложнее обрабатывать многоуровневый словарь, но с этим можно справиться.

```
    def report_grade(self, name, subject, grade):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grades[name]
        total, count = 0, 0
```

```
        for grades in by_subject.values():
            total += sum(grades)
            count += len(grades)
            return total / count
```

Использовать этот класс также очень просто.

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
```

Но представьте, что требования к журналу вновь изменились, и теперь в расчет вклада каждой оценки (`score`) в общий балл необходимо включить весовые коэффициенты (`weight`), которые учитывали бы, что оценки, полученные студентом при сдаче промежуточных и заключительных экзаменов, важнее оценок, полученных за обычные контрольные работы. Одним из возможных способов реализации подобной стратегии могло бы быть изменение внутреннего словаря: вместо задания соответствия между предметами (ключи) и оценками (значения) можно задавать соответствие между предметами и кортежами из оценок и весовых коэффициентов (`score, weight`).

```
class WeightedGradebook(object):
    # ...
    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append((score, weight))
```

В то время как для метода `report_grade()` потребовались самые незначительные изменения (значением стал кортеж), в методе `average_grade()` появился цикл в цикле, в результате чего читаемость кода ухудшилась.

```
    def average_grade(self, name):
        by_subject = self._grades[name]
        score_sum, score_count = 0, 0
        for subject, scores in by_subject.items():
            subject_avg, total_weight = 0, 0
            for score, weight in scores:
                # ...
        return score_sum / score_count
```

Использовать класс тоже стало труднее, поскольку смысл числовых позиционных аргументов совершенно не очевиден.

```
book.report_grade(‚Albert Einstein‘, ‚Math‘, 80, 0.10)
```

Как только у вас возникают трудности сродни описанным, самый раз перейти от словарей и кортежей к иерархии классов.

Поначалу вам ничего не было известно о том, что в будущем придется вводить взвешенные оценки, и поэтому использование дополнительных вспомогательных классов воспринималось как неоправданное усложнение кода. Встроенные типы словарей и кортежей Python упрощали решение задачи путем постепенного добавления новых слоев внутренней "бухгалтерии". Однако от этого подхода следует отказываться при глубине вложения слоев (т.е. когда одни словари содержат другие) более одного уровня. В противном случае другим программистам будет трудно разобраться в вашем коде, и его поддержка превратится в сплошной кошмар. Как только вы замечаете, что поддержка данных усложняется, вынесите ее во вспомогательные классы. Это позволит вам предоставить хорошо определенные интерфейсы, эффективнее инкапсулирующие ваши данные. Кроме того, это позволит вам создать абстрактный слой, отделяющий интерфейсы от конкретного варианта реализации.

Рефакторинг классов

Переход к классам можно начать с самого нижнего уровня дерева зависимостей — баллов. Однако использовать для хранения столь простой информации класс было бы, пожалуй, нерационально. Вероятно, для этой цели лучше подходит кортеж, поскольку баллы представляются неизменяемым типом. В данном случае для отслеживания баллов в списке будем использовать кортеж (`score`, `weight`).

```
grades = []
grades.append((95, 0.45))
# ...
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

Использование простых кортежей в качестве позиционных параметров представляет определенную проблему. Дело в том, что если вы захотите ассоциировать с баллами дополнительную информацию, например замечания преподавателя, то это потребует внесения изменений во всех местах кода, где используется двучленный кортеж, чтобы было понятно, что теперь вместо двух элементов используются три. В приведенном ниже коде вместо третьего элемента кортежа, который далее просто игнорируется, указан символ подчеркивания _ (по принятому в Python соглашению ему соответствует неиспользуемая переменная).

```
grades = []
grades.append((95, 0.45, 'Great job'))
# ...
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

Этот шаблон расширения кортежей до все большего и большего размера аналогичен увеличению количества уровней вложения словарей. Как только вы достигнете уровня двучленного кортежа, самый раз подумать об использовании другого подхода.

Тип `namedtuple`, содержащийся в модуле `collections`, делает именно то, что вам нужно. Он позволяет легко определять классы, ответственные за хранение неизменяемых данных небольшого размера.

```
import collections
Grade = collections.namedtuple('Grade', ('score', 'weight'))
```

Эти классы могут конструироваться с позиционными или именованными аргументами. Доступ к полям осуществляется с помощью именованных атрибутов. Если в дальнейшем требования вновь изменятся и понадобится добавить поведение для контейнеров простых данных, именованные атрибуты упростят переход от именованного кортежа (`namedtuple`) к вашему собственному классу.

Ограничения, присущие типу `namedtuple`

Как бы ни был полезен тип `namedtuple` во многих ситуациях, очень важно понимать, в каких случаях он может принести больше вреда, чем пользы.

- Для классов `namedtuple` нельзя указывать значения аргументов по умолчанию. Из-за этого их неудобно использовать в ситуациях, когда данные могут иметь много необязательных свойств. В случае, если количество используемых атрибутов резко возрастает, лучшим выходом может оказаться определение собственного класса.
- Доступ к значениям атрибутов в экземплярах `namedtuple` все еще может осуществляться с использованием числовых индексов и итерирования. Однако использование этой возможности, особенно в случае экстернализуемых API, впоследствии может затруднить переход к реальным классам. Если вы не контролируете все случаи использования ваших экземпляров `namedtuple`, то лучше определить собственный класс.

Далее можно создать класс, который представляет отдельный предмет с набором баллов.

```
class Subject(object):
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

Затем можно создать класс, который представляет набор предметов, изучаемых отдельным студентом.

```
class Student(object):
    def __init__(self):
        self._subjects = {}

    def subject(self, name):
        if name not in self._subjects:
            self._subjects[name] = Subject()
        return self._subjects[name]

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

Наконец, можно создать контейнер для всех студентов, который динамически формируется с использованием имен студентов в качестве ключей.

```
class Gradebook(object):
    def __init__(self):
    self._students = {}

    def student(self, name):
        if name not in self._students:
            self._students[name] = Student()
        return self._students[name]
```

По сравнению с предыдущей реализацией число строк кода этих классов приблизительно в два раза больше. Однако читать этот код гораздо легче. Приведенный ниже пример использования классов отличается от предыдущего большей ясностью и расширяемостью.

```
book = Gradebook()
albert = book.student('Albert Einstein')
math = albert.subject('Math')
math.report_grade(80, 0.10)
# ...
print(albert.average_grade())

>>>
81.5
```

Если понадобится, вы сможете написать методы, обеспечивающие обратную совместимость и облегчающие адаптацию участков кода, в которых используется старый стиль API, к новой иерархии объектов.

Что следует запомнить

- Избегайте создания словарей, значениями которых являются другие словари или длинные кортежи.
- Воспользуйтесь типом `namedtuple` для простых неизменяемых контейнеров, прежде чем пытаться задействовать гибкость полноценных классов.
- Как только вы замечаете, что словари, в которых хранятся обрабатываемые данные, начинают усложняться, переходите к использованию вспомогательных классов.

Рекомендация 23. Принимайте функции вместо классов в случае простых интерфейсов

Многие из встроенных библиотек API (интерфейсов прикладного программирования) Python предоставляют возможность изменять стандартное поведение компонентов с помощью *перехватчиков*, или *хуков*, путем передачи функций. API могут взаимодействовать с вашим кодом путем его обратного вызова с помощью перехватчиков во время их выполнения. Например, метод `sort()` типа `list` принимает необязательный аргумент `key`, который используется для определения значений индексов в процессе сортировки. В приведенном ниже примере список имен отсортирован на основании их длины, предоставляя лямбда-выражение в качестве перехватчика для `key`.

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=lambda x: len(x))
print(names)

>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

В других языках для определения хуков, как правило, используются абстрактные классы. В языке Python многие хуки — это всего лишь функции, не имеющие состояния, с известными аргументами и возвращаемыми значениями. Функции идеально подходят для хуков, поскольку их легче описать и проще определить, чем классы. Функции могут работать в качестве хуков, поскольку функции в языке Python — это *функции первого класса*. Это означает, что функции и методы можно передавать из одних мест программы в другие и ссылаться на них, как на любое другое значение.

Предположим, например, что вы хотите изменить стандартное поведение класса `defaultdic` (более подробно об этом рассказывается в разделе "Рекомендация 46. Используйте встроенные алгоритмы и структуры данных"). Эта структура данных позволяет вам предоставлять функцию, которая будет вызываться при всякой попытке обращения к отсутствующему ключу. Функция должна возвращать значение по умолчанию, которое отсутствующий ключ должен иметь в словаре. Ниже определим такую функцию-перехватчик, которая для каждого отсутствующего ключа выводит сообщение и возвращает `0` в качестве значения по умолчанию.

```
def log_missing():
    print('Добавлен ключ')
    return 0
```

Располагая начальным словарем и набором желаемых дополнений, мы можем заставить функцию `log_missing()` выполниться и вывести на печать два результата (для `'red'` и `'orange'`).

```
current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]

result = defaultdict(log_missing, current)
print('До:', dict(result))
for key, amount in increments:
    result[key] += amount
```

```
print('После: ', dict(result))

>>>
До: {'green': 12, 'blue': 3}
Добавлен ключ
Добавлен ключ
После: {'orange': 9, 'green': 12, 'blue': 20, 'red': 5}
```

Предоставление функций наподобие `log_missing` упрощает создание и тестирование API, поскольку это позволяет отделить побочные эффекты от детерминированного поведения. Предположим, вы хотите, чтобы используемый для пополнения словаря перехватчик, который передается функции `defaultdict()`, подсчитывал общее количество отсутствующих ключей. Один из способов достижения этой цели заключается в том, чтобы использовать замыкание, сохраняющее состояние (см. раздел "Рекомендация 15. Знайте, как замыкания взаимодействуют с областью видимости переменных"). В приведенном ниже примере определена вспомогательная функция, которая использует такое замыкание в качестве перехватчика.

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count # Замыкание с сохранением состояния
        added_count += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
        result[key] += amount

    return result, added_count
```

Выполнение этой функции дает желаемый результат (2), хотя функции `defaultdict()` ничего не известно о том, что перехватчик `missing()` поддерживает состояние. В этом состоит еще одно преимущество получения простых функций для организации интерфейсов. Сокрытие состояния в замыкании позволяет легко добавлять функциональность в будущем.

```
result, count = increment_with_report(current, increments)
assert count == 2
```

Проблема с определением замыканий для функций-перехватчиков, сохраняющих состояние, заключается в том, что читать их код труднее по сравнению с функциями без состояния. Другой возможный подход —

это определить небольшой класс, инкапсулирующий состояние, которое вы хотите отслеживать.

```
class CountMissing(object):
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0
```

В других языках программирования можно было бы ожидать, что функция `defaultdict()` будет видоизменена для согласования интерфейса с функцией `CountMissing()`. Однако в Python, благодаря тому, что его функции являются функциями первого класса, вы можете ссылаться на метод `CountMissing.missing()` непосредственно из объекта и передавать его функции `defaultdict()` в качестве перехватчика значений по умолчанию. Согласование метода с интерфейсом функции не составляет труда.

```
counter = CountMissing()
result = defaultdict(counter.missing, current) # Ссылка на метод

for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

Понять использование вспомогательного класса, как этот, для обеспечения нужного поведения замыкания, сохраняющего состояние, гораздо легче, чем работу вспомогательной функции `increment_with_report()`. В то же время, если рассматривать класс `CountMissing` изолированно, то его назначение вовсе не очевидно. Кто конструирует объект `CountMissing`? Кто вызывает метод `missing()`? Потребуется ли в будущем добавление в класс других публичных (открытых) методов? Пока вы не увидите, каким образом класс используется совместно с функцией `defaultdict()`, он остается для вас загадкой.

Чтобы подобные ситуации не вызывали недоразумений, Python разрешает классам определять специальный метод `__call__()`, который делает возможным вызов объекта как функции. При вызове встроенной функции `callable()` такой экземпляр возвращает значение `True`.

```
class BetterCountMissing(object):
    def __init__(self):
        self.added = 0

    def __call__(self):
```

```
        self.added += 1
        return 0

    counter = BetterCountMissing()
    counter()
    assert callable(counter)
```

Ниже экземпляр `BetterCountMissing` используется в качестве перехватчика значений по умолчанию для словаря `defaultdict`, чтобы отслеживать количество отсутствующих ключей, которые были добавлены.

```
counter = BetterCountMissing()
result = defaultdict(counter, current) # Опирается на функцию
                                       # __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

Этот код гораздо понятнее, чем пример с `CountMissing.missing`. Наличие метода `__call__()` указывает на то, что экземпляры класса будут где-то использоваться в качестве аргументов функций (как перехватчики API). Этот метод направляет новых читателей кода к точке входа, ответственной за первичное поведение класса. Он является твердым признаком того, что класс предназначен для того, чтобы играть роль замыкания с сохранением состояния.

Что важнее всего, словарю `defaultdict` по-прежнему ничего не известно относительно того, что именно происходит, когда вы используете метод `__call__()`. Все, что необходимо `defaultdict`, — это функция для перехватчика значений по умолчанию. Python предоставляет много различных способов для обеспечения согласования с простыми интерфейсами функций в зависимости от решаемых вами задач.

Что следует запомнить

- Часто, вместо того чтобы определять классы и создавать их экземпляры, вам вполне хватит функций для организации простых интерфейсов между компонентами в Python.
- Ссылки на функции и методы в Python относятся к первому классу, а это означает, что их можно использовать в выражениях как любой другой тип.
- Специальный метод `__call__()` обеспечивает возможность вызова экземпляров класса как обычных функций Python.
- Если вам требуется функция для поддержания состояния, то рассмотрите возможность определения для этой цели класса, предоставляющего метод `__call__()`, вместо того чтобы определять за-

мыкание, сохраняющее состояние (см. раздел "Рекомендация 15. Знайте, как замыкания взаимодействуют с областью видимости переменных").

Рекомендация 24. Используйте полиморфизм @classmethod для конструирования объектов обобщенным способом

В языке Python полиморфизм поддерживают не только объекты, но и классы. Что это означает и какую пользу приносит?

Полиморфизм — это способность нескольких классов в иерархии реализовывать собственные уникальные версии метода. Это позволяет многим классам соответствовать одному и тому же интерфейсу или абстрактному базовому классу и в то же время предоставлять разную функциональность (см. примеры в разделе "Рекомендация 28. Используйте наследование от классов из модуля `collections.abc` для создания пользовательских контейнерных типов").

Предположим, например, что вы пишете реализацию MapReduce[1] и вам нужен общий класс для представления входных данных. В следующем фрагменте кода такого рода класс определен с методом `read()`, подлежащим определению в подклассах.

```
class InputData(object):
    def read(self):
        raise NotImplementedError
```

Ниже определен конкретный подкласс класса `InputData`, который читает данные из файла на диске.

```
class PathInputData(InputData):
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        return open(self.path).read()
```

Вы вольны иметь сколько угодно подклассов `InputData` наподобие `PathInputData`, и каждый из них может реализовать для метода `read()` стандартный интерфейс, предназначенный для возврата байтов дан-

[1] MapReduce — это разработанная компанией Google модель (и фреймворк) распределенных вычислений, позволяющая проводить вычисления над большими наборами данных размером порядка нескольких петабайт в среде компьютерных кластеров. — *Примеч. ред.*

ных для обработки. Другие подклассы InputData могут выполнять чтение данных из сети, прозрачно распаковывать данные и т.п.

Вам понадобится аналогичный абстрактный интерфейс для рабочего узла MapReduce, осуществляющего предварительную обработку входных данных стандартным образом.

```
class Worker(object):
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

Ниже определен конкретный подкласс Worker для реализации функции MapReduce, которую в данном случае мы хотим применить: простой счетчик символов новой строки.

```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

    def reduce(self, other):
        self.result += other.result
```

Может показаться, что это отличная реализация, но на данном этапе мы натолкнулись на серьезное препятствие. Что объединяет все эти фрагменты кода? У нас есть удачный набор классов с разумными интерфейсами и абстракциями, но это может принести пользу лишь в том случае, если сконструированы объекты. Что ответственно за создание объектов и организацию всей работы MapReduce?

Простейший подход заключается в том, чтобы вручную создать и объединить объекты с некоторыми вспомогательными функциями. Ниже мы просмотрим содержимое каталога и сконструируем экземпляр PathInputData для каждого файла.

```
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

Далее создадим экземпляры LineCountWorker, используя экземпляры InputData, возвращенные generate_inputs.

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

Мы выполняем эти экземпляры `Worker`, распределяя всю работу, связанную с вызовами метода `map()`, между несколькими потоками (раздел "Рекомендация 37. Используйте потоки для блокирования ввода-вывода, но не для параллелизма"). Далее повторно вызываем метод `reduce()` для свертки предварительно обработанных данных в один окончательный результат.

```
def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    for thread in threads: thread.join()

    first, rest = workers[0], workers[1:]
    for worker in rest:
        first.reduce(worker)
return first.result
```

Наконец, объединяем все разрозненные части в одной функции для выполнения каждого шага.

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

Будучи примененной к набору тестовых входных файлов, эта функция демонстрирует отличную работу.

```
from tempfile import TemporaryDirectory

def write_test_files(tmpdir):
    # ...

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    result = mapreduce(tmpdir)

    print('Имеется', result, 'строк')

>>>
Имеется 4360 строк
```

Какие здесь недостатки? Огромная проблема состоит в том, что функция `mapreduce()` вовсе не является обобщенной. Если вы захотите написать другой подкласс, `InputData` или `Worker`, то для согласования с ним придется переписать также функции `generate_inputs()`, `create_workers()` и `mapreduce()`.

Решение проблемы сводится к нахождению обобщенного способа конструирования объектов. В других языках вы решили бы эту проблему с помощью полиморфизма конструкторов, потребовав, чтобы каждый подкласс `InputData` предоставлял специальный конструктор, который мог бы использоваться как обобщенный вспомогательными методами, организующими работу MapReduce. Трудность состоит в том, что Python разрешает использовать лишь один метод конструктора `__init__()`. Требовать же, чтобы все подклассы `InputData` имели совместимые конструкторы, неразумно.

Легче всего решить эту проблему с помощью полиморфизма `@classmethod`. Этот подход в точности совпадает с полиморфизмом методов экземпляров, который мы использовали в отношении метода `InputData.read()`, за исключением того, что он применяется к целым классам, а не к сконструированным на их основе объектам.

Давайте применим эту идею к классам MapReduce. Ниже мы расширим класс `InputData`, добавляя в него метод обобщенного класса, ответственный за создание новых экземпляров `InputData` с помощью общего интерфейса.

```
class GenericInputData(object):
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

Здесь мы имеем метод `generate_inputs()`, принимающий словарь и набор конфигурационных параметров, интерпретировать которые — задача конкретного подкласса `GenericInputData`. Ниже мы будем использовать аргумент `config` для нахождения каталога с входными файлами.

```
class PathInputData(GenericInputData):
    # ...
    def read(self):
        return open(self.path).read()

    @classmethod
    def generate_inputs(cls, config):
```

```
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

Аналогичным образом можно создать часть класса `GenericWorker` с вспомогательным методом `create_workers()`. Для генерирования необходимых входных данных ниже будет использован параметр `input_class`, который должен быть подклассом `GenericInputData`. Мы конструируем экземпляры конкретного подкласса `GenericWorker`, используя функцию `cls()` в качестве обобщенного конструктора.

```
class GenericWorker(object):
    # ...
    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers
```

Заметьте, что вызов `input_class.generate_inputs()` — это и есть пример полиморфизма классов, который я пытаюсь вам продемонстрировать. Вы также можете видеть, как вызов функции `cls()` в методе `create_workers()` предоставляет альтернативный способ конструирования объектов `GenericWorker` помимо непосредственного использования метода `__init__()`.

Для нашего конкретного подкласса `GenericWorker` результирующий эффект проявляется в изменении его родительского класса.

```
class LineCountWorker(GenericWorker):
    # ...
```

Наконец, можно переписать функцию `mapreduce()` в полностью обобщенном виде.

```
def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)
```

Обработка новым рабочим узлом `worker` набора тестовых файлов дает тот же результат, что и старая реализация. Различие состоит в том,

что теперь функция mapreduce() требует большего количества параметров и может работать как обобщенная.

```
with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
    result = mapreduce(LineCountWorker, PathInputData, config)
```

Теперь вы можете писать другие подклассы GenericInputData и GenericWorker по своему усмотрению, не внося никаких изменений в связующий программный код.

Что следует запомнить

- Python поддерживает лишь один конструктор в каждом классе: метод __init__().
- Используйте аннотацию @classmethod для определения альтернативных конструкторов в своих классах.
- Используйте полиморфизм методов класса для предоставления обобщенных способов построения и объединения конкретных подклассов.

Рекомендация 25. Инициализация родительских классов с помощью встроенной функции super()

Старый способ инициализации родительского класса из дочернего предполагает непосредственный вызов метода __init__() родительского класса дочерним экземпляром.

```
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value

class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
```

Этот подход хорошо работает для простых иерархий, но во многих случаях оказывается неприемлемым.

Если ваш класс зависит от множественного наследования (то, чего в общем случае следует избегать, как описывается в разделе "Рекомендация 26. Используйте множественное наследование лишь для смешанных вспомогательных классов"), то непосредственный вызов методов __init__() суперклассов может приводить к неожиданному поведению.

Одна из проблем заключается в отсутствии определения очередности вызовов `__init__()` для всех подклассов. Например, ниже мы определим два родительских класса, воздействующих на поле `value` экземпляра.

```
class TimesTwo(object):
    def __init__(self):
        self.value *= 2

class PlusFive(object):
    def __init__(self):
        self.value += 5
```

В приведенном ниже классе его родительские классы определены в одной очередности.

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

Его конструирование дает результат, согласующийся с порядком определения родительских классов.

```
foo = OneWay(5)
print('Результат первого варианта: (5 * 2) + 5 = ', foo.value)

>>>
Результат первого варианта: (5 * 2) + 5 = 15
```

А вот еще один класс, который определяет те же родительские классы, но в другой очередности.

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)

        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

Однако при этом мы оставили тот же порядок вызова конструкторов родительских классов `PlusFive.__init__()` и `TimesTwo.__init__()`, что и прежде, в результате чего поведение класса не будет соответствовать порядку следования родительских классов в его определении, и полученный результат будет отличаться от ожидаемого.

```
bar = AnotherWay(5)
print('Результат второго варианта такой же: ', bar.value)
```

```
>>>
Результат второго варианта такой же: 15
```

Еще одна проблема возникает в случае *ромбовидного наследования*. О ромбовидном наследовании говорят тогда, когда подкласс наследует от двух разных классов, у которых где-то в иерархии имеется собственный общий суперкласс. Следствием ромбовидного наследования является то, что метод `__init__()` общего суперкласса выполняется несколько раз, приводя к неожиданному поведению. Например, ниже мы определяем два дочерних класса, которые наследуют от класса `MyBaseClass`.

```
class TimesFive(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 5

class PlusTwo(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 2
```

Затем определяем дочерний класс, который наследует от обоих этих классов, в результате чего класс `MyBaseClass` становится вершиной ромбовидной иерархической структуры.

```
class ThisWay(TimesFive, PlusTwo):
    def __init__(self, value):
        TimesFive.__init__(self, value)
        PlusTwo.__init__(self, value)

foo = ThisWay(5)
print('Должно быть (5 * 5) + 2 = 27, фактически: ', foo.value)

>>>
Должно быть (5 * 5) + 2 = 27, фактически: 7
```

На выходе мы должны были получить 27, поскольку (5 * 5) + 2 = 27. Но вызов конструктора второго родительского класса, `PlusTwo.__init__()`, сбрасывает значение `self.value` до 5, когда конструктор `MyBaseClass.__init__()` вызывается во второй раз.

Для устранения этих проблем в Python 2.2 была добавлена встроенная функция `super()` и определен *порядок разрешения методов* (method resolution order — MRO). MRO стандартизирует, какие суперклассы должны инициализироваться раньше других, а какие позже (например, инициализация начинается с наиболее глубоко вложенных классов в направлении слева направо в соответствии со схемой иерархии). Он

также гарантирует, что общие суперклассы в ромбовидных иерархиях выполняются только однажды.

Ниже мы создаем ромбовидную иерархию классов, но на этот раз для инициализации родительского класса используется функция `super()` (в стиле Python 2).

```
# Python 2
class TimesFiveCorrect(MyBaseClass):
    def __init__(self, value):
        super(TimesFiveCorrect, self).__init__(value)
        self.value *= 5

class PlusTwoCorrect(MyBaseClass):
    def __init__(self, value):
        super(PlusTwoCorrect, self).__init__(value)
        self.value += 2
```

Теперь верхняя часть ромба, метод `MyBaseClass.__init__()`, выполняется только один раз. Другие родительские классы выполняются в том порядке, в каком они указаны в определении класса.

```
# Python 2
class GoodWay(TimesFiveCorrect, PlusTwoCorrect):
    def __init__(self, value):
        super(GoodWay, self).__init__(value)

foo = GoodWay(5)
print 'Должно быть 5 * (5 + 2) = 35, фактически: ', foo.value

>>>
Должно быть 5 * (5 + 2) = 35, фактически: 35
```

Поначалу может показаться, что порядок выполнения обращен. Разве не метод `TimesFiveCorrect.__init__()` должен был выполниться первым? Не должны ли мы были получить в результате (5 * 5) + 2 = 27? Ответ на оба эти вопроса отрицательный. Порядок соответствует тому, который определен процедурой MRO для данного класса и доступен через метод `mro()`.

```
from pprint import pprint
pprint(GoodWay.mro())

>>>
[<class '__main__.GoodWay'>,
<class '__main__.TimesFiveCorrect'>,
<class '__main__.PlusTwoCorrect'>,
```

```
<class '__main__.MyBaseClass'>,
<class 'object'>]
```

Вызов `GoodWay(5)` инициирует цепочку вызовов `TimesFiveCorrect.__init__()`, `PlusTwoCorrect.__init__()` и `MyBaseClass.__init__()`. И только тогда, когда эти вызовы достигают вершины ромба, все методы инициализации начинают фактически выполнять свою работу в очередности, обратной очередности вызова их функций `__init__()`. Метод `MyBaseClass.__init__()` присваивает переменной `value` значение 5. Метод `PlusTwoCorrect.__init__()` добавляет к этому значению 2, в результате чего `value` становится равным 7. Метод `TimesFiveCorrect.__init__()` умножает это число на 5, так что окончательный результат становится равным 35.

Встроенная функция `super()` работает нормально, но с ее использованием в Python 2 все еще связаны две проблемы.

- Ее синтаксис не отличается краткостью. Вы должны указать класс, в котором она определяется, объект `self`, имя метода (обычно `__init__()`) и все аргументы. Такая конструкция может сбивать с толку новичков-программистов в Python.
- Вы должны указать текущий класс по имени в вызове `super()`. Если впоследствии имя класса понадобится изменить, что очень часто бывает в процессе улучшения иерархии классов, то при этом вы будете вынуждены обновить все вызовы `super()`.

К счастью, Python 3 устраняет эти проблемы, сделав вызовы `super()` без аргументов эквивалентными вызовам `super()` с указанием `__class__` и `self`. В Python 3 вы всегда должны использовать функцию `super()`, поскольку она понятна, имеет сжатый синтаксис и всегда все делает правильно.

```
class Explicit(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value * 2)

class Implicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)

assert Explicit(10).value == Implicit(10).value
```

Этот код работает, поскольку Python 3 позволяет вам надежно ссылаться на текущий класс в методах, используя переменную `__class__`. Это не работает в Python 2, поскольку в этой версии переменная `__class__` не определена. Вы могли бы предположить, что можно исполь-

зовать `self.__class__` в качестве аргумента для `super()`, но это не сработает ввиду особенностей реализации метода `super()` в Python 2.

Что следует запомнить

- Принятая в Python стандартная процедура определения порядка разрешения методов (MRO) устраняет проблемы, связанные с порядком инициализации суперклассов и ромбовидным наследованием.
- Всегда используйте встроенную функцию `super()` для инициализации родственных классов.

Рекомендация 26. Используйте множественное наследование лишь для примесных вспомогательных классов

Python — объектно-ориентированный язык со встроенными возможностями надежной организации множественного наследования (см. раздел "Рекомендация 25. Инициализация родительских классов с помощью встроенной функции `super()`"). Тем не менее лучше вообще избегать множественного наследования.

Если вам все же хочется обеспечить те удобства и возможности инкапсуляции данных, которые предоставляются множественным наследованием, то постарайтесь вместо этого использовать *примесные классы*. Так называют небольшие классы, определяющие всего лишь набор дополнительных методов, которые должен предоставить класс. Примесные классы не определяют собственные атрибуты экземпляра и не требуют наличия собственного конструктора `__init__()` для того, чтобы их можно было вызывать.

Написание примесных классов не составляет труда, поскольку Python делает тривиальной проверку текущего состояния любого объекта, независимо от его типа. Динамическая проверка дает возможность написать обобщенную функциональность только один раз (в примесном классе) и далее применять ее во многих других классах. Примесные классы могут объединяться и организовываться в слои для минимизации объема повторяющегося кода и расширения возможностей повторного использования.

Предположим, например, что вы хотите иметь возможность преобразовывать объекты Python из их представления в памяти в словарь, подготовленный к сериализации. Почему бы не написать эту функциональность в обобщенном виде, чтобы ее можно было использовать с любым из ваших классов?

Ниже мы определим в качестве примера примесный класс, который выполняет эту задачу с помощью общедоступного метода, добавляемого в каждый производный класс.

```
class ToDictMixin(object):
    def to_dict(self):
        return self._traverse_dict(self.__dict__)
```

Реализация тривиальна и основывается на динамическом доступе к атрибутам с помощью встроенной функции `hasattr()`, динамической проверке типа с помощью встроенной функции `isinstance()` экземпляра и доступе к словарю экземпляра `__dict__`.

```
def _traverse_dict(self, instance_dict):
    output = {}
    for key, value in instance_dict.items():
        output[key] = self._traverse(key, value)
    return output

def _traverse(self, key, value):
    if isinstance(value, ToDictMixin):
        return value.to_dict()
    elif isinstance(value, dict):
        return self._traverse_dict(value)
    elif isinstance(value, list):
        return [self._traverse(key, i) for i in value]
    elif hasattr(value, '__dict__'):
        return self._traverse_dict(value.__dict__)
    else:
        return value
```

Ниже определен класс, который использует примесный класс для создания представления бинарного дерева в виде словаря.

```
class BinaryTree(ToDictMixin):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

Трансляция большого количества родственных объектов в словарь упрощается.

```
tree = BinaryTree(10,
    left=BinaryTree(7, right=BinaryTree(9)),
    right=BinaryTree(13, left=BinaryTree(11)))
print(tree.to_dict())

>>>
```

```
{'left': {'left': None,
          'right': {'left': None, 'right': None, 'value': 9},
          'value': 7},
 'right': {'left': {'left': None, 'right': None, 'value': 11},
           'right': None,
           'value': 13},
 'value': 10}
```

Самое лучшее в примесных классах — это то, что вы можете делать их обобщенную функциональность подключаемой, что позволяет при необходимости переопределять поведение объектов. Например, ниже мы определим подкласс `BinaryTree`, который хранит ссылку на свой родительский класс. Эта круговая ссылка приведет к тому, что реализация по умолчанию `ToDictMixin.to_dict` будет выполняться в цикле бесконечно долго.

```
class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                 right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent
```

Решение проблемы заключается в том, чтобы перекрыть метод `ToDictMixin._traverse()` в классе `BinaryTreeWithParent`, обеспечивая обработку лишь тех значений, которые существенны, и предотвращая циклы, с которыми сталкивается примесный класс. Ниже мы переопределим метод `_traverse()` таким образом, чтобы обеспечить пропуск родителя и выполнить лишь вставку его числового значения.

```
def _traverse(self, key, value):
    if (isinstance(value, BinaryTreeWithParent) and
            key == 'parent'):
        return value.value # Предотвратить зацикливание
    else:
        return super()._traverse(key, value)
```

Вызов `BinaryTreeWithParent.to_dict()` будет работать без проблем, поскольку циклические ссылки на свойства не обрабатываются.

```
root = BinaryTreeWithParent(10)
root.left = BinaryTreeWithParent(7, parent=root)
root.left.right = BinaryTreeWithParent(9, parent=root.left)
print(root.to_dict())
```

```
>>>
{'left': {'left': None,
          'parent': 10,
          'right': {'left': None,
```

```
                    'parent': 7,
                    'right': None,
                    'value': 9},
          'value': 7},
 'parent': None,
 'right': None,
 'value': 10}
```

Определяя метод `BinaryTreeWithParent._traverse()`, мы тем самым обеспечиваем любому классу, имеющему атрибут типа `BinaryTreeWithParent`, возможность автоматически работать с `ToDictMixin`.

```
class NamedSubTree(ToDictMixin):
    def __init__(self, name, tree_with_parent):
        self.name = name
        self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right)
print(my_tree.to_dict()) # Никаких бесконечных циклов

>>>
{'name': 'foobar',
 'tree_with_parent': {'left': None,
                      'parent': 7,
                      'right': None,
                      'value': 9}}
```

Примесные классы можно объединять. Предположим, например, что вам нужен примесный класс, который предоставляет обобщенную JSON-сериализацию для любого класса. Вы можете сделать это, предположив, что класс предоставляет метод `to_dict()` (который может или не может предоставляться классом `ToDictMixin`).

```
class JsonMixin(object):
    @classmethod
    def from_json(cls, data):
        kwargs = json.loads(data)
        return cls(**kwargs)

    def to_json(self):
        return json.dumps(self.to_dict())
```

Обратите внимание на то, что класс `JsonMixin` определяет как методы класса, так и методы экземпляра. Примесные классы позволяют добавлять любой вид поведения. В этом примере единственное требование, предъявляемое классом `JsonMixin`, заключается в том, чтобы класс имел метод `to_dict()` и его метод `__init__()` принимал именованные

аргументы (см. раздел "Рекомендация 19. Обеспечивайте опциональное поведение с помощью именованных аргументов").

Следующий примесный класс упрощает создание иерархий вспомогательных классов, которые могут сериализоваться и десериализоваться в формате JSON с минимальными усилиями. Вот пример иерархии классов данных, представляющих части топологии дата-центра.

```
class DatacenterRack(ToDictMixin, JsonMixin):
    def __init__(self, switch=None, machines=None):
        self.switch = Switch(**switch)
        self.machines = [
            Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin):
# ...

class Machine(ToDictMixin, JsonMixin):
# ...
```

Сериализация этих классов в JSON и обратная десериализация выполняются тривиально. В приведенном ниже коде проверяется возможность прямого и обратного обмена данными посредством сериализации-десериализации.

```
serialized = """{
    "switch": {"ports": 5, "speed": 1e9},
    "machines": [
        {"cores": 8, "ram": 32e9, "disk": 5e12},
        {"cores": 4, "ram": 16e9, "disk": 1e12},
        {"cores": 2, "ram": 4e9, "disk": 500e9}
    ]
}"""

deserialized = DatacenterRack.from_json(serialized)
roundtrip = deserialized.to_json()
assert json.loads(serialized) == json.loads(roundtrip)
```

В случае использования примесных классов, подобных этому, будет неплохо, если класс уже является наследником класса `JsonMixin`, располагающегося выше в объектной иерархии. Результирующий класс будет вести себя аналогичным образом.

Что следует запомнить

- ✦ Избегайте использования множественного наследования, если примесные классы позволяют добиться тех же результатов.

- Используйте подключаемое поведение на уровне экземпляров для модификации отдельных классов, когда это может потребоваться примесным классам.
- Объединяйте примесные классы для создания сложной функциональности из простых вариантов поведения.

Рекомендация 27. Предпочитайте общедоступные атрибуты закрытым

В Python есть только два типа видимости атрибутов классов: *общедоступные* (публичные, открытые) и *закрытые* (приватные, частные).

```
class MyObject(object):
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

Доступ к общедоступным атрибутам осуществляется посредством точечной нотации.

```
foo = MyObject()
assert foo.public_field == 5
```

Закрытые атрибуты указываются путем предварения их имен префиксом в виде двух символов подчеркивания. К ним можно обращаться непосредственно с помощью методов содержащего класса.

```
assert foo.get_private_field() == 10
```

При попытке непосредственного доступа к закрытым полям извне класса генерируется исключение.

```
foo.__private_field

>>>
AttributeError: 'MyObject' object has no attribute
↳ '__private_field'
```

Методы класса также имеют доступ к закрытым атрибутам, поскольку они объявляются в пределах охватывающего блока `class`.

```
class MyOtherObject(object):
    def __init__(self):
        self.__private_field = 71

    @classmethod
```

```
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field
bar = MyOtherObject()
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

Как и можно было ожидать в случае закрытых полей, доступ подкласса к закрытым полям его родительского класса невозможен.

```
class MyParentObject(object):
    def __init__(self):
        self.__private_field = 71

class MyChildObject(MyParentObject):
    def get_private_field(self):
        return self.__private_field

baz = MyChildObject()
baz.get_private_field()

>>>
AttributeError: 'MyChildObject' object has no attribute
↳ '_MyChildObject__private_field'
```

Поведение закрытого атрибута реализуется простым преобразованием его имени. Когда компилятору Python встречается обращение к закрытому атрибуту в методах наподобие `MyChildObject.get_private_field()`, он транслирует `__private_field` в обращение `_MyChildObject__private_field()`. В этом примере поле `__private_field` было определено лишь в конструкторе `MyParentObject.__init__()`, откуда следует, что реальным именем закрытого атрибута является `_MyParentObject__private_field`. Попытка обращения к закрытому атрибуту родительского класса из дочернего класса закончится неудачей просто из-за несоответствия преобразованному имени.

Зная эту схему, можно легко обращаться к закрытым атрибутам любого класса, из подкласса или извне, не запрашивая разрешения.

```
assert baz._MyParentObject__private_field == 71
```

Если вы просмотрите словарь атрибутов объекта, то увидите, что закрытые атрибуты в действительности хранятся под именами, которые они получают после преобразования.

```
print(baz.__dict__)

>>>
{'_MyParentObject__private_field': 71}
```

Почему синтаксис закрытых атрибутов не предусматривает фактического навязывания строгого контроля видимости? Программисты на Python считают, что преимущества открытости перевешивают недостатки закрытости.

Помимо этого, сама возможность перехвата обращений к атрибутам (раздел "Рекомендация 32. Используйте методы `__getattr__()`, `__getattribute__()` и `__setattr__()` для отложенных атрибутов") позволяет в любой момент добраться до внутренней структуры объектов в случае необходимости. Если вы можете это сделать, то какой смысл Python пытаться каким-то образом предотвратить возможность доступа к закрытым атрибутам?

Чтобы минимизировать риски неосознанного доступа к внутренним данным объектов, программисты на Python придерживаются правил присваивания имен, описанных в руководстве по стилям (см. раздел "Рекомендация 2. Руководствуйтесь правилами стилевого оформления программ, изложенными в документе PEP 8"). Поля с префиксом в виде одиночного символа подчеркивания (наподобие `_protected_field`) являются *защищенными*, т.е. внешние пользователи класса должны соблюдать осторожность при работе с ними.

Однако многие программисты-новички для указания внутренних API, к которым не должны получать доступ подклассы или внешние объекты, используют закрытые поля.

```
class MyClass(object):
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return str(self.__value)

foo = MyClass(5)
assert foo.get_value() == '5'
```

Такой подход не является правильным. Можно не сомневаться, что наступит момент, когда кто-то, включая и вас самих, захочет создать производный класс для добавления нового поведения или устранения недостатков в имеющихся методах (как, например, в методе `MyClass.get_value()`, который всегда возвращает строку). Выбирая вариант закрытых атрибутов, вы заранее обрекаете перекрывающие методы и расширения подкласса на то, что они будут громоздкими и хрупкими. Ваши потенциальные создатели подклассов все равно будут получать доступ к закрытым полям, когда обойтись без этого будет невозможно.

```
class MyIntegerSubclass(MyClass):
    def get_value(self):
```

```
        return int(self._MyClass__value)

foo = MyIntegerSubclass(5)
assert foo.get_value() == 5
```

Но если иерархия классов на более глубоких уровнях изменится, то эти классы разрушатся, поскольку закрытые ссылки будут недействительными. В приведенном ниже примере у класса `MyClass`, непосредственного родителя класса `MyIntegerSubclass`, имеется свой родительский класс — `MyBaseClass`.

```
class MyBaseClass(object):
    def __init__(self, value):
        self.__value = value
    # ...

class MyClass(MyBaseClass):
    # ...

class MyIntegerSubclass(MyClass):
    def get_value(self):
        return int(self._MyClass__value)
```

Теперь значение атрибуту `__value` присваивается в родительском классе `MyBaseClass`, а не в родительском классе `MyClass`. Это приводит к тому, что ссылка на закрытую переменную `self._MyClass__value` внедряется в класс `MyIntegerSubclass`.

```
foo = MyIntegerSubclass(5)
foo.get_value()

>>>
AttributeError: 'MyIntegerSubclass' object has no attribute
↳ '_MyClass__value'
```

Вообще говоря, лучше ориентироваться на то, чтобы разрешать классам выполнять больше работы с помощью защищенных атрибутов. Документируйте каждое защищенное поле и объясняйте, какие из внутренних API доступны подклассам, а какие не следует затрагивать. Это и совет другим программистам, и рекомендация для вас на будущее относительно того, как безопасно расширять собственный код.

```
class MyClass(object):
    def __init__(self, value):
        # Здесь хранится значение, предоставленное пользователем
        # для объекта. Оно должно допускать преобразование в
        # строку. После присвоения объекту его следует
```

```
        # рассматривать как неизменяемое.
        self._value = value
```

Единственная ситуация, в которой можно всерьез задуматься об использовании закрытых атрибутов, — это когда вас беспокоит возможность конфликта имен с подклассами. Такая проблема может возникнуть, если в дочернем классе ненамеренно определяется атрибут, который уже был определен в родительском классе.

```
class ApiClass(object):
    def __init__(self):
        self._value = 5

    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello' # Конфликт

a = Child()
print(a.get(), 'и', a._value, 'должны различаться')

>>>
hello и hello должны различаться
```

В основном это касается классов, которые являются частью открытого API; подклассы находятся вне вашего контроля, и поэтому вы не можете выполнить рефакторинг для устранения проблемы. Возникновение такого конфликта особенно вероятно, если атрибуты носят распространенные имена (такие, как `value`). Дабы уменьшить риск того, что такое может произойти, используйте закрытый атрибут в родительском классе для гарантии отсутствия атрибутов, имена которых перекрывались бы именами дочерних классов.

```
class ApiClass(object):
    def __init__(self):
        self.__value = 5

    def get(self):
        return self.__value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello' # OK!
```

```
a = Child()
print(a.get(), 'и', a._value, 'различаются')

>>>
5 и hello различаются
```

Что следует запомнить

- Компилятор Python не навязывает использования закрытых атрибутов как средства строгого контроля доступа.
- Вместо того чтобы ограничивать доступ к вашим внутренним API и атрибутам, изначально планируйте их, исходя из того, что они будут использоваться производными классами.
- Вместо принудительного управления доступом с помощью закрытых атрибутов облегчайте разработку производных классов, документируя защищенные поля.
- Используйте закрытые атрибуты лишь в тех случаях, когда хотите избежать возможного конфликта имен с подклассами, которые находятся вне вашего контроля.

Рекомендация 28. Используйте наследование от классов из модуля `collections.abc` для создания пользовательских контейнерных типов

Программирование на языке Python во многом связано с определением классов, содержащих данные, и описанием того, как такие объекты связаны между собой. Каждый класс Python представляет собой того или иного рода контейнер, совместно инкапсулирующий атрибуты и функциональность. Кроме того, Python предоставляет встроенные контейнерные типы для управления данными: списки, кортежи, множества и словари.

Вполне естественно, что в процессе проектирования классов для таких простых случаев, как последовательности, вы захотите образовывать подклассы непосредственно от встроенного типа Python `list`. Предположим, например, что вы хотите создать собственный нестандартный тип списка с дополнительными методами, предназначенными для подсчета частотности его членов.

```
class FrequencyList(list):
    def __init__(self, members):
```

```
        super().__init__(members)

    def frequency(self):
        counts = {}
        for item in self:
            counts.setdefault(item, 0)
            counts[item] += 1
        return counts
```

Создавая подкласс типа `list`, вы получаете всю стандартную функциональность списков и сохраняете семантику, с которой знакомы все программисты на Python. Ваши дополнительные методы могут добавить любое желаемое нестандартное поведение.

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('Длина:', len(foo))
foo.pop()
print('После pop:', repr(foo))
print('Frequency:', foo.frequency())

>>>
Длина: 7
После pop: ['a', 'b', 'a', 'c', 'b', 'a']
Frequency: {'a': 3, 'c': 1, 'b': 2}
```

А теперь предположим, что вы хотите предоставить объект, который напоминает тип `list`, разрешая индексирование, но не является его подклассом. Допустим, вы хотите обеспечить семантику последовательности (наподобие `list` или `tuple`) для класса двоичного дерева.

```
class BinaryNode(object):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

Как сделать так, чтобы этот класс вел себя как последовательность? Python реализует варианты поведения контейнеров с помощью методов экземпляров, имеющих специальные имена. Когда вы обращаетесь к элементу последовательности по индексу:

```
bar = [1, 2, 3]
bar[0]
```

этот код будет интерпретироваться так:

```
bar.__getitem__(0)
```

Чтобы заставить класс `BinaryNode` вести себя как последовательность, можно предоставить пользовательскую реализацию метода `__getitem__()`, который выполняет обход дерева объекта.

```
class IndexableNode(BinaryNode):
    def _search(self, count, index):
        # ...
        # Возвращает (found, count)

    def __getitem__(self, index):
        found, _ = self._search(0, index)
        if not found:
            raise IndexError('Выход индекса за пределы диапазона')
        return found.value
```

Вы можете сконструировать бинарное дерево, как обычно.

```
tree = IndexableNode(
10,
left=IndexableNode(
5,
left=IndexableNode(2),
right=IndexableNode(
6, right=IndexableNode(7))),
right=IndexableNode(
15, left=IndexableNode(11)))
```

Но в дополнение к обходу дерева вы также можете получить доступ к объекту как к списку.

```
print('LRR =', tree.left.right.right.value)
print('Индекс 0 =', tree[0])
print('Индекс 1 =', tree[1])
print('11 в tree?', 11 in tree)
print('17 в tree?', 17 in tree)
print('Дерево:', list(tree))
>>>
LRR = 7
Индекс 0 = 2
Индекс 1 = 5
11 в tree? True
17 в tree? False
Дерево: [2, 5, 6, 7, 10, 11, 15]
```

Проблема в том, что реализации `__getitem__()` еще недостаточно для того, чтобы обеспечить всю семантику последовательностей, которую вы ожидаете.

```
len(tree)

>>>
TypeError: object of type 'IndexableNode' has no len()
```

Встроенной функции `len()` требуется еще один метод под названием `__len__`, который должен иметь реализацию для вашего пользовательского типа последовательности.

```
class SequenceNode(IndexableNode):
    def __len__(self):
        _, count = self._search(0, None)
        return count
tree = SequenceNode(
    # ...
)

print('Дерево имеет %d узлов' % len(tree))

>>>
Дерево имеет 7 узлов
```

К сожалению, и это еще не все. Не хватает также методов `count()` и `index()`, на которые программист на Python рассчитывает при работе с последовательностями типа `list` или `tuple`. Определять собственные контейнерные типы гораздо труднее, чем может показаться.

Чтобы избежать этих трудностей в среде Python, встроенный модуль `collections.abc` определяет набор абстрактных базовых классов, которые предоставляют все типичные методы для каждого контейнерного типа. Если вы наследуете от этих абстрактных базовых классов и при этом забываете реализовать требуемые методы, модуль подскажет вам, что что-то не так.

```
from collections.abc import Sequence

class BadType(Sequence):
    pass

foo = BadType()

>>>
TypeError: Can't instantiate abstract class BadType with
↳ abstract methods __getitem__, __len__
```

Когда вы реализуете все методы, требуемые абстрактным базовым классом, как это сделали мы в случае класса `SequenceNode`, он с готовно-

стью предоставит вам все дополнительные методы, такие как `index()` и `count()`.

```
class BetterNode(SequenceNode, Sequence):
    pass

tree = BetterNode(
    # ...
)

print('Индекс 7:', tree.index(7))
print('Счетчик 10:', tree.count(10))

>>>
Индекс 7: 3
Счетчик 10: 1
```

Преимущества использования этих абстрактных классов проявляются в еще большей степени в случае более сложных типов, таких как `Set` или `MutableMapping`, которые имеют большое количество специальных методов, требующих реализации для обеспечения соответствия соглашениям, принятым в Python.

Что следует запомнить

- Наследуйте непосредственно от контейнерных типов Python (таких, как `list` или `dict`) в простых сценариях использования.
- Не забывайте о многочисленных методах, необходимых для корректной реализации пользовательских контейнерных типов.
- Обеспечивайте соответствие ваших классов необходимым интерфейсам и вариантам поведения, создавая пользовательские контейнерные типы путем наследования интерфейсов, определенных в модуле `collections.abc`.

4 Метаклассы и атрибуты

Метаклассы часто упоминаются при обсуждении возможностей Python, но лишь немногие понимают, что они дают на практике. В названии *метакласс* содержится намек на понятие, которое относится к чему-то, находящемуся над классом и вне его. В двух словах: метаклассы позволяют перехватывать инструкции `class` в Python и обеспечивают специальное поведение всякий раз, когда определяется класс.

Примерно такими же загадочными и столь же мощными являются встроенные возможности динамической настройки доступа к атрибутам. В совокупности с объектно-ориентированными конструкциями Python они представляют собой отличный инструментарий, упрощающий переходы от простых классов к сложным.

Однако в этих мощных возможностях есть много скрытых ловушек. Динамические атрибуты позволяют перекрывать объекты и приводят к неожиданным побочным эффектам. Метаклассы могут создавать чрезвычайно причудливые варианты поведения, абсолютно непостижимые для новичков. Важно, чтобы вы следовали *правилу минимума сюрпризов* и использовали эти механизмы для создания понятных аксиом.

Рекомендация 29. Используйте простые атрибуты вместо методов `get()` и `set()`

У программистов, имеющих опыт работы с другими языками, при переходе к Python вполне естественно может возникать соблазн пытаться явно реализовать методы-получатели и методы-установщики свойств в своих классах.

```
class OldResistor(object):

    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
```

```
        return self._ohms

    def set_ohms(self, ohms):
        self._ohms = ohms
```

Применение этих методов не доставляет хлопот, однако такой стиль не в духе Python.

```
r0 = OldResistor(50e3)
print('До: %5r' % r0.get_ohms())
r0.set_ohms(10e3)
print('После: %5r' % r0.get_ohms())

>>>
До: 50000.0
После: 10000.0
```

Особенно неуклюже эти методы выглядят в операциях наподобие инкрементирования на месте.

```
r0.set_ohms(r0.get_ohms() + 5e3)
```

Эти вспомогательные методы действительно помогают определить интерфейс для вашего класса, упрощая инкапсуляцию функциональности, проверку корректности использования и определение границ. Эти задачи играют большую роль в обеспечении гарантий того, что в процессе проектировании классов вам не придется вносить изменения в вызывающие участки кода по мере последующего усовершенствования классов.

Однако в Python вам почти никогда не понадобится явно реализовывать методы-установщики или методы-получатели. Вместо этого вы, как правило, должны всегда начинать свою реализацию с определения простых открытых атрибутов.

```
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0

r1 = Resistor(50e3)
r1.ohms = 10e3
```

Это придает естественность и ясность таким операциям, как инкрементирование.

```
r1.ohms += 5e3
```

Если впоследствии вы решите, что вам необходимо обеспечить специальное поведение при установке значения атрибута, можете привлечь декоратор `@property` и его соответствующий метод-установщик атрибута. Ниже мы определим новый подкласс `Resistor`, с помощью которого можно изменять значение `curent`, изменяя значение свойства `voltage`. Обратите внимание на то, что правильная процедура предполагает совпадение имен метода-установщика и метода-получателя с именем соответствующего свойства.

```
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```

Теперь при присваивании значения свойству `voltage` будет запускаться метод-установщик `voltage()`, соответствующим образом обновляя свойство `current` объекта.

```
r2 = VoltageResistance(1e3)
print('До: %5r A' % r2.current)
r2.voltage = 10
print('После: %5r A' % r2.current)

>>>
До: 0 A
После: 0.01 A
```

Задание метода-установщика в виде свойства позволяет выполнять проверку типа и корректности значений, передаваемых вашему классу. Ниже мы определим класс, гарантирующий, что все значения сопротивления (`resistance`) будут больше 0 Ом (`ohms`).

```
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
```

```
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
            raise ValueError('%f ohms должно быть > 0' % ohms)
        self._ohms = ohms
```

Попытка присвоить недействительное значение сопротивления атрибуту приведет к возникновению исключения.

```
r3 = BoundedResistance(1e3)
r3.ohms = 0

>>>

ValueError: 0.000000 ohms должно быть > 0
```

Исключение будет генерироваться также при передаче недопустимого значения конструктору.

```
BoundedResistance(-5)

>>>
ValueError: -5.000000 ohms должно быть > 0
```

Это произойдет, поскольку метод `BoundedResistance.__init__()` вызовет метод `Resistor.__init__()`, выполняющий присваивание `self.ohms = -5`. Такое присваивание инициирует вызов метода `@ohms.setter()` из `BoundedResistance()`, немедленно выполняющий проверку допустимости данных, прежде чем завершится конструирование объекта.

Декоратор `@property` можно использовать даже для того, чтобы сделать атрибуты родительских классов неизменяемыми.

```
class FixedResistance(Resistor):
    # ...
    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Невозможно установить атрибут")
        self._ohms = ohms
```

Попытка присвоить значение свойству после того, как объект сконструирован, сгенерирует исключение.

```
r4 = FixedResistance(1e3)
r4.ohms = 2e3

>>>
AttributeError: Невозможно установить атрибут
```

Наибольшим недостатком использования декоратора `@property` является то, что методы атрибута могут совместно использоваться лишь подклассами. Совместное использование той же реализации неродственными классами невозможно. Однако Python поддерживает также дескрипторы (раздел "Рекомендация 31. Использование дескрипторов для повторно используемых методов `@property`"), которые разрешают повторное использование логики свойств и многие другие варианты применения.

Наконец, если вы используете методы `@property` для реализации методов-установщиков и методов-получателей, убеждайтесь в том, что реализуемое поведение не принесет сюрпризов. Например, не устанавливайте другие атрибуты в методах-получателях свойств.

```
class MysteriousResistor(Resistor):
    @property
    def ohms(self):
        self.voltage = self._ohms * self.current
        return self._ohms
    # ...
```

Это приведет к крайне странному поведению.

```
r7 = MysteriousResistor(10)
r7.current = 0.01
print('До: %5r' % r7.voltage)
r7.ohms
print('После: %5r' % r7.voltage)

>>>
До: 0
После: 0.1
```

Наилучшая практика заключается в том, чтобы модифицировать состояние соответствующего объекта только в методах `@property.setter()`. Убеждайтесь в том, что вам удалось избежать любых побочных эффектов вне объекта, которые могут быть неожиданными для вызывающего кода, таких как динамический импорт модулей, выполнение медленных вспомогательных функций или создание дорогостоящих за-

просов к базе данных. Пользователи вашего класса будут ожидать, что его атрибуты обладают теми же качествами, что и любой другой объект Python: быстродействием и простотой. Для выполнения сложных или медленно выполняющихся задач используйте обычные методы.

Что следует запомнить

- Определяйте интерфейсы новых классов с помощью простых открытых атрибутов, избегая использования методов-установщиков и методов-получателей свойств.
- Используйте декоратор `@property` для определения специального поведения при попытке доступа к свойствам ваших объектов, когда это необходимо.
- Следуйте правилу минимума сюрпризов и избегайте непонятных побочных эффектов в своих методах `@property`.
- Убеждайтесь в том, что методы `@property` работают быстро: для выполнения медленных и сложных задач используйте обычные методы.

Рекомендация 30. Старайтесь использовать декораторы `@property` вместо рефакторинга атрибутов

Встроенный декоратор `@property` упрощает выполнение дополнительных действий при доступе к атрибутам экземпляров (см. раздел "Рекомендация 29. Используйте простые атрибуты вместо методов `get()` и `set()`"). Одним из распространенных способов применения декоратора `@property` является переход от простого числового атрибута к выполняемому на лету вычислению. Такая возможность крайне полезна, поскольку позволяет внедрять в класс новое поведение без внесения изменений в вызывающие участки кода. Это также можно использовать в качестве важной полумеры для улучшения ваших интерфейсов с течением времени.

Предположим, например, что вы хотите реализовать квоту *алгоритма дырявого ведра* (другое название — *алгоритм текущего ведра*) с помощью простых объектов Python. Здесь класс `Bucket` представляет оставшуюся квоту и длительность периода времени, в течение которого квота будет доступна.

```
class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
```

```
        self.quota = 0

    def __repr__(self):
        return 'Bucket(quota=%d)' % self.quota
```

В основе работы алгоритма дырявого ведра лежит проверка того, что всякий раз, когда ведро заполняется, квота не переносится из одного периода в следующий.

```
def fill(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

Каждый раз, когда потребителю квоты необходимо выполнить какие-либо действия, он прежде всего должен убедиться в том, что может вычесть долю квоты, которую ему необходимо использовать.

```
def deduct(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        return False
    if bucket.quota - amount < 0:
        return False
    bucket.quota -= amount
    return True
```

Прежде чем использовать этот класс, заполним ведро.

```
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)

>>>
Bucket(quota=100)
```

Затем вычтем необходимую нам квоту.

```
if deduct(bucket, 99):
    print('Хватает для квоты 99')
else:
    print('Недостаточно для квоты 99')
print(bucket)

>>>
Хватает для квоты 99
Bucket(quota=1)
```

Наконец, следует предотвратить выполнение дальнейших действий при попытке вычесть из квоты больше, чем доступно. В этом случае уровень квоты остается неизменным.

```
if deduct(bucket, 3):
    print('Хватает для квоты 3')
else:
    print('Недостаточно для квоты 3')
print(bucket)

>>>
Недостаточно для квоты 3
Bucket(quota=1)
```

Проблема, связанная с этой реализацией, заключается в том, что мы никогда не сможем узнать начальный уровень квоты ведра. Со временем из квоты вычитаются некоторые доли, пока она не уменьшится до нуля. В этот момент `deduct()` всегда будет возвращать значение `False`. Когда это происходит, было бы неплохо знать, блокируется ли код, вызывающий метод `deduct()`, по той причине, что в классе `Bucket` исчерпалась квота, или потому, что `Bucket` вообще не имел квоты.

Для устранения этого недостатка можно изменить класс таким образом, чтобы отслеживать максимальную квоту (`max_quota`), установленную для данного периода времени, и квоту, потребленную за этот период (`quota_consumed`).

```
class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    def __repr__(self):
        return ('Bucket(max_quota=%d, quota_consumed=%d)' %
                (self.max_quota, self.quota_consumed))
```

Воспользуемся методом `@property` для вычисления текущего уровня квоты на лету с помощью новых атрибутов.

```
@property
def quota(self):
    return self.max_quota - self.quota_consumed
```

При присваивании значения атрибуту `quota` предпримем специальные меры для согласования текущего интерфейса класса, используемого методами `fill()` и `deduct()`.

```
@quota.setter
def quota(self, amount):
    delta = self.max_quota - amount
    if amount == 0:
        # Переустановка квоты на новый период
        self.quota_consumed = 0
        self.max_quota = 0
    elif delta < 0:
        # Заполнение квоты на новый период
        assert self.quota_consumed == 0
        self.max_quota = amount
    else:
        # Потребление квоты за период
        assert self.max_quota >= self.quota_consumed
        self.quota_consumed += delta
```

Повторное выполнение приведенного выше демонстрационного кода дает тот же результат.

```
bucket = Bucket(60)
print('Начальное состояние', bucket)
fill(bucket, 100)
print('Заполненное состояние', bucket)

if deduct(bucket, 99):
    print('Хватает для квоты 99')
else:
    print('Недостаточно для квоты 99')

print('Сейчас', bucket)

if deduct(bucket, 3):
    print('Хватает для квоты 3')
else:
    print('Недостаточно для квоты 3')

print('Все еще имеется', bucket)

>>>
Initial Bucket(max_quota=0, quota_consumed=0)
Filled Bucket(max_quota=100, quota_consumed=0)
Хватает для квоты 99
Сейчас Bucket(max_quota=100, quota_consumed=99)
Недостаточно для квоты 3
Все еще имеется Bucket(max_quota=100, quota_consumed=99)
```

Самое главное — это то, что код, использующий `Bucket.quota`, не нуждается в изменении и даже не обязан знать, что класс изменился. Новые варианты использования класса `Bucket` могут выполняться соответствующим образом и обращаться непосредственно к `max_quota` и `quota_consumed`.

Декоратор `@property` мне нравится, в частности, тем, что он позволяет вносить инкрементные улучшения в модель данных с течением времени. Знакомясь с приведенным выше примером с классом `Bucket`, вы, возможно, подумали про себя: "В первую очередь следует реализовать `fill()` и `deduct()` как методы экземпляров". Хотя вы, вероятно, были бы правы (см. раздел "Рекомендация 22. Отдавайте предпочтение структуризации данных с помощью классов, а не словарей или кортежей"), на практике встречается множество ситуаций, в которых начальные варианты объектов содержат плохо определенные интерфейсы или просто выступают в роли контейнеров данных. Это случается тогда, когда с течением времени увеличиваются размеры кода и область видимости, многочисленные авторы вносят правки, не заботясь о долговременной чистоте кода, и т.п. Декоратор `@property` — это инструмент, помогающий решать проблемы, с которыми приходится сталкиваться в реальном коде. Не переусердствуйте. Если обнаруживается, что вы то и дело расширяете методы `@property`, то, вероятно, самый раз выполнить рефакторинг класса вместо латания дыр в плохо спроектированном коде.

Что следует запомнить

- Используйте декоратор `@property` для того, чтобы предоставить существующим экземплярам атрибутов новую функциональность.
- Вносите инкрементные улучшения моделей данных, используя декоратор `@property`.
- Постарайтесь выполнить рефакторинг класса и всех вызывающих участков кода, если обнаруживается, что вы чрезмерно используете декоратор `@property`.

Рекомендация 31. Используйте дескрипторы для повторно используемых методов `@property`

Большой проблемой встроенного декоратора `@property` (см. разделы "Рекомендация 29. Используйте простые атрибуты вместо методов `get()` и `set()`" и "Рекомендация 30. Старайтесь использовать декораторы `@property` вместо рефакторинга атрибутов") является повторное использование. Декорируемые им методы не могут повторно применяться

к нескольким атрибутам одного и того же класса. Они также не могут повторно использоваться с другими классами, не имеющими родственных отношений с данным.

Предположим, вы хотите, чтобы класс осуществлял проверку того, что оценка, полученная студентом за выполнение домашнего задания, выражена в процентах.

```
class Homework(object):
    def __init__(self):
        self._grade = 0
    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError('Значение оценки должно находиться
↳ в пределах от 0 до 100')
        self._grade = value
```

Применение декоратора `@property` упрощает использование этого класса.

```
galileo = Homework()
galileo.grade = 95
```

Предположим, вы также хотите оценить студента за сдачу экзамена, охватывающего несколько предметов, каждый из которых оценивается отдельно.

```
class Exam(object):
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError('Значение оценки должно находиться
↳ в пределах от 0 до 100')
```

Трудоемкость такого подхода быстро возрастает. Каждая составляющая экзамена требует добавления нового декоратора `@property` и организации соответствующей проверки.

```
    @property
    def writing_grade(self):
        return self._writing_grade

    @writing_grade.setter
    def writing_grade(self, value):
        self._check_grade(value)
        self._writing_grade = value

    @property
    def math_grade(self):
        return self._math_grade

    @math_grade.setter
    def math_grade(self, value):
        self._check_grade(value)
        self._math_grade = value
```

Кроме того, такой подход не является общим. Если вы захотите повторно использовать эти средства проверки вне контекста домашних заданий и экзаменов, то вам придется повторно переписывать шаблон `@property` и метод `_check_grade()`.

Лучший способ решения этой задачи в Python — использовать *дескриптор*. Протокол дескриптора определяет, как именно доступ к атрибуту интерпретируется языком. Класс `descriptor` может предоставлять методы `__get__()` и `__set__()`, позволяющие повторно использовать поведение, обеспечивающее валидацию оценок без использования какого-либо шаблона. В данном случае дескрипторы также являются более предпочтительным вариантом, чем примесные классы (см. раздел "Рекомендация 26. Используйте множественное наследование лишь для примесных вспомогательных классов"), поскольку они дают возможность повторно использовать ту же логику для многих других атрибутов одного и того же класса.

Ниже мы определим новый класс `Exam` с атрибутами класса, которые являются экземплярами `Grade`. Класс `Grade` реализует протокол дескриптора. Прежде чем я объясню, как работает класс `Grade`, вам важно понять, что делает Python, когда ваш код обращается к таким дескрипторным атрибутам через экземпляр `Exam`.

```
class Grade(object):
    def __get__(*args, **kwargs):
        # ...

    def __set__(*args, **kwargs):
        # ...
```

```
class Exam(object):
    # Атрибуты класса
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```

Когда вы присваиваете значение свойству:

```
exam = Exam()
exam.writing_grade = 40
```

оно будет интерпретироваться следующим образом:

```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

Когда вы извлекаете свойство:

```
print(exam.writing_grade)
```

оно будет интерпретироваться следующим образом:

```
print(Exam.__dict__['writing_grade'].__get__(exam, Exam))
```

Ответственным за это поведение является метод `__getattribute__()` объекта (раздел "Рекомендация 32. Используйте методы `__getattr__()`, `__getattribute__()` и `__setattr__()` для отложенных атрибутов"). В двух словах: если в экземпляре `Exam` атрибут `writing_grade` отсутствует, то Python обратится вместо него к атрибуту класса `Exam`. Если этот атрибут класса является объектом, у которого есть методы `__get__()` и `__set__()`, то Python предположит, что вы хотите следовать протоколу дескриптора.

Теперь, когда вам известно об этом поведении, а также о том, как я использовал декоратор `@property` для валидации оценок в классе `Homework`, вам будет понятна представленная ниже попытка реализации дескриптора `Grade`.

```
class Grade(object):
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError('Значение оценки должно находиться
↪ в пределах от 0 до 100')
        self._value = value
```

К сожалению, это неверный подход, который приводит к неустойчивому поведению. Доступ к нескольким атрибутам одного и того же экземпляра Exam работает так, как ожидается.

```
first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
print('Литературоведение', first_exam.writing_grade)
print('Наука', first_exam.science_grade)

>>>
Литературоведение 82
Наука 99
```

Однако попытка доступа к этим же атрибутам, но относящимся к нескольким экземплярам Exam, приводит к неожиданному поведению.

```
second_exam = Exam()
second_exam.writing_grade = 75
print('Второй', second_exam.writing_grade, 'правильно')
print('Первый ', first_exam.writing_grade, 'неправильно')

>>>
Второй 75 правильно
Первый 75 неправильно
```

Суть проблемы заключается в том, что для атрибута класса writing_grade один экземпляр Grade разделяется всеми экземплярами Exam. Экземпляр Grade для этого атрибута конструируется только один раз за все время жизни программы при первоначальном определении класса Exam, а не всякий раз, когда создается экземпляр Exam.

Чтобы решить эту проблему, нужно, чтобы класс Grade отслеживал свои значения для каждого отдельного экземпляра Exam. Мы можем это обеспечить, сохраняя состояние каждого экземпляра в словаре.

```
class Grade(object):
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError('Значение оценки должно находиться
```

```
↳ в пределах от 0 до 100')
        self._values[instance] = value
```

Эта реализация проста и хорошо работает, но может приводить к утечке памяти. В словаре будут храниться ссылки на каждый экземпляр `Exam`, когда-либо переданный методу `__set__()` на протяжении всего времени жизни программы. Вследствие этого счетчик ссылок на экземпляры никогда не обнуляется, тем самым препятствуя освобождению занимаемой объектами памяти сборщиком мусора.

Для устранения этого недостатка можно использовать встроенный модуль Python `weakref`. Он предоставляет специальный класс под названием `WeakKeyDictionary`, который может заменить собой простой словарь, используемый для хранения `_values`. Уникальность поведения `WeakKeyDictionary` состоит в том, что он удаляет экземпляры `Exam` из своего набора ключей, как только среде выполнения становится известно, что в нем хранится последняя из оставшихся в программе ссылок на экземпляры. Python организует все вместо вас и убедится в том, что словарь `_values` будет пуст, когда ни один из экземпляров `Exam` больше не будет использоваться.

```
class Grade(object):
    def __init__(self):
        self._values = WeakKeyDictionary()
    # ...
```

Когда используется эта реализация дескриптора `Grade`, все работает так, как и ожидалось.

```
class Exam(object):
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
second_exam.writing_grade = 75
print('Первый ', first_exam.writing_grade, 'правильно')
print('Второй', second_exam.writing_grade, 'правильно')

>>>
Первый 82 правильно
Второй 75 правильно
```

Что следует запомнить

- Повторно используйте поведение и валидацию методов `@property`, определяя собственные классы дескрипторов.
- Используйте класс `WeakKeyDictionary` для гарантии того, что ваши классы дескрипторов не приводят к утечке памяти.
- Не увязните в попытках детально разобраться в том, как метод `__getattribute__()` использует протокол дескриптора для получения и установки значений атрибутов.

Рекомендация 32. Используйте методы `__getattr__()`, `__getattribute__()` и `__setattr__()` для отложенных атрибутов

Перехватчики в языке Python упрощают написание обобщенного кода для объединения систем. Предположим, например, что вы хотите представить строки вашей базы данных (БД) в виде объектов Python. База данных имеет настроенную схему. Ваш код, который использует объекты, соответствующие строкам БД, также должен знать, что собой представляет эта база данных. Однако в Python коду, подключающему объекты Python к БД, не обязательно знать схему ваших строк, и он может быть обобщенным.

Как это возможно? С обычными атрибутами экземпляров, методами `@property` и дескрипторами этого сделать нельзя, поскольку все они должны быть заранее определены. В Python подобное динамическое поведение обеспечивается специальным методом `__getattr__()`. Если ваш класс определяет `__getattr__()`, то всякий раз, когда атрибут не удается найти в словаре экземпляров объекта, будет вызываться этот метод.

```
class LazyDB(object):
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
    value = 'Значение для %s' % name
    setattr(self, name, value)
    return value
```

В приведенном ниже коде мы попытаемся обратиться к отсутствующему свойству `foo`. Это вынуждает Python вызвать метод `__getattr__()`, который изменяет словарь экземпляра `__dict__`.

```
data = LazyDB()
print('До:', data.__dict__)
print('foo: ', data.foo)
print('После: ', data.__dict__)

>>>
До: {'exists': 5}
foo: Значение для foo
После: {'exists': 5, 'foo': 'Значение для foo'}
```

Далее давайте добавим в класс `LazyDB` регистрацию вызовов метода `__getattr__()`, чтобы продемонстрировать, когда именно это фактически происходит. Обратите внимание на то, что для получения реального значения свойства используется вызов `super().__getattr__()` во избежание бесконечной рекурсии.

```
class LoggingLazyDB(LazyDB):
    def __getattr__(self, name):
        print('Вызов __getattr__(%s)' % name)
        return super().__getattr__(name)

data = LoggingLazyDB()
print('exists:', data.exists)
print('foo: ', data.foo)
print('foo: ', data.foo)

>>>
exists: 5
Вызов __getattr__(foo)
foo: Значение для foo
foo: Значение для foo
```

Атрибут `exists` присутствует в словаре экземпляра, и поэтому метод `__getattr__()` для него никогда не вызывается. Поскольку атрибут `foo` первоначально отсутствует в словаре, то при первом обращении к нему вызывается метод `__getattr__()`. Но вызов `__getattr__()` для атрибута `foo` одновременно инициирует и вызов метода `setattr()`, который вносит `foo` в словарь экземпляров. Вот почему при втором обращении к атрибуту `foo` метод `__getattr__()` не вызывается.

Такое поведение особенно полезно в случаях наподобие отложенного доступа к данным без схемы. Метод `__getattr__()` выполняется только один раз для того, чтобы выполнить тяжелую работу по загрузке свойства, а все дальнейшие попытки доступа возвращают уже существующий результат.

Предположим также, что в этой системе с базой данных вам надо выполнять транзакции. Поэтому, когда пользователь в следующий раз об-

ратится к свойству, вы хотите знать, все ли еще действительна соответствующая строка базы данных и открыта ли по-прежнему транзакция. Перехватчик `__getattr__()` не позволит вам сделать это надежно, поскольку он будет использовать словарь экземпляра объекта в качестве наиболее быстрого пути доступа к существующим атрибутам.

Для таких вариантов использования в языке Python предусмотрен другой перехватчик — `__getattribute__()`. Этот специальный метод вызывается при всяком обращении к атрибуту через объект, даже в тех случаях, когда он существует в словаре атрибутов. Это дает возможность выполнять такие операции, как проверка глобального состояния транзакции при каждом обращении к свойству. Ниже мы определим класс `ValidatingDB`, регистрирующий каждый вызов `__getattribute__()`.

```
class ValidatingDB(object):
    def __init__(self):
        self.exists = 5

    def __getattribute__(self, name):
    print('Вызов __getattribute__(%s)' % name)
    try:
        return super().__getattribute__(name)
    except AttributeError:
        value = 'Значение для %s' % name
        setattr(self, name, value)
        return value

data = ValidatingDB()
print('exists:', data.exists)
print('foo: ', data.foo)
print('foo: ', data.foo)

>>>
Вызов __getattribute__(exists)
exists: 5
Вызов __getattribute__(foo)
foo: Значение для foo
Вызов __getattribute__(foo)
foo: Значение для foo
```

В случае, если свойство, к которому осуществляется динамический доступ, не существует, вы можете сгенерировать исключение `AttributeError`, чтобы вызвать стандартную реакцию Python на отсутствие свойства как для метода `__getattr__()`, так и для метода `__getattribute__()`.

```
class MissingPropertyDB(object):
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError('Отсутствует %s' % name)
        # ...
data = MissingPropertyDB()
data.bad_name

>>>
AttributeError: Отсутствует bad_name
```

Код Python, реализующий обобщенную функциональность, часто опирается на встроенную функцию `hasattr()` для определения существования свойств и встроенную функцию `getattr()` для извлечения их значений. Кроме того, прежде чем вызвать `__getattr__()`, выполняется поиск имени атрибута в словаре экземпляра.

```
data = LoggingLazyDB()
print('До: ', data.__dict__)
print('foo существует: ', hasattr(data, 'foo'))
print('После: ', data.__dict__)
print('foo существует: ', hasattr(data, 'foo'))

>>>
До: {'exists': 5}
Вызов __getattr__(foo)
foo существует: True
После: {'exists': 5, 'foo': 'Значение для foo'}
foo существует: True
```

В приведенном выше примере `__getattr__()` вызывается только один раз. В отличие от этого, классы, которые реализуют `__getattribute__()`, будут вызывать этот метод всякий раз, когда для объекта выполняется метод `hasattr()` или `getattr()`.

```
data = ValidatingDB()
print('foo существует: ', hasattr(data, 'foo'))
print('foo существует: ', hasattr(data, 'foo'))

>>>
Called __getattribute__(foo)
foo существует: True
Вызов __getattribute__(foo)
foo существует: True
```

Предположим далее, что вы хотите организовать отложенную загрузку данных в БД при присваивании значений вашему объекту Python. Вы можете сделать это с помощью метода `__setattr__()` — перехватчи-

ка, который позволяет перехватывать произвольные операции присваивания значений атрибутам. В отличие от извлечения атрибутов с помощью методов `__getattr__()` и `__getattribute__()`, в данном случае не нужны два метода. Метод `__setattr__()` вызывается каждый раз при присваивании значения атрибута для экземпляра (либо непосредственно, либо через встроенную функцию `setattr()`).

```
class SavingDB(object):
    def __setattr__(self, name, value):
        # Сохранить некоторые данные в журнал БД
        # ...
        super().__setattr__(name, value)
```

Ниже мы определим регистрирующий класс `SavingDB`. Его метод `__setattr__()` всегда вызывается при каждом присваивании значения атрибута.

```
class LoggingSavingDB(SavingDB):
    def __setattr__(self, name, value):
        print('Вызов __setattr__(%s, %r)' % (name, value))
        super().__setattr__(name, value)

data = LoggingSavingDB()
print('До: ', data.__dict__)
data.foo = 5
print('После: ', data.__dict__)
data.foo = 7
print('Окончательно:', data.__dict__)

>>>
До: {}
Вызов __setattr__(foo, 5)
После: {'foo': 5}
Вызов __setattr__(foo, 7)
Окончательно: {'foo': 7}
```

Проблема использования методов `__getattribute__()` и `__setattr__()` заключается в том, что они вызываются при каждой попытке доступа к атрибуту для объекта, даже если вы, возможно, этого не хотели. Например, предположим, вы хотите, чтобы при осуществлении вашим объектом попытки доступа к атрибуту выполнялся фактический поиск ключей в ассоциированном словаре.

```
class BrokenDictionaryDB(object):
    def __init__(self, data):
        self._data = data
```

```
    def __getattribute__(self, name):
        print('Вызов __getattribute__(%s)' % name)
        return self._data[name]
```

Это требует доступа к `self._data` из метода `__getattribute__()`. Однако, если вы попытаетесь фактически это сделать, Python войдет в рекурсивный цикл до исчерпания стека, после чего выполнение программы прекратится.

```
data = BrokenDictionaryDB({'foo': 3})
data.foo

>>>
Вызов __getattribute__(foo)
Вызов __getattribute__(_data)
Вызов __getattribute__(_data)
...
Traceback ...
RuntimeError: maximum recursion depth exceeded
```

Проблема в том, что метод `__getattribute__()` обращается к `self._data`, что приводит к повторному выполнению `__getattribute__()`, который вновь обращается к `self._data`, и т.д. Решение этой проблемы заключается в использовании метода `super().__getattribute__()` вашим экземпляром для извлечения значений из словаря атрибутов экземпляра. Это позволяет избежать рекурсии.

```
class DictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        data_dict = super().__getattribute__('_data')
        return data_dict[name]
```

Точно так же вам потребуется, чтобы методы `__setattr__()`, которые изменяют атрибуты объекта, использовали метод `super().__setattr__()`.

Что следует запомнить

- Используйте методы `__getattr__()` и `__setattr__()` для отложенной загрузки и сохранения атрибутов объекта.
- Необходимо понимать, что метод `__getattr__()` вызывается только один раз в случае попытки доступа к отсутствующему атрибуту, тогда как метод `__getattribute__()` вызывается при каждой попытке доступа к атрибуту.

- Избегайте бесконечных рекурсий в методах `__getattribute__()` и `__setattr__()`, используя для непосредственного доступа к атрибутам экземпляра методы из `super()` (т.е. класса объекта).

Рекомендация 33. Верификация подклассов с помощью метаклассов

Одно из простейших применений метаклассов — верификация того, что класс определен корректно. В процессе создания сложной иерархии классов вы можете захотеть контролировать соблюдение определенного стиля, требовать перекрытия методов или поддерживать строгие отношения между атрибутами класса. Метаклассы позволяют осуществить это, предоставляя надежный способ выполнения верифицирующего кода всякий раз, когда определяется новый подкласс.

Верифицирующий код нередко выполняется в методе `__init__()` при конструировании объектов, принадлежащих типу данного класса (см., например, раздел "Рекомендация 28. Используйте наследование от классов из модуля `collections.abc` для создания пользовательских контейнерных типов"). Использование метаклассов для верификации классов может генерировать ошибки гораздо раньше.

Прежде чем мы перейдем к рассмотрению того, как определить метакласс, предназначенный для верификации подклассов, необходимо, чтобы вы поняли, как метаклассы действуют в отношении стандартных объектов. Метакласс определяется путем наследования типа. По умолчанию метакласс получает содержимое инструкций ассоциированного с ним класса в своем методе `__new__()`. С помощью приведенного ниже кода вы можете изменить информацию о классе до фактического конструирования типа.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print((meta, name, bases, class_dict))
        return type.__new__(meta, name, bases, class_dict)

class MyClass(object, metaclass=Meta):
stuff = 123

def foo(self):
    pass
```

Метакласс имеет доступ к именам класса, его родительских классов и всех атрибутов класса, определенных в теле класса.

```
>>>
(<class '__main__.Meta'>,
```

```
 'MyClass',
 (<class 'object'>,),
 {'__module__': '__main__',
  '__qualname__': 'MyClass',
  'foo': <function MyClass.foo at 0x102c7dd08>,
  'stuff': 123})
```

В Python 2 для этого предусмотрен несколько отличный синтаксис, в соответствии с которым метакласс указывается с помощью атрибута класса `__metaclass__`. Интерфейс `Meta.__new__()` означает то же самое.

```
# Python 2
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        # ...

class MyClassInPython2(object):
    __metaclass__ = Meta
    # ...
```

Чтобы обеспечить верификацию всех параметров класса, прежде чем он будет определен, можно добавить функциональность в метод `Meta.__new__()`. Предположим, например, что вы хотите представить многоугольник произвольного типа. Это можно сделать, определив специальный верифицирующий метакласс и использовав его в базовом классе вашей иерархии классов многоугольников. Обратите внимание на то, что очень важно не применять эти же правила верификации к базовому классу.

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Не следует проверять абстрактный класс Polygon
        if bases != (object,):
            if class_dict['sides'] < 3:
                raise ValueError('Многоугольник должен иметь
↳ не менее 3 сторон')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(object, metaclass=ValidatePolygon):
    sides = None # Определяется подклассами

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3
```

Если вы попытаетесь определить многоугольник, в котором число сторон меньше трех, то в результате верификации инструкция `class` будет опознана как недействительная сразу вслед за телом инструкции. Это означает, что при наличии такого определения класса ваша программа даже не сможет начать выполнение.

```
print('До class')
class Line(Polygon):
    print('Перед sides')
    sides = 1
    print('После sides')
print('После class')

>>>
До class
До sides
После sides
Traceback ...
ValueError: Многоугольник должен иметь не менее 3 сторон
```

Что следует запомнить

- Используйте метаклассы для гарантии корректности формирования подклассов во время их определения, прежде чем из них будут конструироваться объекты этого типа.
- Метаклассы имеют несколько отличающийся синтаксис в Python 2 и Python 3.
- Метод `__new__()` метаклассов выполняется по завершении обработки всего тела инструкции `class`.

Рекомендация 34. Регистрируйте существование классов с помощью метаклассов

Другая распространенная область применения метаклассов — автоматическая регистрация типов в программе. Регистрация может быть полезна при выполнении обратного поиска, когда требуется обратное отображение простого идентификатора в соответствующий класс.

Предположим, например, что вы хотите реализовать собственное сериализованное представление объекта Python с помощью JSON. Вы должны иметь возможность принять объект и превратить его в строку JSON. Ниже мы сделаем это обобщенным способом, определяя базовый класс, который записывает параметры конструктора и превращает их в словарь JSON.

```
class Serializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})
```

Этот класс облегчает процесс сериализации простых, неизменяемых структур данных наподобие `Point2D` в строку.

```
class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Point2D(%d, %d)' % (self.x, self.y)

point = Point2D(5, 3)
print('Объект: ', point)
print('Сериализованный объект:', point.serialize())

>>>
Объект: Point2D(5, 3)
Сериализованный объект: {"args": [5, 3]}
```

Затем необходимо десериализовать эту JSON-строку и сконструировать объект `Point2D`, который она представляет. Ниже мы определим другой класс, который может десериализовать данные из их родительского класса `Serializable`.

```
class Deserializable(Serializable):
    @classmethod
    def deserialize(cls, json_data):
        params = json.loads(json_data)
        return cls(*params['args'])
```

Использование класса `Deserializable` облегчает сериализацию и десериализацию простых, неизменяемых объектов обобщенным способом.

```
class BetterPoint2D(Deserializable):
    # ...
point = BetterPoint2D(5, 3)
print('До: ', point)
data = point.serialize()
print('Сериализованный объект:', data)
after = BetterPoint2D.deserialize(data)
```

```
print('После: ', after)

>>>
До: BetterPoint2D(5, 3)
Сериализованный объект: {"args": [5, 3]}
После: BetterPoint2D(5, 3)
```

Проблема этого подхода заключается в том, что он работает только в том случае, если вам заранее известен целевой тип сериализованных данных (например, `Point2D`, `BetterPoint2D`). В идеальном случае у вас может быть много классов, требующих сериализации в JSON, и только одна функция, которая могла бы десериализовать любой из них обратно в соответствующий объект Python.

Для этого можно включить имя сериализованного объекта в JSON-данные.

```
class BetterSerializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })
    def __repr__(self):
        # ...
```

Далее можно обеспечить поддержку обратного отображения имен классов в конструкторы этих объектов. Общая функция десериализации будет работать для любого класса, переданного `register_class()`.

```
registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)
    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])
```

Чтобы гарантировать, что десериализация всегда работает так, как надо, следует вызвать `register_class()` для каждого класса, необходимость в сериализации которого может возникнуть в будущем.

```
class EvenBetterPoint2D(BetterSerializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

register_class(EvenBetterPoint2D)
```

Теперь можно десериализовать любую JSON-строку, даже не зная, какой класс в ней содержится.

```
point = EvenBetterPoint2D(5, 3)
print('До: ', point)
data = point.serialize()
print('Сериализованный объект:', data)
after = deserialize(data)
print('После: ', after)

>>>
После: EvenBetterPoint2D(5, 3)
Сериализованный объект: {"class": "EvenBetterPoint2D",
"args": [5, 3]}
После: EvenBetterPoint2D(5, 3)
```

Проблема этого подхода заключается в том, что вы можете забыть вызвать `register_class()`.

```
class Point3D(BetterSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x = x
        self.y = y
        self.z = z
# Забыли вызвать register_class! Оп-па!
```

Это приведет к тому, что ваш код не сработает на этапе выполнения, когда вы, наконец, попытаетесь десериализовать объект класса, который забыли зарегистрировать.

```
point = Point3D(5, 9, -4)
data = point.serialize()
deserialize(data)

>>>
KeyError: 'Point3D'
```

Даже если вы решите наследовать класс `BetterSerializable`, вы все равно не сможете воспользоваться всеми его возможностями, если забудете вызвать `register_class()` после тела инструкции `class`. Этот

подход чреват ошибками и особенно труден для начинающих. Похожие риски могут возникать также в случае *декораторов классов* в Python 3.

Что если бы вы могли каким-то образом воздействовать на намерения программиста в отношении использования класса `BetterSerializable` и гарантировать, что `register_class()` вызывается во всех случаях? Метаклассы делают это возможным, перехватывая инструкцию `class` при определении подклассов (см. раздел "Рекомендация 33. Верификация подклассов с помощью метаклассов"). Это позволяет регистрировать новый тип сразу же после тела класса.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(BetterSerializable,
                             metaclass=Meta):
    pass
```

Когда мы определяем подкласс `RegisteredSerializable`, мы можем быть уверенными в том, что вызов `register_class()` будет обязательно выполнен и десериализация всегда будет работать так, как ожидается.

```
class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z

v3 = Vector3D(10, -7, 3)
print('До: ', v3)
data = v3.serialize()
print('Сериализованный объект:', data)
print('После: ', deserialize(data))

>>>
До: Vector3D(10, -7, 3)
Сериализованный объект: {"class": "Vector3D", "args": [10, -7, 3]}
После: Vector3D(10, -7, 3)
```

Использование метаклассов для регистрации классов гарантирует, что вы никогда не пропустите класс, если дерево наследования построено правильно. Этот механизм хорошо работает для сериализации, как это было уже продемонстрировано, и применим к технологии *объектно-реляционных отображений* (ORM) при работе с базами данных, подключаемым системам и системным перехватчикам.

Что следует запомнить

- Регистрация классов — это полезный шаблон построения модульных программ на языке Python.
- Метаклассы обеспечивают автоматический запуск регистрирующего кода в вашей программе каждый раз при определении производных классов на основе базового класса.
- Использование метаклассов для регистрации классов устраняет ошибки, гарантируя, что вызов регистрирующего кода никогда не будет пропущен.

Рекомендация 35. Аннотирование атрибутов классов с помощью метаклассов

Еще одним полезным следствием использования метаклассов является возможность изменения или аннотирования свойств после того, как класс был определен, но до того, как он будет фактически использоваться. Обычно такой подход применяется совместно с *дескрипторами* (см. раздел "Рекомендация 31. Используйте дескрипторы для повторно используемых методов `@property`"), расширяя возможности анализа их использования внутри классов, в которых они содержатся.

Предположим, например, что вам нужно определить новый класс, представляющий строку вашей пользовательской базы данных. Вы хотели бы, чтобы для каждого столбца таблицы базы данных в вашем классе имелось соответствующее свойство. С этой целью мы определим ниже класс дескриптора, связывающий атрибуты с именами столбцов.

```
class Field(object):
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

Сохранив имя столбца в дескрипторе `Field`, мы можем сохранить всю информацию о состоянии отдельного экземпляра в словаре экземпляра в виде защищенных полей, используя встроенные функции `setattr()` и `getattr()`. На первых порах такой подход представляется более удоб-

ным, чем создание дескрипторов с помощью модуля weakref во избежание утечки памяти.

Определение класса, представляющего строку, требует предоставления имени столбца для каждого атрибута класса.

```
class Customer(object):
    # Атрибуты класса
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')
```

Использование класса не составляет труда. В приведенном ниже примере показано, что дескрипторы Field изменяют словарь экземпляра __dict__ так, как ожидалось.

```
foo = Customer()
print('До:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euclid'
print('После: ', repr(foo.first_name), foo.__dict__)

>>>
До: '' {}
После: 'Euclid' {'_first_name': 'Euclid'}
```

Однако здесь, по всей видимости, выполняется лишняя работа. Мы уже объявили имя поля, когда присваивали сконструированный объект Field атрибуту Customer.first_name в теле инструкции class. Зачем тогда передавать имя поля (в данном случае 'first_name') конструктору Field?

Проблема в том, что порядок операций в определении класса Customer противоположен порядку их чтения слева направо. Прежде всего, конструктор Field вызывается как Field('first_name'). Затем возвращаемое при этом значение присваивается атрибуту Customer.first_name. Поэтому дескриптор Field не может знать заранее, какому атрибуту класса он будет присвоен.

Для устранения имеющейся избыточности мы можем использовать метакласс. Метаклассы позволяют непосредственно перехватывать инструкции class и предпринимать необходимые действия сразу же за телом этой инструкции. В данном случае можно использовать метакласс для автоматического присваивания значений атрибутам Field.name и Field.internal_name дескриптора вместо многократного указания имени поля вручную.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
```

```
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls
```

Ниже мы определим базовый класс, который использует метакласс. Все классы, представляющие строки базы данных, должны наследовать этот класс для гарантии того, что они используют данный метакласс.

```
class DatabaseRow(object, metaclass=Meta):
    pass
```

Для работы с метаклассом дескриптор полей почти не требует изменений. Единственное отличие — это то, что его конструктору не требуется передавать какие-либо аргументы. Вместо этого его атрибуты устанавливаются вышеупомянутым методом `Meta.__new__()`.

```
class Field(object):
    def __init__(self):
        # Эти значения будут присвоены метаклассом
        self.name = None
        self.internal_name = None
    # ...
```

Благодаря использованию метакласса, нового базового класса `DatabaseRow` и нового дескриптора `Field` из определения строки базы данных устраняется имевшаяся ранее избыточность.

```
class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

Новый класс ведет себя точно так же, как и прежний.

```
foo = BetterCustomer()
print('До:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euler'
print('После: ', repr(foo.first_name), foo.__dict__)

>>>
До: '' {}
После: 'Euler' {'_first_name': 'Euler'}
```

Что следует запомнить

- Метаклассы позволяют изменять атрибуты класса до того, как класс будет полностью определен.
- Дескрипторы в сочетании с метаклассами обеспечивают возможность декларативного описания поведения и интроспекцию времени выполнения.
- Использование метаклассов наряду с дескрипторами позволяет избежать как утечки памяти, так и использования модуля `weakref`.

5 Одновременность и параллелизм

Термин *псевдопараллелизм*, или *одновременность выполнения* (concurrency), означает видимое одновременное выполнение нескольких вычислений на одном компьютере. Например, на компьютере с одноядерным центральным процессором (CPU) операционная система может быстро переключаться между программами, активизируя каждую из них в определенные моменты времени.

Параллелизм — это реальное параллельное выполнение нескольких вычислений в одно и то же время. Компьютеры с многоядерными процессорами могут выполнять несколько задач одновременно. Каждое ядро выполняет инструкции отдельной программы, благодаря чему программы могут выполняться в течение одного и того же промежутка времени.

В пределах одной программы псевдопараллелизм упрощает программистам решение некоторых типичных задач. Механизм псевдопараллельного выполнения программ допускает множество различных путей выполнения, которые могут быть объединены для достижения конечного результата таким образом, что создается впечатление, будто программы работают одновременно и независимо друг от друга.

Ключевым отличием истинного параллелизма от псевдопараллелизма является *выигрыш в производительности*. Если программа выполняется параллельно по двум возможным путям, то время, необходимое для выполнения определенного объема вычислений, сокращается в два раза, что равносильно двукратному увеличению производительности. В противоположность этому при псевдопараллелизме программы могут использовать для выполнения тысячи отдельных путей, однако никакого ускорения вычислений наблюдаться не будет.

Python упрощает написание программ, работающих в режиме видимой параллельности. Кроме того, Python может использоваться для выполнения параллельной работы посредством системных вызовов, подпроцессов и C-расширений. Однако организовать истинное параллельное выполнение псевдопараллельного кода на Python крайне затрудни-

тельно. Очень важно понимать, как лучше всего использовать Python в этих двух ситуациях, между которыми существуют тонкие различия.

Рекомендация 36. Использование модуля subprocess для управления дочерними процессами

В Python имеются надежные, прошедшие испытание временем библиотеки для выполнения дочерних процессов и управления ими. Благодаря этому Python отлично подходит для "склейки" различных инструментальных средств, таких как утилиты командной строки. Если существующие сценарии оболочки начинают усложняться, как это часто происходит с течением времени, то будет вполне естественно переписать их на Python в интересах повышения удобочитаемости и легкости сопровождения.

Дочерние процессы, запускаемые в коде Python, способны выполняться параллельно, позволяя вам использовать Python для того, чтобы задействовать все ядра процессора и максимизировать производительность программ. И хотя возможности самого Python могут ограничиваться CPU (раздел "Рекомендация 37. Используйте потоки для блокирования операций ввода-вывода, но не для параллелизма"), его легко применять для координации задач, интенсивно использующих процессорное время.

За многие годы в Python предлагалось немало способов для выполнения подпроцессов, в том числе модули `popen`, `popen2` и `os.exec*`. В современном Python самым лучшим и самым простым способом управления дочерними процессами является использование встроенного модуля `subprocess`.

Запуск дочернего процесса с использованием модуля `subprocess` осуществляется очень просто. Ниже показано, как это делается с помощью конструктора `Popen`. Метод `communicate()` читает выходные данные дочернего процесса и ожидает прекращения его выполнения.

```
proc = subprocess.Popen(
    ['echo', 'Привет из дочернего элемента!'],
    stdout=subprocess.PIPE)
out, err = proc.communicate()
print(out.decode('utf-8'))

>>>
Hello from the child!
```

Дочерние процессы выполняются независимо от своего родительского процесса — интерпретатора Python. Их состояние можно периодически опрашивать, пока Python выполняет другую работу.

```
proc = subprocess.Popen(['sleep', '0.3'])
while proc.poll() is None:
    print('Выполнение...')
    # Некие трудоемкие операции
    # ...
print('Код выхода', proc.poll())

>>>
Выполнение...
Выполнение...
Код выхода 0
```

Разрыв связи между дочерним и родительским процессами означает, что родительский процесс может беспрепятственно запускать множество параллельных дочерних процессов. Вы можете это сделать, запустив сначала дочерние процессы все вместе.

```
def run_sleep(period):
    proc = subprocess.Popen(['sleep', str(period)])
    return proc

start = time()
procs = []
for _ in range(10):
    proc = run_sleep(0.1)
    procs.append(proc)
```

Далее можно использовать метод `communicate()` для того, чтобы дождаться завершения операций ввода-вывода процессов и прекращения их выполнения.

```
for proc in procs:
    proc.communicate()
end = time()
print('Завершено за %.3f секунд' % (end - start))

>>>
Завершено за 0.117 секунд
```

Примечание

Если бы эти процессы выполнялись последовательно, то общее время задержки составило бы 1 секунду, а не ~0,1 секунды.

Вы также можете отправлять данные из своей программы на Python в подпроцесс и получать его вывод. Это позволяет использовать другие программы для параллельной работы. Предположим, например, что вы хотите использовать средство командной строки `openssl` для шифрова-

ния определенных данных. Запустить дочерний процесс с аргументами командной строки и каналами ввода-вывода не составляет труда.

```
def run_openssl(data):
    env = os.environ.copy()
    env['password'] = b'\xe24U\n\xd0Ql3S\x11'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
proc.stdin.write(data)
proc.stdin.flush() # Убедиться в получении дочерним
                   # процессом всего ввода
return proc
```

Ниже показано, как в функцию шифрования перенаправляются случайные байты, но на практике это был бы пользовательский ввод, дескриптор файла, сетевой сокет и т.п.

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_openssl(data)
    procs.append(proc)
```

Дочерние процессы будут выполняться параллельно и получать свой ввод. А теперь дожидаемся завершения их выполнения и получаем их окончательный вывод.

```
for proc in procs:
    out, err = proc.communicate()
    print(out[-10:])

>>>
b'o4,G\x91\x95\xfe\xa0\xaa\xb7'
b'\x0b\x01\\\xb1\xb7\xfb\xb2C\xe1b'
b'ds\xc5\xf4;j\x1f\xd0c-'
```

Вы также можете создавать цепочки параллельных процессов по аналогии с программными каналами UNIX, направляя вывод одного дочернего процесса на вход другого и т.д. Вот пример функции, запускающей дочерний процесс, который организует перенаправление входного потока на утилиту командной строки `md5`.

```
def run_md5(input_stdin):
    proc = subprocess.Popen(
        ['md5'],
        stdin=input_stdin,
```

```
        stdout=subprocess.PIPE)
    return proc
```

Примечание

Встроенный модуль Python `hashlib` предоставляет функцию `md5()`, поэтому выполнение подпроцесса, как показано выше, не всегда необходимо. Целью автора было лишь продемонстрировать, как подпроцессы могут перенаправлять входные и выходные потоки.

Далее мы можем запустить группу процессов `openssl` для шифрования определенных данных и другой набор процессов для хеширования данных, зашифрованных с помощью функции `md5()`.

```
input_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_openssl(data)
    input_procs.append(proc)
    hash_proc = run_md5(proc.stdout)
    hash_procs.append(hash_proc)
```

Операции ввода-вывода между дочерними процессами будут осуществляться автоматически, как только процессы будут запущены. Все, что вам остается сделать, — это дождаться их завершения и вывести на печать окончательный результат.

```
for proc in input_procs:
    proc.communicate()
for proc in hash_procs:
    out, err = proc.communicate()
print(out.strip())

>>>
b'7a1822875dcf9650a5a71e5e41e77bf3'
b'd41d8cd98f00b204e9800998ecf8427e'
b'1720f581cfdc448b6273048d42621100'
```

Если вас беспокоит возможность того, что дочерние процессы могут никогда не завершиться или каким-либо образом заблокировать входные или выходные каналы, передайте методу `communicate()` параметр `timeout`. В этом случае, если дочерний процесс не отвечает на запросы в течение заданного промежутка времени, будет сгенерировано исключение, что позволяет прервать выполнение процесса с аномальным поведением.

```
proc = run_sleep(10)
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Код выхода', proc.poll())
>>>
Код выхода -15
```

К сожалению, параметр `timeout` доступен лишь в Python 3.3 и более поздних версиях. В более ранних версиях Python для введения таймаутов при выполнении операций ввода-вывода вам придется использовать модуль `select` в каналах `proc.stdin`, `proc.stdout` и `proc.stderr in`.

Что следует запомнить

- Используйте модуль `subprocess` для выполнения дочерних процессов и управления их потоками ввода-вывода.
- Дочерние процессы выполняются параллельно с интерпретатором Python, что позволяет максимально использовать возможности процессора.
- Используйте параметр `timeout` с методом `communicate()` для того, чтобы избежать блокировки и зависания дочерних процессов.

Рекомендация 37. Используйте потоки для блокирования операций ввода-вывода, но не для параллелизма

Стандартная реализация Python носит название CPython. CPython выполняет программы на языке Python в два этапа: сначала он анализирует и компилирует исходный код в байт-код, а затем выполняет байт-код, используя интерпретатор на стековой основе. Интерпретатор байт-кода имеет состояние, которое должно поддерживаться и быть когерентным во время выполнения программы. Python усиливает когерентность с помощью механизма *глобальной блокировки интерпретатора* (Global Interpreter Lock, GIL).

По существу, GIL — это взаимоисключающая блокировка (мьютекс), которая защищает Python от воздействия вытесняющей (приоритетной) многопоточности, когда один поток берет на себя управление программой, прерывая другой поток. Такое прерывание, если оно происходит в неподходящее время, может разрушить состояние интерпре-

татора. GIL предотвращает подобные прерывания и гарантирует, что каждая инструкция байт-кода будет работать корректно с реализацией CPython и ее модулями C-расширений.

Вместе с тем GIL сопровождается важным отрицательным побочным эффектом. В случае программ, написанных на таких языках, как C++ или Java, наличие нескольких потоков выполнения означает, что ваша программа может использовать несколько ядер CPU в одно и то же время. Несмотря на то что Python поддерживает многопоточность, использование GIL приводит к тому, что только один из потоков может выполняться в каждый отдельный момент времени. Это означает, что если вы прибегаете к потокам, рассчитывая организовать параллельные вычисления и ускорить работу своей программы, то будете сильно разочарованы.

Предположим, например, что вам нужно выполнить интенсивные вычисления. В качестве примера воспользуемся вычислением факториалов натуральных чисел.

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

Последовательная факторизация ряда чисел потребует довольно длительного времени.

```
numbers = [2139079, 1214759, 1516637, 1852285]
start = time()
for number in numbers:
    list(factorize(number))
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 1.040 секунд
```

В других языках использование многопоточности в данном случае имело бы смысл, поскольку это позволило бы задействовать все ядра CPU вашего компьютера. Попробуем сделать то же самое в Python. Ниже мы определим поток Python для выполнения тех же вычислений.

```
from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number
```

```
    def run(self):
        self.factors = list(factorize(self.number))
```

А затем запустим потоки для параллельного вычисления факториала каждого из заданных чисел.

```
start = time()
threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)
```

Наконец, ожидаем завершения выполнения всех потоков.

```
for thread in threads:
    thread.join()
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 1.061 секунд
```

Удивительно то, что для этого расчета потребовалось еще больше времени, чем при последовательном вычислении факториалов. В других языках можно было бы рассчитывать на ускорение вычислений чуть менее, чем в четыре раза, поскольку всегда существуют накладные расходы, связанные с созданием потоков и их координацией. На компьютере автора с двухъядерным процессором, который был использован для этих вычислений, можно было ожидать двукратного ускорения. Поэтому тот факт, что многопоточные вычисления при наличии двух ядер привели к снижению производительности, кажется совершенно неожиданным. Это демонстрирует влияние GIL на программы, выполняющиеся в стандартном интерпретаторе CPython.

Существуют способы заставить CPython взаимодействовать с несколькими ядрами, но они не работают со стандартным классом `Thread` (раздел "Рекомендация 41. Старайтесь использовать модуль `concurrent.futures` для обеспечения истинного параллелизма") и могут требовать значительных ресурсов. Зная об этих ограничениях, вы, вероятно, удивляетесь, зачем вообще Python поддерживает потоки? На это есть две веские причины.

Во-первых, с многопоточностью легче организовать программу так, чтобы создавалось впечатление, будто она одновременно выполняет несколько задач. Управлять одновременным выполнением нескольких задач самостоятельно очень трудно (см., например, раздел "Рекомендация 40. Используйте сопрограммы для одновременного выполнения не-

скольких функций"). Привлекая многопоточность, вы можете поручить Python псевдопараллельное выполнение функций. Такой подход работает, поскольку CPython гарантирует равноправный доступ потоков к процессору, пусть даже GIL предоставляет возможность выполняться в каждый отдельный промежуток времени поочередно только одному из них.

Второй причиной, по которой Python поддерживает потоки, является блокирующий ввод-вывод, что происходит в тех случаях, когда Python выполняет системные вызовы определенного типа. Через механизм системных вызовов программа на языке Python запрашивает взаимодействие операционной системы с внешним окружением от вашего имени. Блокирующий ввод-вывод включает такие операции, как чтение и запись файлов, взаимодействие с сетями, обмен данными с такими устройствами, как дисплеи, и т.д. Потоки помогают обрабатывать операции блокирующего ввода-вывода путем изоляции программы на то время, которое требуется операционной системе для того, чтобы ответить на ваши запросы.

Предположим, например, что вы хотите послать сигнал дистанционно управляемому дрону через последовательный порт. Мы будем использовать медленный системный вызов (`select`) в качестве посредника для выполнения этой операции. Приведенная ниже функция запрашивает у операционной системы блокирование программы на 0,1 секунды с последующим возвратом ей управления аналогично тому, как это происходило бы в случае использования синхронного последовательного порта.

```
import select, socket

def slow_systemcall():
    select.select([socket.socket()], [], [], 0.1)
```

С увеличением количества последовательно выполняемых системных вызовов общее время выполнения увеличивается по линейному закону.

```
start = time()
for _ in range(5):
    slow_systemcall()
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 0.503 секунд
```

Проблема в том, что в то время, пока выполняется функция `slow_systemcall()`, в программе ничего не может происходить. Основной поток выполнения программы блокируется на системном вызове `select()`. На практике такая ситуация может иметь катастрофические последствия. В то время, когда вы отправляете сигнал, вы должны иметь возможность рассчитать дальнейшие действия дрона, иначе он разобьется. Как только вы оказываетесь в ситуации, когда требуется одновременно выполнять операции блокирующего ввода-вывода и проводить вычисления, постарайтесь переместить системные вызовы в другие потоки.

Ниже показано выполнение ряда вызовов функции `slow_systemcall()` в отдельных потоках. Это позволяет связываться одновременно с несколькими портами (и дронами), оставляя основной поток свободным для выполнения необходимых вычислений.

```
start = time()
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

Запустив потоки, можем выполнять расчет дальнейшего движения дрона, пока не дождемся завершения выполнения потоков, связанных с системными вызовами.

```
def compute_helicopter_location(index):
    # ...

for i in range(5):
    compute_helicopter_location(i)
for thread in threads:
    thread.join()
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 0.102 секунд
```

При параллельной организации работы времени потребовалось в 5 раз меньше, чем при последовательной. Это демонстрирует, что все системные вызовы выполняются параллельно в разных потоках, несмотря на ограничения, налагаемые GIL. GIL препятствует параллельному выполнению пользовательского когда на Python, однако не оказывает отрицательного влияния на системные вызовы. Это работает, поскольку потоки Python освобождают GIL непосредственно перед осуществле-

нием системных вызовов и заново захватывают GIL, как только эти вызовы будут выполнены.

Кроме потоков, существует много других способов для работы с блокирующим вводом-выводом, например встроенный модуль `asyncio`, и эти альтернативные способы обладают важными преимуществами. Однако данные варианты требуют проведения дополнительной работы по рефакторингу кода с целью адаптации к другой модели выполнения (раздел "Рекомендация 40. Используйте сопрограммы для одновременного выполнения нескольких функций"). Использование потоков — это простейший способ блокирования операций ввода-вывода, требующий внесения минимальных изменений в вашу программу.

Что следует запомнить

- Параллельное выполнение байт-кода в потоках на нескольких ядрах CPU невозможно ввиду глобальной блокировки интерпретатора (GIL).
- Потоки выполнения Python остаются полезными, несмотря на GIL, поскольку они обеспечивают простой способ организации псевдоодновременного выполнения нескольких вычислений.
- Используйте потоки Python для параллельного выполнения нескольких системных вызовов. Это позволяет выполнять операции блокирующего ввода-вывода одновременно с вычислениями.

Рекомендация 38. Используйте класс `Lock` для предотвращения гонки данных в потоках

Узнав о глобальной блокировке интерпретатора (GIL) (см. предыдущую рекомендацию), многие новички-программисты подумают, что они могут вообще отказаться от использования взаимоисключающих блокировок (мьютексов) в своем коде. Если уж GIL препятствует параллельному выполнению потоков Python на нескольких ядрах CPU, то он должен блокировать и структуры данных, не так ли? Тестирование на некоторых типах, таких как списки и словари, может даже показать, что такое допущение не лишено оснований.

Но будьте осторожны, ведь на самом деле это не совсем так: GIL не защитит вас. Несмотря на то что в каждый момент времени может выполняться только один поток Python, операции, выполняемые потоком над структурой данных, могут прерываться между любыми двумя инструкциями байт-кода в интерпретаторе Python. Это небезопасно, если к одним и тем же объектам обращаются одновременно несколько потоков.

Инвариантность ваших структур данных может быть нарушена практически в любой момент времени прерываниями, которые могут оставить вашу программу в поврежденном состоянии.

Предположим, например, что вы хотите написать программу для выполнения множества параллельных подсчетов, например записывать данные об уровне освещенности, поступающие от целой сети светочувствительных датчиков. Если вы хотите определить общее количество световых проб за определенное время, то должны накапливать эти данные в новом классе.

```
class Counter(object):
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

Представьте, что каждый датчик имеет собственный рабочий поток, поскольку считывание данных требует выполнения операций блокирующего ввода-вывода. Каждый раз после получения очередных данных от датчика счетчик увеличивается на 1, пока не будет достигнуто максимально допустимое количество считываний.

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # Чтение данных, поступающих от датчика
        # ...
        counter.increment(1)
```

Ниже мы определим функцию, которая запускает рабочий поток для каждого датчика и дожидается, пока от них не будет получено установленное количество данных.

```
def run_threads(func, how_many, counter):
    threads = []
    for i in range(5):
        args = (i, how_many, counter)
        thread = Thread(target=func, args=args)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
```

Запуск пяти потоков на выполнение осуществляется очень просто, и конечный результат должен быть очевиден.

```
how_many = 10**5
counter = Counter()
```

```
run_threads(worker, how_many, counter)
print('Ожидаемое значение счетчика: %d, фактическое: %d' %
      (5 * how_many, counter.count))

>>>
Ожидаемое значение счетчика: 500000, фактическое: 278328
```

Однако это совершенно не тот результат! Что случилось? Как в такой простой задаче что-то могло пойти не так, особенно если учесть, что только один поток интерпретатора Python может выполняться в каждый момент времени?

Интерпретатор Python соблюдает принцип равноправия всех выполняющихся потоков, гарантируя, что каждый из них получает примерно одинаковое количество процессорного времени. Для этого Python поочередно приостанавливает выполнение одного потока и возобновляет выполнение другого. Проблема в том, что вы не знаете точно, когда именно Python приостанавливает ваши потоки. Поток может быть приостановлен посреди операции, которая могла бы рассматриваться как атомарная. В данном случае именно это и происходило.

Метод `increment()` объекта `Counter` выглядит очень просто.

```
counter.count += offset
```

Однако применение оператора `+=` к атрибуту объекта в действительности связано с тремя отдельными операциями, выполняемыми интерпретатором Python "за кулисами". Приведенная выше инструкция эквивалентна следующим операциям.

```
value = getattr(counter, 'count')
result = value + offset
setattr(counter, 'count', result)
```

Выполнение потоков Python, инкрементирующих значения счетчиков `counter`, может приостанавливаться между любыми двумя из этих операций. Это может стать проблемой, если перемежающийся характер выполнения операций приводит к тому, что счетчику присваиваются старые версии `value`. Вот пример плохой организации взаимодействия двух потоков, A и B.

```
# Выполнение в потоке A
value_a = getattr(counter, 'count')
# Переключение к контексту потока B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Обратное переключение к контексту потока A
```

```
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

Поток A вытеснил поток B, уничтожая его данные в counter. Именно это и произошло в приведенном выше примере с датчиками освещенности.

Встроенный модуль threading Python включает мощный набор средств, предназначенных для предотвращения состояния гонки (гонки данных) и других форм повреждения структур данных. Простейшее и наиболее употребительное из них — это класс Lock, представляющий взаимоисключающую блокировку (мьютекс).

Использование блокировки позволяет обеспечить защиту текущего значения класса Counter от одновременного доступа к нему нескольких потоков. В каждый момент времени блокировку может захватить только один поток. Ниже для захвата и освобождения блокировки мы будем использовать инструкцию with; это упрощает определение того, какой именно код выполняется во время удерживания блокировки (более подробно об этом говорится в разделе "Рекомендация 43. Обеспечивайте возможность повторного использования блоков try/finally с помощью инструкций contextlib и with").

```
class LockingCounter(object):
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

Теперь можно выполнить рабочие потоки, как и прежде, но с использованием класса LockingCounter.

```
counter = LockingCounter()
run_threads(worker, how_many, counter)
print('Ожидаемое значение счетчика: %d, фактическое: %d' %
      (5 * how_many, counter.count))
>>>
Ожидаемое значение счетчика: 500000, фактическое: 500000
```

Полученный результат совпадает с тем, который ожидался. Проблема была решена с помощью класса Lock.

Что следует запомнить

- Несмотря на то что в Python предусмотрена глобальная блокировка интерпретатора, ответственность за предотвращение возник-

новения состояния гонки между потоками в программе возлагается на вас.

- Предоставление потокам возможности изменять одни и те же объекты без применения блокировок приведет к повреждению структур данных вашей программы.
- Класс `Lock`, включенный во встроенный модуль `threading`, является стандартным средством Python, реализующим взаимоисключающую блокировку.

Рекомендация 39. Использование очередей для координации работы потоков

В программах на Python, в которых выполняется много одновременных вычислений, часто возникает необходимость в координации работы потоков. Одним из наиболее полезных способов организации одновременной работы потоков является конвейер функций.

Работа конвейеров напоминает работу сборочных производственных линий. Конвейеры объединяют множество последовательных фаз, каждая из которых выполняет специфические функции. В начало конвейера постоянно добавляются новые порции данных для обработки. Каждая функция может использовать псевдопараллельный режим для выполнения части работы на протяжении своей фазы. Обрабатываемые данные последовательно передаются от одной функции к другой, пока не будут пройдены все фазы обработки. Такой подход особенно подходит для выполнения работы, включающей операции блокирующего ввода-вывода или подпроцессы, т.е. виды операций, допускающих параллельную обработку с помощью Python (см. раздел "Рекомендация 37. Используйте потоки для блокирования операций ввода-вывода, но не для параллелизма").

Предположим, например, что вы хотите создать систему, которая принимает постоянный поток изображений от цифровой камеры, изменяет их размеры, а затем помещает в фотогалерею на сайте. Конвейер для такой программы может быть разбит на три фазы. На первой фазе получаем новые изображения. На второй фазе загруженные изображения пропускаются через функцию, изменяющую их размеры. На последней фазе обработанные изображения выгружаются на сайт.

Представьте, что вы уже написали функции Python, выполняющие описанные фазы: `download`, `resize`, `upload`. Как вам создать конвейер (`pipeline`) для одновременного выполнения всех видов обработки?

Первое, что вам нужно, — это способ передачи работы между фазами конвейера. Эту задачу можно смоделировать в рамках потокобезопас-

ной модели "производитель–потребитель" (см. предыдущую рекомендацию, чтобы понять важность потокобезопасности в Python; обратитесь к разделу "Рекомендация 46. Используйте встроенные алгоритмы и структуры данных" для получения информации о классе deque).

```
class MyQueue(object):
    def __init__(self):
        self.items = deque()
        self.lock = Lock()
```

Производитель (ваша цифровая камера) добавляет новые изображения в конец списка изображений, подлежащих обработке.

```
def put(self, item):
    with self.lock:
        self.items.append(item)
```

Потребитель, первая фаза вашего конвейера обработки, удаляет изображения, находящиеся в начале упомянутого списка.

```
def get(self):
    with self.lock:
        return self.items.popleft()
```

Ниже мы представим каждую фазу конвейера как поток Python, который получает работу из одной очереди, наподобие этой, выполняет соответствующую функцию и помещает результат в другую очередь. Мы также отследим, сколько раз рабочий поток проверял наличие нового ввода и какой объем работы выполнен.

```
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

Нюанс состоит в том, что рабочий поток должен правильно обрабатывать случаи, когда входная очередь пуста, поскольку предыдущая фаза еще не завершила свою работу. Ниже мы учтем это, перехватывая исключение IndexError. Можете считать это аналогом приостановки сборочной линии.

```
    def run(self):
        while True:
            self.polled_count += 1
            try:
```

```
            item = self.in_queue.get()
        except IndexError:
            sleep(0.01) # Отсутствие данных для обработки
        else:
            result = self.func(item)
            self.out_queue.put(result)
            self.work_done += 1
```

Теперь можно объединить все три фазы путем создания очередей для точек их координации и соответствующих рабочих потоков.

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

Мы можем запустить потоки, а затем передать первой фазе конвейера некоторый объем работы. Ниже мы будем использовать простой экземпляр `object` в качестве заместителя реальных данных, которые требуются функции `download()`.

```
for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
```

Теперь дожидаемся, когда все элементы данных будут обработаны конвейером, и заканчиваем очередью `done_queue`.

```
while len(done_queue.items) < 1000:
    # Выполнить полезную работу во время ожидания
    # ...
```

Этот код работает корректно, но имеется один интересный побочный эффект, порождаемый потоками, которые опрашивают свои входные очереди относительно наличия новой работы. Часть кода, в которой мы перехватываем исключения `IndexError` в методе `run()`, выполняется большое количество раз.

```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print('Обработано', processed, 'элементов после',
      polled, 'опросов')
```

```
>>>
Обработано 1000 элементов после 3030 опросов
```

Когда рабочие функции выполняются с различной скоростью, ранняя фаза может задержать выполнение более поздней, создавая затор в конвейере. Это приводит к так называемому *голоданию* более поздних фаз, которые начинают интенсивно опрашивать в цикле свои входные потоки в ожидании новых порций работы. В результате рабочие потоки напрасно расходуют драгоценное процессорное время, не совершая никаких полезных действий (они постоянно генерируют и перехватывают исключения `IndexError`).

Но это лишь один из недостатков данной реализации. Существуют еще три проблемы, с которыми также необходимо бороться. Во-первых, проверка того, что все имеющиеся на входе порции работы выполнены, требует еще одного активного ожидания в очереди `done_queue`. Во-вторых, метод `run()` класса `Worker` будет бесконечно долго выполняться в своем цикле активного ожидания. Не существует способа послать рабочему потоку сигнал о том, что настало время завершить выполнение.

В-третьих, самая серьезная проблема — это то, что заторы в конвейере могут привести к краху программы. Если первая фаза отрабатывает быстро, а вторая медленно, то очередь, соединяющая первую и вторую фазы, будет все время расти.

Вторая фаза не будет иметь возможности сравняться с первой по скорости работы. По прошествии достаточно длительного времени и при наличии больших объемов входных данных программе в конечном счете не хватит памяти, и она аварийно завершится.

Следующий отсюда вывод вовсе не говорит о том, что конвейеры — это плохо; он лишь указывает на то, что самостоятельное создание надежной очереди "производитель–потребитель" — нелегкая задача.

На помощь приходит класс `Queue`

Вся функциональность, необходимая вам для преодоления всех вышеперечисленных проблем, предоставляется классом `Queue` из встроенного модуля `queue`.

Класс `Queue` устраняет активное ожидание в рабочем потоке, блокируя метод `get()` до тех пор, пока не будут доступны новые данные. Например, запустим поток, ожидающий входные данные из очереди.

```
from queue import Queue
queue = Queue()

def consumer():
    print('Потребитель ожидает данные')
```

```
    queue.get() # Выполняется после метода put() (см. ниже)
    print('Потребитель завершил обработку данных')

thread = Thread(target=consumer)
thread.start()
```

Несмотря на то что этот поток запускается первым, он не завершится до тех пор, пока элемент данных не будет помещен в экземпляр Queue и методу get() будет что возвратить.

```
print('Производитель поставляет данные')
queue.put(object()) # Выполняется перед методом get() (см. выше)
thread.join()
print('Производитель завершил поставку данных')

>>>
Потребитель ожидает данные
Производитель поставляет данные
Потребитель завершил обработку данных
Производитель завершил поставку данных
```

Чтобы разрешить проблему с заторами в конвейере, класс Queue позволяет задать максимальный объем данных, подлежащих обработке, накопление которого между фазами вы считаете допустимым. Определение размера этого буфера приводит к тому, что вызовы метода put() блокируются, если очередь уже заполнена. Например, ниже будет определен поток, который ожидает в течение некоторого времени, прежде чем получить данные из очереди.

```
queue = Queue(1) # Размер буфера 1

def consumer():
    time.sleep(0.1) # Ожидание
    queue.get() # Выполняется вторым
    print('Потребитель завершил обработку данных 1')
    queue.get() # Выполняется четвертым
    print('Потребитель завершил обработку данных 2')

thread = Thread(target=consumer)
thread.start()
```

Ожидание должно позволить потоку производителя поместить с помощью метода put() оба объекта в очередь, прежде чем поток потребителя вызовет метод get(). Однако размер буфера Queue равен 1. Это означает, что производитель, добавляющий элементы в очередь, должен будет ждать, пока поток потребителя вызовет метод get() по крайней

мере однажды, прежде чем второй вызов метода put() прекратит блокировку и добавит второй элемент в очередь.

```
queue.put(object()) # Выполняется первым
print('Производитель поставил данные 1')
queue.put(object()) # Выполняется третим
print('Производитель поставил данные 2')
thread.join()
print('Производитель завершил поставку данных')

>>>
Производитель поставил данные 1
Потребитель завершил получение данных 1
Производитель поставил данные 2
Потребитель завершил получение данных 2
Производитель завершил поставку данных
```

Кроме того, класс Queue может отслеживать прогресс выполнения работы с помощью метода task_done. Это позволяет вам дожидаться, пока входная очередь фазы очистится, и устраняет необходимость в опросе очереди done_queue в конце вашего конвейера. Например, ниже мы определим поток потребителя, который вызывает метод task_done(), когда заканчивает обработку элемента.

```
in_queue = Queue()

def consumer():
    print('Потребитель ожидает получения данных')
    work = in_queue.get() # Выполняется вторым
    print('Потребитель обрабатывает данные')
    # Выполнение работы
    # ...
    print('Потребитель завершил обработку данных')
    in_queue.task_done() # Выполняется третим

Thread(target=consumer).start()
```

Теперь код производителя не должен соединяться с потоком потребителя или выполнять опрос. Производитель может просто ждать, пока входная очередь in_queue завершит работу, вызвав метод join() для экземпляра Queue. Даже когда входная очередь станет пустой, соединение с объектом in_queue будет невозможным до тех пор, пока метод task_done() не будет вызван для каждого элемента из тех, которые были когда-либо помещены в очередь.

```
in_queue.put(object()) # Выполняется первым
print('Производитель выжидает поставлять данные')
```

```
in_queue.join() # Выполняется четвертым
print('Производитель завершил поставку данных')

>>>
Потребитель ожидает получения данных
Производитель выжидает поставлять данные
Потребитель обрабатывает данные
Потребитель завершил обработку данных
Производитель завершил поставку данных
```

Мы можем поместить все эти виды поведения вместе в подкласс `Queue`, который также будет сообщать рабочему потоку, когда следует прекратить обработку. Ниже мы определим метод `close()`, добавляющий в очередь специальный элемент, который указывает, что после него больше не будет никаких входных элементов.

```
class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)
```

Затем определим для очереди итератор, который ищет этот специальный элемент и прекращает итерирование, когда находит его. Этот метод `__iter__()` также вызывает метод `task_done()` в соответствующие моменты времени, позволяя отслеживать продвижение выполнения работы в очереди.

```
    def __iter__(self):
        while True:
            item = self.get()
            try:
                if item is self.SENTINEL:
                    return # Приводит к завершению потока
                yield item
            finally:
                self.task_done()
```

Теперь можно переопределить наш рабочий поток, чтобы он опирался на поведение класса `ClosableQueue`. Поток завершится после прохождения цикла `for`.

```
class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        # ...

    def run(self):
        for item in self.in_queue:
```

```
            result = self.func(item)
            self.out_queue.put(result)
```

Заново создадим набор рабочих потоков, используя новый класс.

```
download_queue = ClosableQueue()
# ...
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    # ...
]
```

После запуска рабочих потоков, как мы делали это раньше, также отправим сигнал паузы, как только все входные данные будут переданы для обработки конвейеру путем закрытия входной очереди первой фазы.

```
for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
download_queue.close()
```

Наконец, дожидаемся завершения работы путем присоединения к каждой очереди, соединяющей фазы. Каждый раз, когда завершается выполнение одной фазы, мы сигнализируем следующей фазе о необходимости приостановиться путем закрытия ее входной очереди. По завершении обработки в очереди done_queue содержатся все выходные объекты, как и ожидалось.

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'элементов обработано')

>>>
1000 элементов обработано
```

Что следует запомнить

- Конвейеризация многостадийных задач — замечательный способ одновременного выполнения последовательных стадий за счет использования нескольких потоков Python.
- Не упускайте из виду проблемы, которые могут подстерегать вас при построении многопоточных конвейеров: состояния активного

ожидания, остановка рабочих потоков и взрывной характер утечки памяти.

- В классе `Queue` предусмотрены все средства, которые могут потребоваться вам для построения надежных конвейеров: блокирующие операции, регулируемый размер буфера и присоединение к потоку в ожидании его завершения.

Рекомендация 40. Используйте сопрограммы для одновременного выполнения нескольких функций

Потоки предоставляют программистам возможность выполнять сразу несколько функций (см. раздел "Рекомендация 37. Используйте потоки для блокирования операций ввода-вывода, но не для параллелизма"). Но с потоками связаны три большие проблемы.

- Безопасная координация их взаимодействия требует использования специальных средств (см. разделы "Рекомендация 38. Используйте класс `Lock` для предотвращения гонки данных в потоках" и "Рекомендация 39. Использование очередей для координации работы потоков"). Из-за этого анализировать код, в котором используются потоки, труднее, чем процедурный однопоточный код. Следствием этого являются трудности расширения и последующего сопровождения кода.
- С потоками связан большой расход памяти: около 8 Мбайт на один поток выполнения. Для многих компьютеров, если количество потоков исчисляется десятками, такие объемы памяти не являются существенными. Но что если вы захотите, чтобы ваша программа выполняла тысячи функций "одновременно"? Эти функции могут соответствовать пользовательским запросам к серверу, пикселям на экране, частицам в имитации физических процессов и т.п. Выполнение потока для каждого отдельного вида активности просто не сработает.
- Запуск потоков — дорогостоящее мероприятие. Если вы собираетесь постоянно создавать в программе функции, предназначенные для одновременного выполнения, то накладные расходы, связанные с использованием потоков, значительно замедлят ее работу.

Python позволяет обойти все эти трудности с помощью *сопрограмм*. Сопрограммы обеспечивают возможность видимого одновременного выполнения многих функций в программах на Python. Они реализованы в виде расширения генераторов (см. раздел "Рекомендация 16. Не

упускайте возможность использовать генераторы вместо возврата списков"). Стоимостью запуска сопрограммы-генератора является один вызов функции. После этого каждая из сопрограмм, пока она не завершится, требует менее 1 Кбайт памяти.

Работа сопрограмм основана на использовании генератора и отправке значения в функцию-генератор с помощью метода `send()` после каждого выражения `yield`. Функция-генератор получает значение, переданное функции `send()` в качестве результата вычисления соответствующего выражения `yield`.

```
def my_coroutine():
    while True:
        received = yield
        print('Получено:', received)

it = my_coroutine()
next(it) # Подготовить сопрограмму
it.send('Первый')
it.send('Второй')

>>>
Получено: Первый
Получено: Второй
```

Чтобы подготовить генератор к получению первого значения от метода `send()`, требуется начальный вызов метода `next()`, продвигающего генератор к первому выражению `yield`. Использование пары `yield` и `send()` является стандартным способом, позволяющим изменять следующее значение генератора в ответ на внешний ввод.

Предположим, например, что вы хотите реализовать сопрограмму-генератор, которая выдает минимальное значение из всех тех, которые ей были до сих пор переданы. Ниже ключевое слово `yield`, за которым не следует никакого выражения, подготавливает сопрограмму с начальным минимальным значением, переданным извне. Затем генератор повторно возвращает новое минимальное значение каждый раз после получения очередного входного значения.

Код, в котором используется генератор, может выполняться пошагово и будет выводить результирующее минимальное значение, заново вычисляемое после каждого очередного ввода.

```
it = minimize()
next(it) # Подготовить генератор
print(it.send(10))
print(it.send(4))
print(it.send(22))
```

```
print(it.send(-1))
>>>
10
4
4
-1
```

Эта функция-генератор может работать бесконечно долго, выполняя следующий шаг после каждого вызова метода `send()`. Подобно потокам, сопрограммы — независимые функции, которые могут получать входные данные из их окружения и вырабатывать результирующий вывод. Отличаются они тем, что сопрограммы приостанавливаются на каждом выражении `yield` в функции-генераторе и возобновляют свою работу после каждого вызова метода `send()` извне. В этом и состоит суть магического механизма сопрограмм.

Такое поведение позволяет коду, использующему генератор, предпринимать определенные действия после каждого выражения `yield` в сопрограмме. Код-потребитель может использовать выходные значения генератора для вызова других функций и обновления структур данных. Что немаловажно, он может продвигать другие функции-генераторы к их следующим выражениям `yield`. Обеспечивая продвижение множества отдельных генераторов в строго согласованной манере, они создают видимость одновременного исполнения, имитируя согласованное поведение потоков Python.

Игра "Жизнь"

Позвольте мне продемонстрировать согласованное поведение сопрограмм на конкретном примере. Предположим, вы хотите использовать сопрограммы для реализации игры "Жизнь", придуманной математиком Джоном Конвеем в 1970 году. Правила игры очень просты. Имеется двумерная сетка с ячейками произвольного размера. Каждая ячейка может быть в одном из двух состояний: живой (alive) или пустой (empty).

```
ALIVE = '*'
EMPTY = '-'
```

Игра развивается в соответствии с тактами дискретного времени. С каждым тактом часов для каждой клетки подсчитывается, какое количество соседних с ней клеток (общее количество которых равно 8) находится в живом состоянии. В зависимости от результата для каждой клетки определяется, будет ли она и далее жить, умрет или регенерирует. Ниже приведен пример игры для поля размером `5×5` для четырех новых поколений, развившихся из начального (стрелка времени направ-

лена слева направо). Конкретные правила игры будут разъяснены немного позже.

```
  0   |   1   |   2   |   3   |   4
----- | ----- | ----- | ----- | -----
-*--- | --*-- | --**- | --*-- | -----
--**- | --**- | -*--- | -*--- | -**--
---*- | --**- | --**- | --*-- | -----
----- | ----- | ----- | ----- | -----
```

Можно смоделировать эту игру, представляя каждую ячейку сопрограммой-генератором, выполняемой в строгом согласовании с другими сопрограммами.

Чтобы это реализовать, мне, во-первых, нужен способ извлечения состояния соседних клеток. Это можно сделать с помощью сопрограммы `count_neighbors()`, которая работает, предоставляя объекты `Query` с использованием ключевого слова `yield`. Класс `Query` определим самостоятельно. Он предназначен для того, чтобы предоставить сопрограмме-генератору возможность запрашивать информацию у своего окружения.

```
Query = namedtuple('Query', ('y', 'x'))
```

Сопрограмма возвращает объект `Query` для каждого соседа. Результатом вычисления каждого выражения `yield` будет значение `ALIVE` или `EMPTY`. Это контракт интерфейса, который определен между сопрограммой и использующим ее кодом. Генератор `count_neighbors()` видит состояния соседей и возвращает количество живых соседних клеток.

```
def count_neighbors(y, x):
    n_ = yield Query(y + 1, x + 0) # Север
    ne = yield Query(y + 1, x + 1) # Северо-восток
    # Определить e_, se, s_, sw, w_, nw ...
    # ...
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Мы можем протестировать сопрограмму `count_neighbors()`, используя фиктивные данные. Ниже показано, как будут возвращаться объекты `Query` для каждого соседа конкретной клетки. Генератор `count_neighbors()` ожидает получения состояний клеток, соответствующих каждому объекту `Query`, через метод `send()` сопрограммы. Результат окончательного подсчета возвращается исключением `StopIteration`,

которое генерируется, когда генератор исчерпывается инструкцией return.

```
it = count_neighbors(10, 5)
q1 = next(it) # Получить первый объект query
print('Первый результат: ', q1)
q2 = it.send(ALIVE) # Передать состояние q1, получить q2
print('Второй результат:', q2)
q3 = it.send(ALIVE) # Передать состояние q2, получить q3
# ...
try:
    it.send(EMPTY) # Передать состояние q8, получить count
except StopIteration as e:
    print('Счетчик: ', e.value) # Значение, возвращенное return

>>>
Первый результат: Query(y=11, x=5)
Второй результат: Query(y=11, x=6)
...
Счетчик: 2
```

Теперь мы должны иметь возможность указать, что клетка хочет перейти в новое состояние в соответствии со значением счетчика соседей, возвращенным генератором `count_neighbors()`.

Для этого определим другую сопрограмму под названием `step_cell()`. Этот генератор будет указывать переходы ячеек из одного состояния в другое, возвращая объекты `Transition`. Вот еще один класс, который определим точно так же, как и класс `Query`.

```
Transition = namedtuple('Transition', ('y', 'x', 'state'))
```

Сопрограмма `step_cell()` получает координаты клетки в сетке в виде аргументов. Она возвращает объект `Query` для получения начального состояния клетки с этими координатами и выполняет генератор `count_neighbors()` для проверки состояния окружающих ее клеток. Затем она запускает логику игры для определения того, какое состояние должна иметь данная клетка на следующем такте часов. И наконец, возвращает объект `Transition`, чтобы сообщить окружению, каким будет ее следующее состояние.

```
def game_logic(state, neighbors):
    # ...

def step_cell(y, x):
    state = yield Query(y, x)
    neighbors = yield from count_neighbors(y, x)
```

```
    next_state = game_logic(state, neighbors)
    yield Transition(y, x, next_state)
```

Что немаловажно, вызов `count_neighbors()` осуществляется с использованием выражения `yield from`. Это выражение позволяет Python объединять сопрограммы-генераторы, тем самым упрощая повторное использование небольших функциональных частей и построение сложных сопрограмм из простых. Когда наступает исчерпание генератора `count_neighbors()`, возвращаемое им (с помощью инструкции `return`) конечное значение будет передано генератору `step_cell()` в качестве результата выражения `yield from`.

Наконец, можно определить простую логику для игры "Жизнь" Конвея. Она включает только три правила.

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY # Умереть: слишком мало соседей
        elif neighbors > 3:
            return EMPTY # Умереть: слишком много соседей
    else:
        if neighbors == 3:
            return ALIVE # Восстановиться
    return state
```

Протестируем сопрограмму `step_cell()`, используя фиктивные данные.

```
it = step_cell(10, 5)
q0 = next(it) # Начальный объект query
print('Я: ', q0)
q1 = it.send(ALIVE) # Передать свое состояние,
                    # получить состояние соседа
print('Q1: ', q1)
# ...
t1 = it.send(EMPTY) # Передать q8, получить решение
print('Исход: ', t1)

>>>
Me: Query(y=10, x=5)
Q1: Query(y=11, x=5)
...
Исход: Transition(y=10, x=5, state='-')
```

Цель игры заключается в согласованном выполнении этой логики в отношении всей совокупности клеток. Для этого вставим сопрограмму `step_cell()` в сопрограмму `simulate()`. Эта сопрограмма продвигает игру, многократно возвращая с помощью `yield` значения сопрограммы

step_cell(). После обработки всех координат она возвращает объект TICK, указывая на то, что все клетки текущего поколения изменили свое состояние.

```
TICK = object()

def simulate(height, width):
    while True:
        for y in range(height):
            for x in range(width):
                yield from step_cell(y, x)
        yield TICK
```

В сопрограмме simulate() примечательно то, что она совершенно не связана с окружающей средой. Мы еще даже не определили, как именно сетка должна быть представлена в объектах Python, каким образом осуществляется внешняя обработка значений Query, Transition и TICK и как задается начальное состояние игры. Однако логика игры ясна. Каждая клетка будет переходить из одного состояния в другое, выполняя сопрограмму step_cell(), после чего выполняется один такт часов. Этот процесс может продолжаться бесконечно долго, пока работает сопрограмма simulate().

В том и состоит красота сопрограмм. Они позволяют сосредоточиться на логике, которую вы пытаетесь реализовать, и разрывают связь между инструкциями вашего кода для окружения и реализацией, воплощающей ваши намерения. Это позволяет создавать видимость параллельного выполнения сопрограмм, а также дает возможность со временем усовершенствовать реализацию, не изменяя сопрограммы.

Теперь необходимо реализовать выполнение сопрограммы simulate() в реальном окружении. Для этого нужно представить состояние каждой клетки сетки. Ниже определен класс, который будет содержать сетку.

```
class Grid(object):
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)

    def __str__(self):
        # ...
```

Сетка позволяет получить и задать значение любой координаты. Координаты, выходящие за границу с одной стороны, будут переходить на

другую сторону, превращая сетку в своего рода бесконечное циклическое пространство.

```
def query(self, y, x):
    return self.rows[y % self.height][x % self.width]

def assign(self, y, x, state):
    self.rows[y % self.height][x % self.width] = state
```

Наконец, можно определить функцию, которая интерпретирует значения, вырабатываемые сопрограммой `simulate()` и всеми ее внутренними сопрограммами. Эта функция превращает инструкции в сопрограммах во взаимодействие с окружением. Она изменяет состояние всех клеток как один шаг, а затем возвращает новую сетку, содержащую следующее состояние.

```
def live_a_generation(grid, sim):
    progeny = Grid(grid.height, grid.width)
    item = next(sim)
    while item is not TICK:
        if isinstance(item, Query):
            state = grid.query(item.y, item.x)
            item = sim.send(state)
        else: # Должно быть Transition
            progeny.assign(item.y, item.x, item.state)
            item = next(sim)
    return progeny
```

Чтобы увидеть, как работает эта функция, нужно создать сетку и установить ее начальное состояние. Эти задачи решает приведенный ниже код, в котором "живые" клетки образуют некую фигуру.

```
grid = Grid(5, 9)
grid.assign(0, 3, ALIVE)
# ...
print(grid)

>>>
---*-----
----*----
--***----
---------
---------
```

Теперь можно продвигать игру по одному поколению за один раз. Вы видите, как фигура, следуя простым правилам игры, заложенным в функции `game_logic()`, перемещается по сетке вниз и вправо.

```
class ColumnPrinter(object):
    # ...

columns = ColumnPrinter()
sim = simulate(grid.height, grid.width)
for i in range(5):
    columns.append(str(grid))
    grid = live_a_generation(grid, sim)

print(columns)
```

```
>>>
    0     |     1     |    2      |    3      |    4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Наилучшей стороной описанного подхода является то, что внесение изменений в функцию `game_logic()` не требует обновления окружающего кода. Мы можем изменить правила игры или расширить сферы взаимного влияния клеток при существующем механизме взаимодействия объектов `Query`, `Transition` и `TICK`. Это отчетливо демонстрирует, как сопрограммы обеспечивают возможность разделения задач, что является важным принципом проектирования программ.

Сопрограммы в Python 2

К сожалению, в Python 2 отсутствуют некоторые синтаксические изюминки, которые делают сопрограммы столь элегантными в Python 3. Речь идет о двух ограничениях. Во-первых, отсутствует выражение `yield from`. Это означает, что если вы хотите скомпоновать вместе сопрограммы-генераторы в Python 2, то вам потребуется включить дополнительный цикл в точке делегирования полномочий.

```
# Python 2
def delegated():
    yield 1
    yield 2

def composed():
    yield 'A'
    for value in delegated(): # yield from в Python 3
        yield value
    yield 'B'
```

```
print list(composed())

>>>
['A', 1, 2, 'B']
```

Второе ограничение — это отсутствие поддержки возврата значений из генераторов Python 2. Для получения того же поведения, корректно взаимодействующего с блоками `try/except/finally`, вы должны определить собственный тип исключения и генерировать его, когда хотите возвратить значение.

```
# Python 2
class MyReturn(Exception):
    def __init__(self, value):
        self.value = value

def delegated():
    yield 1
    raise MyReturn(2) # return 2 в Python 3
    yield 'Не достигнуто'

def composed():
    try:
        for value in delegated():
            yield value
    except MyReturn as e:
        output = e.value
    yield output * 4

print list(composed())

>>>
[1, 8]
```

Что следует запомнить

- Сопрограммы обеспечивают эффективный способ видимого одновременного выполнения десятков тысяч функций.
- В генераторе значением выражения `yield` будет любое значение, переданное методу `send()` генератора из внешнего кода.
- Сопрограммы являются мощным инструментом, позволяющим отделять основную логику вашей программы от взаимодействия с окружающим кодом. Выражения `yield from` и возврат значений из генераторов в Python 2 не поддерживаются.

Рекомендация 41. Старайтесь использовать модуль `concurrent.futures` для обеспечения истинного параллелизма

Иногда при написании программ на Python могут возникать нюансы, связанные с проблемами производительности. Даже после оптимизации кода (раздел "Рекомендация 58. Сначала — профилирование, затем — оптимизация") ваша программа может выполняться намного медленнее, чем вам хотелось бы. В случае современных компьютеров, оборудованных многоядерными процессорами, разумно предположить, что одним из возможных решений мог бы быть параллелизм. Что если бы вы могли разбить свой вычислительный код на несколько независимых задач, одновременно выполняющихся на многих ядрах CPU?

К сожалению, глобальная блокировка интерпретатора Python (GIL) препятствует использованию истинного параллелизма в потоках (см. раздел "Рекомендация 37. Используйте потоки для блокирования операций ввода-вывода, но не для параллелизма"), поэтому данный вариант не подходит. Еще одна типичная рекомендация — переписать наиболее критичные в отношении производительности участки кода в виде модуля расширения на языке C. Последний приближает вас к "железу" и может выполняться быстрее, чем Python, устраняя необходимость в параллелизме. Кроме того, C-расширения могут запускать собственные потоки, способные выполняться параллельно и использовать несколько ядер CPU. API для C-расширений, предлагаемый в Python, хорошо документирован и может прийти на помощь в качестве аварийного выхода.

Однако переписывание кода на язык C — занятие дорогостоящее. Код, короткий и понятный на языке Python, после переделки на C может стать слишком детальным и запутанным. Подобное портирование кода требует обширного тестирования, гарантирующего эквивалентность функциональности конечного результата и первоначального варианта и отсутствие внесенных ошибок. Иногда это того стоит, что и объясняет наличие широкой экосистемы модулей C-расширений среди сообщества пользователей Python, поскольку, например, это позволяет ускорить синтаксический анализ кода, обработку изображений и матричные вычисления. Существуют даже такие инструменты с открытым исходным кодом, как Cython (`http://cython.org`) и Numba (`http://numba.pydata.org`), которые могут упростить переход к C.

Суть проблемы заключается в том, что одного только переписывания части кода программы на C чаще всего недостаточно. В оптимизированных программах на Python обычно отсутствует какая-либо одна часть кода, которую можно считать основным виновником медленной работы, и, скорее всего, имеется несколько таких частей. Чтобы в пол-

ной мере воспользоваться преимуществами C в отношении доступа к аппаратным средствам и возможностями потоков, приходится портировать большие куски программы, что порождает новые риски и требует тщательного тестирования конечного кода. Наверняка должен существовать лучший способ сохранения ваших инвестиций в Python для разрешения трудных вычислительных проблем.

Встроенный модуль `multiprocessing`, доступный через встроенный модуль `concurrent.futures`, может предоставить как раз то, что вам нужно. Он позволяет использовать для параллельных вычислений несколько ядер CPU, выполняя несколько интерпретаторов Python в виде дочерних процессов. Эти дочерние процессы, а вместе с ними и их глобальные блокировки интерпретаторов, отделены от основного интерпретатора. Каждый дочерний процесс может полностью задействовать одно ядро CPU и имеет ссылку на основной процесс, откуда он получает инструкции по выполнению вычислений и куда возвращает результаты.

Предположим, например, что вы хотите выполнить некоторые интенсивные вычисления с помощью Python и использовать несколько ядер CPU. В качестве иллюстрации выберем простой алгоритм нахождения наибольшего общего делителя двух чисел. Конечно, это далеко не те интенсивные вычисления, которые, например, приходится выполнять при моделировании потока жидкости с помощью уравнения Навье–Стокса, однако для наших целей этого будет вполне достаточно.

```
def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

При обычном выполнении этой функции, в отсутствие параллелизма, длительность вычислений линейно растет с увеличением объема входных данных.

```
numbers = [(1963309, 2265973), (2030677, 3814172),
           (1551645, 2229620), (2039045, 2020802)]
start = time()
results = list(map(gcd, numbers))
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 1.170 секунд
```

Выполнение этого кода с помощью нескольких потоков Python не приведет к ускорению расчетов, поскольку GIL препятствует параллельному использованию Python нескольких ядер CPU. Ниже мы выполним те же вычисления, но с использованием модуля `concurrent.futures`, входящего в него класса `ThreadPoolExecutor` и двух рабочих потоков (по количеству ядер в CPU моего компьютера)

```
start = time()
pool = ThreadPoolExecutor(max_workers=2)
results = list(pool.map(gcd, numbers))
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 1.199 секунд
```

Этот вариант работает еще медленнее из-за накладных расходов, связанных с обслуживанием пула потоков.

А теперь сюрприз: изменив всего одну строку кода, мы совершим чудо. Если вместо класса `ThreadPoolExecutor` использовать класс `ProcessPoolExecutor` из модуля `concurrent.futures`, то работа программы ускорится.

```
start = time()
pool = ProcessPoolExecutor(max_workers=2) # The one change
results = list(pool.map(gcd, numbers))
end = time()
print('Заняло %.3f секунд' % (end - start))

>>>
Заняло 0.663 секунд
```

Как видите, этот результат, полученный при выполнении программы на моем компьютере с двухъядерным процессором, значительно лучше предыдущих! За счет чего это стало возможным? Вот что на самом деле делает класс `ProcessPoolExecutor` (посредством низкоуровневых конструкций, предоставляемых модулем `multiprocessing`).

1. Передает каждый элемент входных данных из списка `numbers` функции `map()`.
2. Сериализует его в двоичные данные с помощью модуля `pickle` (раздел "Рекомендация 44. Повышайте надежность модуля `pickle` с помощью модуля `copyreg`").
3. Копирует сериализованные данные из процесса основного интерпретатора в процесс дочернего интерпретатора через локальный сокет.

4. Выполняет обратную десериализацию данных в объекты Python, используя модуль `pickle` в дочернем процессе.
5. Импортирует модуль Python, содержащий функцию `gcd()`.
6. Выполняет функцию по отношению к входным данным параллельно с другими дочерними процессами.
7. Сериализует результат в байты.
8. Копирует байты в обратном направлении через сокет.
9. Десериализует байты обратно в объекты Python в родительском процессе.
10. Выполняет слияние всех результатов, предоставленных дочерними процессами, в один список, подлежащий возврату.

Несмотря на то что все эти операции покажутся программисту простыми, модуль `multiprocessing` и класс `ProcessPoolExecutor` проделывают огромный объем работы, чтобы параллелизм стал возможным. В большинстве других языков программирования все, что вам требуется для координации работы двух потоков, — это единственная блокировка или атомарная операция. Использование модуля `multiprocessing` сопряжено с большими накладными расходами, связанными с выполнением операций сериализации и десериализации данных между родительским процессом и дочерними.

Эта схема хорошо приспособлена для некоторых типов изолированных задач, когда объемы данных, которыми обмениваются родительские и дочерние процессы, невелики, однако обработка этих данных требует значительных вычислительных усилий. Под термином "изолированные" я подразумеваю задачи, не нуждающиеся в разделении состояния с другими частями программы. Алгоритм нахождения наибольшего общего делителя является одним из примеров таких задач, но аналогичным образом работают и многие другие алгоритмы.

Если ваши вычисления не соответствуют этим характеристикам, то накладные расходы в связи с использованием модуля `multiprocessing` сведут на нет ускорение работы программы за счет параллелизма. На этот случай в модуле `multiprocessing` предусмотрены более развитые средства, обеспечивающие разделение памяти, межпроцессные блокировки, очереди и прокси-объекты. Однако все эти средства очень сложны. Обсуждать их применительно к пространству памяти одного процесса, совместно используемого несколькими потоками Python, уже довольно сложно. Расширение же этой сложности на дополнительные процессы с привлечением сокетов еще более затрудняет обсуждение данной темы.

Я бы рекомендовал избегать непосредственного использования любых частей модуля `multiprocessing` и вместо этого использовать их через более простой модуль `concurrent.futures`. Можете начать с привлечения класса `ThreadPoolExecutor` для выполнения изолированных задач с небольшим, но требующим интенсивной обработки объемом данных, в потоках. Впоследствии вы сможете ускорить вычисления с помощью класса `ProcessPoolExecutor`. Наконец, после того как вы исчерпаете все другие возможности, вы сможете проанализировать возможность непосредственного использования модуля `multiprocessing`.

Что следует запомнить

- Перенос разрешения проблем, связанных с узкими местами CPU, в модули C-расширений может служить эффективным способом повышения производительности при максимально полном использовании возможностей кода Python. Однако такой подход сопровождается большими накладными расходами и чреват внесением новых ошибок.
- Модуль `multiprocessing` предоставляет мощные инструменты, обеспечивающие параллельное выполнение некоторых типов вычислительных задач в Python с минимальными усилиями.
- К мощным средствам модуля `multiprocessing` лучше всего обращаться через встроенный модуль `concurrent.futures` и его простой класс `ProcessPoolExecutor`.
- Избегайте использования наиболее развитых средств модуля `multiprocessing` ввиду их сложности.

6 Встроенные модули

Поставка стандартной библиотеки в пакете Python следует принципу "батарейки в комплекте". Многие другие языки поставляются с небольшим количеством пакетов общего назначения, требуя от вас самостоятельно проводить поиск важной функциональности. Несмотря на то что Python располагает внушительным репозиторием модулей, созданных усилиями сообщества пользователей, его поставка комплектуется таким образом, чтобы установка по умолчанию включала модули для распространенных областей использования языка.

Полный набор стандартных модулей настолько велик, что его рассмотрение в данной книге не представляется возможным. Однако некоторые из встроенных пакетов настолько переплетаются с идиомами Python, что они вполне могли бы составить часть спецификации языка. Эти жизненно важные встроенные модули приобретают особое значение при написании запутанных, подверженных ошибкам частям программы.

Рекомендация 42. Определяйте декораторы функций с помощью модуля `functools.wraps`

Предположим, например, что вы хотите выводить на печать аргументы и возвращаемое значение функции. Это особенно полезно при отладке стека вызовов из рекурсивной функции. Определим следующий декоратор.

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('%s(%r, %r) -> %r' %
              (func.__name__, args, kwargs, result))
        return result
    return wrapper
```

Мы можем применить этот декоратор к функции, используя символ @.

```
@trace
def fibonacci(n):
    """Возвратить n-е число Фибоначчи"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

Символ @ эквивалентен вызову декоратора для функции, которую он обертывает, и присвоению возвращаемого значения исходному имени в той же области видимости.

```
fibonacci = trace(fibonacci)
```

Вызов этой декорированной функции выполнит код `wrapper` перед тем и после того, как выполнится код функции `fibonacci`, выводя на печать аргументы и возвращая значение на каждом уровне рекурсивного стека.

```
fibonacci(3)


>>>
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((3,), {}) -> 2
```

Этот код работает, но при этом проявляется непредусмотренный побочный эффект. Значению, возвращаемому декоратором, т.е. функции, которая вызывается, неизвестно, что она называется `fibonacci`.

```
print(fibonacci)
>>>
<function trace.<locals>.wrapper at 0x107f7ed08>
```

Усмотреть причину этого не столь уж трудно. Функция `trace()` возвращает функцию `wrapper()`, которую она определяет. Функция `wrapper()` — это то, что присваивается имени `fibonacci` во вмещающем модуле в силу действия декоратора. Такое поведение проблематично, поскольку оно дестабилизирует работу инструментов, выполняющих ретроспекцию, таких как отладчики (раздел "Рекомендация 57. Используйте интерактивную отладку с помощью пакета `pdb`") и сериализаторы объектов (раздел "Рекомендация 44. Повышайте надежность встроенного модуля `pickle` с помощью модуля `copyreg`").

Например, встроенная функция `help()` бесполезна в отношении декорированной функции `fibonacci()`.

```
help(fibonacci)
>>>
Справка по функции wrapper - в модуле __main__:

wrapper(*args, **kwargs)
```

Решение заключается в использовании вспомогательной функции `wraps()` из встроенного модуля `functools`. Это декоратор, упрощающий написание других декораторов. Будучи примененным к функции `wrapper()`, он скопирует все важные метаданные о внутренней функции во внешнюю функцию.

```
def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # ...
    return wrapper

@trace
def fibonacci(n):
    # ...
```

Теперь применение функции `help()` дает желаемый результат даже для декорированной функции.

```
help(fibonacci)
>>>
Справка по функции fibonacci - в модуле __main__:

fibonacci(n)
Возвратить n-е число Фибоначчи
```

Вызов `help()` — это всего лишь один из возможных примеров того, как декораторы могут незаметно порождать проблемы различного рода. Функции Python имеют много других стандартных атрибутов (например, `__name__`, `__module__`), которые должны сохраняться для поддержания интерфейсов функций в языке. Использование вспомогательной функции `wraps()` гарантирует, что вы всегда будете получать корректное поведение.

Что следует запомнить

- ✦ Декораторы — часть синтаксиса Python, позволяющая одной функции модифицировать другую во время выполнения.

- Использование декораторов может приводить к странному поведению инструментов, выполняющих интроспекцию, таких как отладчики.
- Во избежание возможных проблем используйте декораторы из встроенного модуля `functools`.

Рекомендация 43. Обеспечивайте возможность повторного использования блоков `try/finally` с помощью инструкций `contextlib` и `with`

В Python инструкция `with` используется для указания того, что код выполняется в специальном контексте. Например, взаимоисключающие блокировки (см. раздел "Рекомендация 38. Используйте класс `Lock` для предотвращения гонки данных в потоках") могут использоваться в инструкциях `with` для указания того, что код в строках с отступом выполняется лишь тогда, когда удерживается блокировка.

```
lock = Lock()
with lock:
    print('Удержание блокировки')
```

Приведенный пример эквивалентен следующей конструкции `try/finally`, поскольку класс `Lock` надлежащим образом активизирует инструкцию `with`.

```
lock.acquire()
try:
    print('Удержание блокировки')
finally:
    lock.release()
```

Версия этого кода, в которой используется инструкция `with`, лучше, поскольку она устраняет необходимость в написании повторяющегося кода конструкции `try/finally`. С помощью встроенного модуля `contextlib` вы можете легко обеспечить возможность использования своих объектов и функций в инструкциях `with`. Этот модуль содержит декоратор `contextmanager`, который позволяет использовать простую функцию с инструкцией `with`. Это намного проще, чем определять новый класс со специальными методами `__enter__()` и `__exit__()` (стандартный способ).

Предположим, вы хотите, чтобы объем отладочной информации для некоторой части вашего кода зависел от уровня ошибки. Ниже мы опре-

делим функцию, которая выводит информацию для двух уровней ошибок.

```
def my_function():
    logging.debug('Отладочные данные')
    logging.error('Сообщение об ошибке')
    logging.debug('Дополнительные отладочные данные')
```

В нашей программе уровнем ошибок по умолчанию является WARNING, поэтому при выполнении данной функции на экран будет выводиться лишь сообщение об ошибке.

```
my_function()
>>>
Сообщение об ошибке
```

Мы можем временно повысить для этой функции уровень протоколирования ошибок, требующих вывода отчета, определив *менеджер контекста*. Эта вспомогательная функция сначала повышает уровень протоколирования ошибок до выполнения кода в блоке with, а затем вновь понижает его.

```
@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old_level)
```

Выражение yield является тем участком кода, в котором будет выполняться содержимое блока with. Возникающие в блоке with исключения будут повторно генерироваться выражением yield, и их можно перехватывать во вспомогательной функции (более подробно о том, как это работает, см. в разделе "Рекомендация 40. Используйте сопрограммы для одновременного выполнения нескольких функций").

Теперь мы можем вновь вызвать ту же функцию протоколирования, но в контексте debug_logging. На этот раз во время работы блока with на экран будет выводиться вся отладочная информация. При выполнении этой же функции после выхода из блока with никакая отладочная информация не выводится.

```
with debug_logging(logging.DEBUG):
    print('В блоке:')
    my_function()
print('После блока:')
```

```
my_function()

>>>
В блоке:
Отладочные данные
Сообщение об ошибке
Дополнительные отладочные данные
После блока:
Error log here
```

Использование целевых переменных with

Менеджер контекста, передаваемый инструкции `with`, также может возвращать объект, который присваивается локальной переменной в части `as` составной инструкции. Благодаря этому код, выполняющийся в блоке `with`, получает возможность непосредственно взаимодействовать с его контекстом.

Предположим, например, что вы хотите осуществлять запись в файл и быть уверенным в том, что он всегда корректно закрывается. Это можно сделать, передавая функцию `open()` инструкции `with`. Функция `open()` возвращает дескриптор для целевой переменной предложения `as` блока `with` и закрывает его при выходе из блока `with`.

```
with open('my_output.txt', 'w') as handle:
    handle.write('Некоторые данные!')
```

Описанный подход более предпочтителен, чем открытие и закрытие файла каждый раз вручную. Он гарантирует, что в конечном счете, когда поток управления покинет пределы блока `with`, файл действительно будет закрыт. Он также стимулирует вас к минимизации объема кода, выполняющегося в то время, пока файл остается открытым, что уже само по себе соответствует наилучшей практике программирования.

Все, что надо сделать для того, чтобы ваши собственные функции могли предоставлять значения для целевых выражений предложения `as`, — это вернуть значение из контекстного менеджера с помощью ключевого слова `yield`. Ниже мы определим менеджер контекста для извлечения экземпляра `Logger`, установим для него уровень ошибок и возвратим с помощью `yield` для дальнейшего использования в качестве целевого объекта предложения `as`.

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
```

```
    try:
        yield logger
    finally:
        logger.setLevel(old_level)
```

В результате вызова протоколирующих методов наподобие `debug()` для целевого объекта предложения `as` будут выводиться отладочные сообщения, поскольку в блоке `with` установлен достаточно низкий уровень протоколирования ошибок. В случае непосредственного использования модуля `logging` не будут выводиться никакие сообщения, поскольку установленным по умолчанию уровнем является `WARNING`.

```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug('Мое сообщение!')
    logging.debug('Эта строка не будет выведена')

>>>
Это мое сообщение!
```

После выхода из инструкции `with` вызов отладочных протоколирующих методов для экземпляра класса `Logger` с именем `'my-log'` не будет сопровождаться выводом сообщений, поскольку к этому времени восстанавливается заданный по умолчанию уровень протоколирования. Сообщения об ошибках выводятся всегда.

```
logger = logging.getLogger('my-log')
logger.debug('Отладочная информация не будет выводиться')
logger.error('Сообщения об ошибках будут выводиться')

>>>
Сообщения об ошибках будут выводиться
```

Что следует запомнить

- Инструкция `with` обеспечивает возможность повторного использования логики блоков `try/finally` и уменьшает визуальный шум.
- Встроенный модуль `contextlib` предоставляет декоратор `contextmanager`, упрощающий использование собственных функций в инструкциях `with`.
- Значение, возвращаемое с помощью ключевого слова `yield`, контекстными менеджерами передается предложению `as` инструкции `with`. Это удобный способ предоставления коду непосредственного доступа к источнику специального контекста.

Рекомендация 44. Повышайте надежность встроенного модуля pickle с помощью модуля copyreg

Встроенный модуль pickle может сериализовать объекты Python в байтовые потоки и выполнять обратную операцию десериализации байтов в объекты. Получаемые таким способом байтовые потоки не следует использовать для обмена данными между сторонами, для которых не установлены доверительные отношения. Модуль pickle обеспечивает возможность передачи объектов Python между программами, которые вы контролируете посредством двоичных каналов.

Примечание

Формат сериализации, используемый модулем pickle, изначально небезопасен. Сериализованные данные по существу представляют собой программу, описывающую способ реконструирования исходного объекта Python. Это означает, что вредоносная полезная нагрузка может быть использована с целью создания угроз для любой части программы на Python, которая пытается десериализовать данные.

В противоположность этому модуль json спроектирован как безопасный. Сериализованные JSON-данные содержат простое описание иерархии объектов. Десериализованные данные не подвергают программы на Python никаким дополнительным рискам. Для обмена данными между программами или людьми, не пользующимися взаимными доверительными отношениями, должны использоваться такие форматы, как JSON.

Предположим, например, что вы хотите использовать объект Python для того, чтобы представлять прогресс игрока. Состояние игры включает уровень, на котором находится игрок, и количество оставшихся жизней.

```
class GameState(object):
    def __init__(self):
        self.level = 0
        self.lives = 4
```

В процессе выполнения игры программа изменяет этот объект.

```
state = GameState()
state.level += 1 # Игрок переходит на следующий уровень
state.lives -= 1 # Игрок должен повторить попытку
```

Когда игрок заканчивает игру, программа может сохранить ее состояние в файле, чтобы впоследствии ее можно было продолжить с того же состояния. Модуль pickle упрощает решение этой задачи. Ниже мы сохраним объект GameState непосредственно в файл с помощью метода dump().

```
state_path = 'game_state.bin'
with open(state_path, 'wb') as f:
    pickle.dump(state, f)
```

Позже мы сможем загрузить файл и получить объект `GameState` в его прежнем состоянии, словно он и не сериализовался.

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
print(state_after.__dict__)
```

```
>>>
{'lives': 3, 'level': 1}
```

Все было бы хорошо, но есть одна проблема. Как быть в том случае, когда возможности игры со временем расширятся. Предположим, вы решили, что игрок может зарабатывать очки, стремясь добиться максимально высокого счета в игре. Для подсчета очков, зарабатываемых игроком, в класс `GameState` необходимо ввести новое поле.

```
class GameState(object):
    def __init__(self):
        # ...
        self.points = 0
```

Сериализация новой версии класса `GameState` с помощью модуля `pickle` будет работать точно так же, как и до этого. Ниже мы имитируем круговой процесс сохранения и восстановления состояния игры через файл путем сериализации строки с помощью метода `dump()` и обратного преобразования строки в объект с помощью метода `load()`.

```
state = GameState()
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'lives': 4, 'level': 0, 'points': 0}
```

Но что произойдет с сохраненными гораздо ранее старыми объектами `GameState`, которые пользователь также может захотеть восстановить? Ниже мы восстановим файл старой игры, используя программу с новым определением класса `GameState`.

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
print(state_after.__dict__)
```

```
>>>
{'lives': 3, 'level': 1}
```

Атрибут points отсутствует! Ситуация становится еще более запутанной, если учесть, что возвращенный объект является экземпляром нового класса GameState.

```
assert isinstance(state_after, GameState)
```

Такое поведение является побочным эффектом, обусловленным особенностями работы модуля pickle. Основным его назначением является упрощение сериализации объектов. Как только вы попытаетесь выйти за пределы тривиальной области применимости модуля pickle, функциональность модуля начинает рушиться самым неожиданным образом.

От этих проблем можно без труда избавиться, используя встроенный модуль copyreg. Этот модуль позволяет регистрировать функции, ответственные за сериализацию объектов Python, тем самым предоставляя вам возможность управлять поведением модуля pickle и делать его более надежным.

Значения атрибутов по умолчанию

В простейших случаях, для того чтобы десериализованные объекты GameState всегда имели все атрибуты, можно использовать аргументы по умолчанию. С этой целью переопределим конструктор.

```
class GameState(object):
    def __init__(self, level=0, lives=4, points=0):
        self.level = level
        self.lives = lives
        self.points = points
```

Чтобы использовать этот конструктор совместно с модулем pickle, определим вспомогательную функцию, которая принимает объект GameState и превращает его в кортеж параметров для модуля copyreg. Возвращенный кортеж содержит функцию, которую следует использовать для десериализации объектов, и параметры, передаваемые функции, выполняющей десериализацию.

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    return unpickle_game_state, (kwargs,)
```

Теперь нужно определить вспомогательную функцию unpickle_game_state(), которая принимает сериализованные данные и параметры от функции pickle_game_state и возвращает соответствующий объект GameState. Она представляет собой миниатюрную оболочку для конструктора.

```
def unpickle_game_state(kwargs):
    return GameState(**kwargs)
```

Далее регистрируем эти объект и функцию с помощью встроенного модуля copyreg.

```
copyreg.pickle(GameState, pickle_game_state)
```

Сериализация и десериализация работают так же, как и перед этим.

```
state = GameState()
state.points += 1000
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'lives': 4, 'level': 0, 'points': 1000}
```

Выполнив регистрацию, мы можем изменить определение класса GameState так, чтобы предоставить пользователю счетчик оставшихся в его распоряжении магических заклинаний. Это изменение аналогично тому, которое вносилось, когда мы добавляли в GameState поле для очков, заработанных игроком.

```
class GameState(object):
    def __init__(self, level=0, lives=4, points=0, magic=5):
```

Однако, в отличие от предыдущего, десериализация старого объекта GameState приведет к корректным данным об игре, и отсутствующих атрибутов уже не будет. Это работает, поскольку метод unpickle_game_state() вызывает непосредственно конструктор GameState. В случае отсутствия некоторых параметров в конструкторе используются значения по умолчанию, предусмотренные для именованных аргументов. Благодаря этому старые состояния игры при их десериализации получают для поля magic значения по умолчанию.

```
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'level': 0, 'points': 1000, 'magic': 5, 'lives': 4}
```

Управление версиями

Иногда в объекты Python приходится вносить изменения, удаляя некоторые поля в интересах обратной совместимости. В этом случае подход, основанный на использовании аргументов, заданных по умолчанию, перестает работать.

Предположим, вы пришли к выводу, что концепция игры, предполагающая ограниченное количество жизней, неудачна, и решили вообще исключить ее из игры. Ниже мы переопределим класс `GameState`, в котором поле, отображающее количество жизней, отсутствует.

```
class GameState(object):
    def __init__(self, level=0, points=0, magic=5):
        # ...
```

Проблема в том, что это разрушает десериализацию данных старых игр. Функция `unpickle_game_state()` передаст в конструктор `GameState` все поля старых игр, даже те, которые удалены.

```
pickle.loads(serialized)

>>>
TypeError: __init__() got an unexpected keyword argument 'lives'
```

Решение состоит в том, чтобы добавить в функцию, передаваемую в `copyreg`, параметр, определяющий версию. При обработке объекта `GameState` для сериализуемых новых данных будет указана версия 2.

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs['version'] = 2
    return unpickle_game_state, (kwargs,)
```

Старые версии данных не будут сопровождаться номером версии, что позволяет соответствующим образом манипулировать аргументами, передаваемыми конструктором `GameState`.

```
def unpickle_game_state(kwargs):
    version = kwargs.pop('version', 1)
    if version == 1:
        kwargs.pop('lives')
    return GameState(**kwargs)
```

Теперь десериализация старого объекта работает корректно.

```
copyreg.pickle(GameState, pickle_game_state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'magic': 5, 'level': 0, 'points': 1000}
```

Вы можете продолжать использовать такой подход для обработки изменений, появляющихся в будущих версиях того же класса. В функцию `unpickle_game_state()` можно помещать любую логику, необходимую для адаптации старой версии класса к новой.

Стабильные пути поиска при импорте

Используя модуль pickle, вы можете столкнуться с еще одной проблемой — нарушениями работы кода из-за переименования классов. На протяжении жизненного цикла программы часто приходится выполнять рефакторинг кода, изменяя имена классов и перемещая их в другие модули.

К сожалению, это нарушает нормальную работу модуля pickle, если не предпринять меры предосторожности.

Ниже мы изменим имя класса GameState на BetterGameState, полностью удалив класс из программы.

```
class BetterGameState(object):
    def __init__(self, level=0, points=0, magic=5):
        # ...
```

Теперь попытка десериализовать старый объект GameState будет неуспешной, поскольку этот класс не сможет быть найден.

```
pickle.loads(serialized)

>>>
AttributeError: Can't get attribute 'GameState' on <module
↳ '__main__' from 'my_code.py'>
```

Причиной появления этого исключения стало то, что в данных, обработанных модулем pickle, жестко закодирован путь к классу сериализованного объекта.

```
print(serialized[:25])

>>>
b'\x80\x03c__main__\nGameState\nq\x00)'
```

Решение заключается в том, чтобы вновь использовать модуль copyreg. Можете указать для функции стабильный идентификатор, который следует использовать при десериализации объекта. Это позволяет перемещать сериализованные данные в другие классы с другими именами при их десериализации и дает еще один уровень косвенности.

```
copyreg.pickle(BetterGameState, pickle_game_state)
```

Если вы используете модуль copyreg, то увидите, что в сериализованных данных закодирован не путь к классу BetterGameState, а путь к функции unpickle_game_state().

```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized[:35])
```

```
>>>
b'\x80\x03c__main__\nunpickle_game_state\nq\x00}'
```

Единственным недостатком этой техники является то, что она не позволяет изменить путь к модулю, в котором находится функция `unpickle_game_state()`. Если уж вы сериализовали данные с функцией, то она должна оставаться доступной по этому пути поиска для десериализации в будущем.

Что следует запомнить

- Встроенный модуль `pickle` может быть полезным лишь для операций сериализации и десериализации объектов между доверенными программами.
- Модуль `pickle` может не работать, если пытаться использовать его в более сложных случаях, чем самые простейшие.
- Используйте встроенный модуль `copyreg` с модулем `pickle` для добавления недостающих значений атрибутов, разрешайте использование различных версий классов и предоставляйте стабильные пути импорта.

Рекомендация 45. Используйте модуль `datetime` вместо модуля `time` для локальных часов

Всемирное скоординированное время (UTC) — это стандартное, не зависящее от временного пояса представление времени. Стандарт UTC отлично подходит для компьютеров, в которых время исчисляется в секундах, истекших с момента наступления полуночи 1 января 1970 года — начала эры UNIX. Люди по-разному отсчитывают время, в зависимости от того, в каком часовом поясе находятся. Мы говорим "полдень" или "8 часов утра", а не "15:00 по UTC минус 7 часов". Если ваша программа обрабатывает время, то вам, вероятно, придется преобразовывать время из системы UTC в местное время, более привычное людям.

Python предоставляет два способа преобразования времени с учетом часовых поясов. Старый способ, предполагающий использование встроенного модуля `time`, катастрофически подвержен ошибкам. Новый же способ, в котором используется встроенный модуль `datetime`, замечательно работает в сочетании с предложенным сообществом пользователей пакетом `pytz`.

Чтобы понять, почему модуль `datetime` является наилучшим выбором, а использования модуля `time` следует избегать, нужно ознакомиться с каждым из них.

Модуль time

Функция localtime() из встроенного модуля time позволяет преобразовывать временные метки UNIX (количество секунд в UTC, истекших с момента наступления эры UNIX) в локальное время, соответствующее часовому поясу хоста (в моем случае — тихоокеанскому летнему времени, PDT).

```
from time import localtime, strftime

now = 1407694710
local_tuple = localtime(now)
time_format = '%Y-%m-%d %H:%M:%S'
time_str = strftime(time_format, local_tuple)
print(time_str)

>>>
2014-08-10 11:18:30
```

Как выполнить преобразование из одного часового пояса в другой? Предположим, например, что вы собираетесь совершить перелет из Сан-Франциско в Нью-Йорк и хотите знать, который час будет в Сан-Франциско, когда вы прибудете в Нью-Йорк.

Непосредственное манипулирование значениями, которые возвращаются функциями time(), localtime() и strptime(), — плохая идея. Часовые пояса подвержены изменениям в соответствии с местным законодательством. Учесть все это довольно сложно, особенно если вы хотите обрабатывать расписания отправления и прибытия авиарейсов в города, разбросанные по всему земному шару.

Многие операционные системы поддерживают конфигурационные файлы, которые автоматически учитывают изменение часовых поясов. В Python работу с часовыми поясами обеспечивает модуль time. Например, ниже выполняется синтаксический разбор времени отлета в привязке к локальному времени в Сан-Франциско, соответствующему Тихоокеанскому летнему времени.

```
parse_format = '%Y-%m-%d %H:%M:%S %Z'
depart_sfo = '2014-05-01 15:45:16 PDT'
time_tuple = strptime(depart_sfo, parse_format)
time_str = strftime(time_format, time_tuple)
print(time_str)

>>>
2014-05-01 15:45:16
```

Увидев, как PDT работает с функцией `strptime()`, вы могли бы предположить, что другие часовые пояса, известные моему компьютеру, также будут работать. К сожалению, это не так. Вместо этого `strptime()` генерирует исключение для восточного поясного времени (часовой пояс Нью-Йорка).

```
arrival_nyc = '2014-05-01 23:33:24 EDT'
time_tuple = strptime(arrival_nyc, time_format)

>>>
ValueError: unconverted data remains: EDT
```

В данном случае проблема обусловлена зависимостью модуля `time` от платформы. Его фактическое поведение определяется характером взаимодействия базовых C-функций с операционной системой хоста. Это делает ненадежной функциональность модуля `time` в Python, в результате чего он не работает одинаково корректно со всеми часовыми поясами. Следовательно, вам следует избегать использования модуля `time` для этих целей. Если вы все же должны использовать его, то ограничьтесь его применением лишь для преобразований между UTC и локальным временем, установленным для хоста. Для всех остальных типов преобразований используйте модуль `datetime`.

Модуль `datetime`

В Python имеется возможность представлять время также с помощью класса `datetime` из встроенного модуля `datetime`. Модуль `datetime`, как и модуль `time`, можно использовать для перехода от текущего времени, выраженного в системе UTC, к местному времени.

Ниже время в системе UTC преобразовано в заданное на компьютере автора тихоокеанское поясное время.

```
from datetime import datetime, timezone

now = datetime(2014, 8, 10, 18, 18, 30)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)

>>>
2014-08-10 11:18:30-07:00
```

С помощью модуля `datetime` легко выполняется и обратное преобразование местного времени во временные метки UNIX в системе UTC.

```
time_str = '2014-08-10 11:18:30'
now = datetime.strptime(time_str, time_format)
```

```
time_tuple = now.timetuple()
utc_now = mktime(time_tuple)
print(utc_now)

>>>
1407694710.0
```

В отличие от модуля time, в модуле datetime имеются средства, обеспечивающие надежные переходы между часовыми поясами с помощью класса tzinfo и соответствующих методов. Чего не хватает, так это определений часовых поясов, отличных от UTC.

К счастью, сообщество пользователей Python устранило этот пробел с помощью модуля pytz, доступного для загрузки из каталога пакетов Python (https://pypi.python.org/pypi/pytz/). Модуль pytz содержит полную базу данных обо всех определениях часовых поясов, которые вам когда-либо могут понадобиться.

Для эффективного использования модуля pytz всегда необходимо сначала преобразовать местное время в UTC. Любые операции с использованием модуля datetime необходимо выполнять, используя значения UTC (например, смещения). Окончательным шагом является преобразование в местное время.

Например, ниже мы преобразуем время прибытия самолета в Нью-Йорк по местному времени во Всемирное скоординированное время. Несмотря на то что некоторые из вызовов функций кажутся избыточными, все они необходимы в случае использования модуля pytz.

```
arrival_nyc = '2014-05-01 23:33:24'
nyc_dt_naive = datetime.strptime(arrival_nyc, time_format)
eastern = pytz.timezone('US/Eastern')
nyc_dt = eastern.localize(nyc_dt_naive)
utc_dt = pytz.utc.normalize(nyc_dt.astimezone(pytz.utc))
print(utc_dt)

>>>
2014-05-02 03:33:24+00:00
```

Получив время в стандарте UTC, можем преобразовать его в местное время Сан-Франциско.

```
pacific = pytz.timezone('US/Pacific')
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))
print(sf_dt)

>>>
2014-05-01 20:33:24-07:00
```

Столь же легко можно выполнить преобразование в местное время Непала.

```
nepal = pytz.timezone('Asia/Katmandu')
nepal_dt = nepal.normalize(utc_dt.astimezone(nepal))
print(nepal_dt)

>>>
2014-05-02 09:18:24+05:45
```

С модулями `datetime` и `pytz` эти преобразования выполняются согласованным образом во всех средах, независимо от того, какая операционная система установлена на компьютере.

Что следует запомнить

- Избегайте использования модуля `time` для преобразования времени между различными часовыми поясами.
- Используйте встроенный модуль `datetime` вместе с модулем `pytz` для надежного преобразования времени в различных часовых поясах.
- Всегда представляйте время, используя UTC, и выполняйте преобразование в местное время в качестве завершающего шага перед представлением.

Рекомендация 46. Используйте встроенные алгоритмы и структуры данных

Реализуя на языке Python программы, обрабатывающие нетривиальные объемы данных, вы непременно столкнетесь с падением производительности, обусловленным алгоритмической сложностью кода. Обычно этот результат не следует связывать с быстродействием языка Python как такового (см. раздел "Рекомендация 41. Старайтесь использовать модуль `concurrent.futures` для обеспечения истинного параллелизма", если это не так). Скорее всего, причина заключается в том, что используемые вами алгоритмы и структуры данных не являются наилучшими для вашей задачи. К счастью, в стандартной библиотеке Python имеется множество алгоритмов и структур данных, которые вы сможете в готовом виде встраивать в свои программы. Корректная реализация некоторых наиболее ценных инструментов, которые могли бы вам пригодиться, оказывается сложной. Поэтому реализация общедоступной функциональности поможет вам сэкономить время и избавит от головной боли.

Двухсторонняя очередь

Класс `dequeue` из модуля `collections` — это двухсторонняя (двусвязная) очередь. Он обеспечивает возможность добавления и удаления элементов как в начало, так и в конец очереди, причем эти операции выполняются по алгоритму с классом временной сложности *постоянного времени*. Благодаря этому данный класс является идеальным для очередей с дисциплиной обслуживания FIFO (сокращ. от "First In, First Out" — "первым пришел — первым обслужен").

```
fifo = deque()
fifo.append(1) # Производитель
x = fifo.popleft() # Потребитель
```

Встроенный тип `list` также содержит упорядоченную последовательность элементов наподобие очереди. Он тоже позволяет вставлять и удалять элементы в конце очереди по алгоритму с тем же классом сложности (постоянное время). Однако операции вставки и удаления элементов в начале списка работают по алгоритму *линейного времени*, работающему значительно медленнее алгоритма постоянного времени.

Упорядоченный словарь

Поскольку стандартные словари не упорядочены, порядок итерирования по элементам словаря с одними и теми же ключами и значениями может быть различным. Такое поведение является побочным эффектом того способа, который используется для реализации хеш-таблицы словаря, обеспечивающей быстрый доступ к элементам.

```
a = {}
a['foo'] = 1
a['bar'] = 2

# Заполнить 'b' случайными данными, чтобы вызвать
# коллизию хеш-функции
while True:
    z = randint(99, 1013)
    b = {}
    for i in range(z):
        b[i] = i
    b['foo'] = 1
    b['bar'] = 2
    for i in range(z):
        del b[i]
    if str(b) != str(a):
        break
```

```
print(a)
print(b)
print('Равны?', a == b)

>>>
{'foo': 1, 'bar': 2}
{'bar': 2, 'foo': 1}
Равны? True
```

Класс `OrderedDict` из модуля `collections` — это специальный тип словаря, отслеживающего порядок, в котором вставляются ключи. Итерирование по ключам класса `OrderedDict` обладает предсказуемым поведением. Такое детерминированное поведение кода может значительно упростить его тестирование и отладку.

```
a = OrderedDict()
a['foo'] = 1
a['bar'] = 2
b = OrderedDict()
b['foo'] = 'red'
b['bar'] = 'blue'

for value1, value2 in zip(a.values(), b.values()):
    print(value1, value2)

>>>
1 red
2 blue
```

Словарь по умолчанию

Словари удобны для учета данных и отслеживания статистики. Одной из трудностей работы со словарями является то, что вы не можете заранее предполагать наличие какого-либо ключа. Из-за этого даже такие простые операции, как инкрементирование счетчика, хранящегося в словаре, становятся громоздкими.

```
stats = {}
key = 'my_counter'
if key not in stats:
    stats[key] = 0
stats[key] += 1
```

Класс `defaultdict` из модуля `collections` упрощает операции такого рода, автоматически сохраняя значение по умолчанию в случае отсутствия ключа. Все, что от вас требуется, — это предоставить функцию, которая будет возвращать значение по умолчанию всякий раз, когда

ключ отсутствует. В приведенном ниже примере для этого используется встроенная функция `int()`, которая возвращает `0` (дополнительный пример см. в разделе "Рекомендация 23. Принимайте функции вместо классов в случае простых интерфейсов"). Теперь инкрементирование счетчика значительно упрощается.

```
stats = defaultdict(int)
stats['my_counter'] += 1
```

Очереди типа кучи

Куча — удобный тип структуры данных для поддержания приоритетных очередей. Модуль `heapq` предоставляет такие функции для создания куч в стандартных списках, как `heappush`, `heappop` и `nsmallest`.

В кучи можно вставлять элементы с любым приоритетом в любом порядке.

```
a = []
heappush(a, 5)
heappush(a, 3)
heappush(a, 7)
heappush(a, 4)
```

Первыми всегда удаляются элементы с наивысшим приоритетом (наименьшим числовым значением приоритета).

```
print(heappop(a), heappop(a), heappop(a), heappop(a))
>>>
3 4 5 7
```

Результирующий список можно легко использовать вне модуля `heapq`. Доступ к куче по индексу `0` всегда возвращает наименьший элемент.

```
a = []
heappush(a, 5)
heappush(a, 3)
heappush(a, 7)
heappush(a, 4)
assert a[0] == nsmallest(1, a)[0] == 3
```

Вызов метода `sort()` для списка сохраняет его инвариантность.

```
print('До:', a)
a.sort()
print('После: ', a)

>>>
До: [3, 4, 7, 5]
После: [3, 4, 5, 7]
```

Длительность выполнения каждой из этих операций heapq логарифмически зависит от длины списка. В случае выполнения той же работы с помощью стандартного списка Python эта зависимость имеет линейный характер.

Двоичный поиск

Длительность поиска элемента линейно растет с увеличением длины списка, если вызывать метод index().

```
x = list(range(10**6))
i = x.index(991234)
```

Функции модуля bisect, такие как bisect_left(), обеспечивают эффективный алгоритм двоичного поиска в последовательности отсортированных элементов. Возвращаемый индекс представляет точку вставки значения в последовательность.

```
i = bisect_left(x, 991234)
```

Бинарный поиск имеет логарифмический класс сложности. Это означает, что двоичный поиск в списке, содержащем миллион элементов, займет приблизительно столько же времени, сколько и линейный поиск с помощью метода index(), выполняемый в списке, содержащем 14 элементов. Комментарии здесь излишни!

Итераторы

Встроенный модуль itertools содержит большое количество функций, предназначенных для работы с итераторами (более подробно см. в разделах "Рекомендация 16. Не упускайте возможность использовать генераторы вместо возврата списков" и "Рекомендация 17. Не забывайте о мерах предосторожности при итерировании аргументов"). Не все они доступны в Python 2, но их можно легко встроить, действуя согласно простым способам, документированным в модуле. Справку об этом вы сможете получить, выполнив в командной строке интерактивного сеанса работы с Python команду help(itertools).

Функции itertools можно разделить на три основные категории.

- Связывание итераторов.
  - chain(). Объединяет несколько итераторов в один последовательный итератор.
  - cycle(). Бесконечно повторяет элементы итератора.
  - tee(). Разбивает итератор на несколько параллельных итераторов.

  - `zip_longest()`. Разновидность встроенной функции `zip()`, которая работает с итераторами различной длины.
- ✦ Фильтрация элементов из итератора.
  - `islice()`. Создает срезы итератора по числовым индексам без копирования.
  - `takewhile()`. Возвращает элементы из итератора, пока функция-предикат возвращает значение `True`.
  - `dropwhile()`. Возвращает элементы из итератора, как только функция-предикат впервые возвращает значение `False`.
  - `filterfalse()`. Возвращает все элементы из итератора, когда функция-предикат возвращает `False`. Противоположность встроенной функции `filter()`.
- ✦ Сочетания элементов из итераторов.
  - `product()`. Возвращает декартово произведение элементов из итератора, что представляет отличную альтернативу глубоко вложенным генераторам списков.
  - `permutations()`. Возвращает упорядоченные перестановки длиной N с элементами из итератора.
  - `combination()`. Возвращает неупорядоченные сочетания длиной N с неповторяющимися элементами из итератора.

Здесь не были упомянуты многие другие функции, которые предлагаются в модуле `itertools`. Поэтому всякий раз, когда вам приходится иметь дело со сложным итерационным кодом, обращайтесь к документации `itertools`, и вы наверняка найдете там описание функций, которые помогут вам справиться со стоящей перед вами задачей.

Что следует запомнить

- ✦ Используйте встроенные модули Python для работы с алгоритмами и структурами данных.
- ✦ Не пытайтесь самостоятельно изобретать велосипед. Вряд ли вам удастся сделать это лучше, чем это было сделано до вас.

Рекомендация 47. Используйте класс `Decimal`, когда на первый план выходит точность

Язык Python великолепно подходит для написания кода, который взаимодействует с числовыми данными. Целочисленный тип в Python может представлять значения практически любого размера. Его тип чисел двойной точности с плавающей точкой соответствует требовани-

ям стандарта IEEE 754. Кроме того, язык предоставляет стандартные комплексные числа для работы с мнимыми значениями. Однако всего этого еще недостаточно для всех ситуаций без исключения.

Предположим, например, что вы хотите вычислить, какая сумма должна быть взыскана с пользователя за международный телефонный звонок. Вам известна длительность разговора (скажем, 3 минуты 42 секунды). Вам также известна стоимость одной минуты звонка из США в Антарктику (1,45 доллара за минуту). Какова должна быть сумма оплаты?

Использование чисел с плавающей точкой дает вполне разумный результат.

```
rate = 1.45
seconds = 3*60 + 42
cost = rate * seconds / 60
print(cost)

>>>
5.364999999999999
```

Но при округлении до ближайшего значения целых центов происходит округление до меньшего значения, а не большего, как вам хотелось бы, чтобы покрыть все расходы, связанные со звонком клиента.

```
print(round(cost, 2))

>>>
5.36
```

Предположим далее, что вы хотите поддерживать также очень короткие звонки, осуществляемые между пунктами, плата за соединение между которыми значительно ниже. Например, вычислим размер оплаты за телефонный звонок длительностью 5 секунд по тарифу 0,05 доллара за минуту.

```
rate = 0.05
seconds = 5
cost = rate * seconds / 60
print(cost)

>>>
0.004166666666666667
```

Результирующее значение с плавающей точкой настолько мало, что оно округляется до нуля. А ведь так не должно быть!

```
print(round(cost, 2))
```

```
>>>
0.0
```

Проблема разрешается с помощью класса `Decimal` из встроенного модуля `decimal`. Класс `Decimal` позволяет выполнять операции над числами с фиксированной десятичной точкой, для которых по умолчанию допускается до 28 десятичных значащих цифр. При необходимости данное количество может быть увеличено. Это позволяет обойти проблемы с точностью, которые присущи числам с плавающей точкой, удовлетворяющим стандарту IEEE 754. Кроме того, данный класс обеспечивает *бо*льшую степень контроля над выполнением операций округления.

Например, повторное вычисление с помощью класса `Decimal` размера оплаты за телефонный звонок в Антарктику приводит к точному, а не приближенному результату.

```
rate = Decimal('1.45')
seconds = Decimal('222') # 3*60 + 42
cost = rate * seconds / Decimal('60')
print(cost)

>>>
5.365
```

В классе `Decimal` имеется встроенная функция, позволяющая выполнять округление для нужного количества десятичных знаков после запятой с желаемым поведением операции округления.

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(rounded)

>>>
5.37
```

Использование метода `quantize()` для этих целей также позволяет корректно обрабатывать небольшие суммы к оплате для коротких дешевых звонков. Как показано ниже, вычисления с помощью класса `Decimal` по-прежнему дают сумму менее 1 цента за один звонок.

```
rate = Decimal('0.05')
seconds = Decimal('5')
cost = rate * seconds / Decimal('60')
print(cost)

>>>
0.004166666666666666666666666667
```

Однако поведение метода `quantize()` гарантирует, что это значение будет округлено до целых центов в “правильную” сторону.

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(rounded)

>>>
0.01
```

В то время как класс `Decimal` отлично работает с числами с фиксированной точкой, у него все еще имеются ограничения, связанные с точностью (например, результат вычисления 1/3 все равно будет приближенным). Для представления рациональных чисел в отсутствие каких-либо ограничений в отношении точности используйте класс `Fraction` из встроенного модуля `fractions`.

Что следует запомнить

- В модулях Python имеются встроенные типы и классы, которые могут представлять практически любой тип числовых значений.
- Класс `Decimal` идеально подходит для таких ситуаций, как вычисление денежных значений, которые требуют высокой точности и вполне определенного порядка округления результатов.

Рекомендация 48. Знайте, где искать модули, разработанные сообществом Python

В Python имеется центральный репозиторий модулей (`https://pypi.python.org`), которые вы можете устанавливать и использовать в своих программах. Эти модули создаются и сопровождаются такими же людьми, как и вы, — сообществом пользователей Python. Если вы сталкиваетесь с незнакомой вам задачей, то каталог пакетов Python (Python Package Index — PyPI) — замечательное место для поиска кода, который приблизит вас к достижению цели.

Для доступа к PyPI следует воспользоваться утилитой командной строки под названием `pip`, которая устанавливается по умолчанию в версии 3.4 и более поздних версиях (она также доступна по команде `python -m pip`). Инструкции для установки `pip` для более ранних версий можно найти по адресу `https://packaging.python.org`.

После установки `pip` вы сможете очень легко установить любой другой новый модуль PyPI. Например, ниже мы установим модуль `pytz`, который уже использовался в одном из предыдущих разделов (см. раздел "Рекомендация 45. Используйте модуль `datetime` вместо модуля `time` для локальных часов").

```
$ pip3 install pytz
Downloading/unpacking pytz
```

```
    Downloading pytz-2014.4.tar.bz2 (159kB): 159kB downloaded
    Running setup.py (...) egg_info for package pytz

Installing collected packages: pytz
    Running setup.py install for pytz

Successfully installed pytz
Cleaning up...
```

В приведенном примере мы использовали команду `pip3` для установки пакета, соответствующего версии Python 3. Также доступна команда `pip` (без цифры "3"), позволяющая установить пакеты для Python 2. В настоящее время большинство пакетов доступно для любой версии Python (см. раздел "Рекомендация 1. Следите за тем, какую версию Python вы используете"). Средство `pip` также может использоваться совместно со средством `pyvenv` для отслеживания наборов пакетов, подлежащих установке для ваших проектов (раздел "Рекомендация 53. Используйте виртуальные среды для изолированных и воспроизводимых зависимостей").

Каждый модуль в каталоге PyPI имеет собственную программную лицензию. Большинство пакетов, особенно популярных, имеют бесплатные лицензии или лицензии на открытое ПО (более подробно см. `http://opensource.org`). В большинстве случаев эти лицензии позволяют вам включать копию модуля в свою программу (если сомневаетесь, обратитесь к юристам).

Что следует запомнить

- Каталог пакетов Python (Python Package Index — PyPI) содержит большое количество общих пакетов, которые создаются и сопровождаются сообществом пользователей Python.
- Средство командной строки `pip` используется для установки пакетов из PyPI.
- Средство `pip` устанавливается по умолчанию в Python 3.4 и более поздних версиях; для более ранних версий его необходимо устанавливать самостоятельно.
- Большинство модулей PyPI бесплатны и относятся к категории ПО с открытым исходным кодом.

7 Совместная работа

В Python предусмотрены средства, облегчающие создание четко определенных интерфейсов прикладного программирования (API) с отчетливыми границами между ними. Сообщество Python выработало принципы наилучшей практики, которые максимально облегчают обслуживание кода в процессе его эксплуатации. Также существуют стандартные инструменты, поставляемые с Python, которые обеспечивают совместную работу команд программистов, работающих в разных средах.

Совместная работа над программами Python требует тщательного подхода к стилю написания кода. Даже если вы работаете независимо, существует большая вероятность того, что вы будете использовать код, содержащийся в стандартных библиотеках или пакетах с открытым исходным кодом, написанных другими людьми. Очень важно, чтобы вам были понятны механизмы, упрощающие сотрудничество с другими программистами на языке Python.

Рекомендация 49. Снабжайте строками документирования каждую функцию, класс и модуль

Документация играет чрезвычайно важную роль в Python ввиду динамичной природы этого языка. Python предоставляет встроенную поддержку присоединения документации к блокам кода. В отличие от многих других языков, документация из исходного кода программы доступна непосредственно во время выполнения программы.

Например, вы можете добавить так называемые *строки документирования* (docstrings) сразу же за инструкцией `def` функции.

```
def palindrome(word):
    """Возвращает True, если заданное слово является
       палиндромом."""
    return word == word[::-1]
```

Строки документирования можно извлекать из самой программы на Python с помощью специального атрибута функции, который называется `__doc__`.

```
print(repr(palindrome.__doc__))
```

```
>>>
Возвращает True, если заданное слово является палиндромом.'
```

Строки документирования могут присоединяться к функциям, классам и модулям, становясь частью процесса компиляции и выполнения программы. Поддержка этого средства и атрибута `__doc__` имеет три следствия.

- Легкодоступная документация упрощает интерактивную разработку. Используя встроенную функцию `help()`, можно проверять наличие строк документирования в функциях, классах и модулях. Благодаря этому разработка алгоритмов, тестирование прикладных программных интерфейсов и написание небольших фрагментов кода с помощью интерактивного интерпретатора Python ("оболочки" Python) и таких средств, как IPython Notebook (`http://ipython.org`), доставляет одно удовольствие.
- Наличие стандартного способа подготовки документации упрощает создание инструментов, преобразующих текст в более приемлемые форматы (например, HTML). Благодаря этому сообщество пользователей Python получило в свое распоряжение такие великолепные средства генерирования документации, как Sphinx (`http://sphinxdoc.org`). Кроме того, это подтолкнуло к созданию основанных сообществом сайтов, таких как Read the Docs (`https://readthedocs.org`), которые бесплатно предоставляют возможность размещения хорошо подготовленной документации для проектов Python с открытым исходным кодом.
- Первоклассная, легкодоступная и отлично выглядящая документация Python поощряет людей к написанию еще большего количества документации такого же качества. Члены сообщества Python убеждены в важности документирования кода. Они считают, что понятие "хороший код" означает, кроме всего прочего, хорошо документированный код. Это означает, что в большинстве случаев вы можете рассчитывать на то, что библиотеки Python снабжены отличной документацией.

Чтобы приобщиться к этой великолепной культуре документирования кода, вы должны следовать нескольким рекомендациям, касающимся написания строк Dostrings. Более подробное обсуждение этого

вопроса вы найдете на сайте PEP 257 (`http://www.python.org/dev/peps/pep-0257/`). Также существуют стандарты оформления строк документирования, которые вы должны обязательно соблюдать.

Документирование модулей

Каждый модуль должен снабжаться строками документирования верхнего уровня. Этот строковый литерал должен быть первой инструкцией в файле исходного кода, а также начинаться и заканчиваться тремя двойными кавычками (`"""`). В него помещают сведения о модуле и его содержимом ознакомительного характера.

Первая строка документирования должна содержать единственное предложение, описывающее назначение модуля. В следующих абзацах должно содержаться подробное описание работы модуля, рассчитанное на пользователей. Строки документирования модуля служат отправным пунктом, в котором можно отдельно выделить важные классы и функции, принадлежащие модулю.

Ниже приведен пример строки документирования модуля.

```
# words.py
#!/usr/bin/env python3
"""Библиотека для тестирования слов с помощью различных
   лингвистических шаблонов.

Тестирование отношений между словами иногда может быть затруднено!
Данный модуль предоставляет методы, позволяющие легко определить,
какие из найденных слов имеют специальные свойства.

Доступные функции:
- palindrome: определяет, является ли слово палиндромом.
- check_anagram: определяет, являются ли данная пара слов
  анаграммой.
...
"""

# ...
```

Если модуль является утилитой командной строки, то строки документирования — это отличное место для размещения информации о порядке запуска данной утилиты.

Документирование классов

Каждый класс должен снабжаться строками документирования уровня класса. В основном эти строки следуют тому же шаблону, что и строки документирования уровня модуля. Первая строка должна состоять

из одного предложения и содержать описание назначения класса. В последующих абзацах приводятся наиболее важные детали того, как работает этот класс.

В строках документирования уровня класса должны быть отдельно указаны открытые атрибуты и методы. Они также должны содержать рекомендации относительно того, как подклассы должны взаимодействовать с защищенными атрибутами (см. раздел "Рекомендация 27. Предпочитайте открытые атрибуты закрытым") и методами суперкласса.

Ниже приведен пример строки документирования класса.

```
class Player(object):
    """Представляет игрока в игре.

    Подклассы могут переопределять метод 'tick', обеспечивающий
    пользовательскую анимацию движений игрока в зависимости от
    запаса силы и т.п.
Открытые атрибуты:
- power: неизрасходованная сила(значение с плавающей запятой
         (в интервале от 0 до 1).
- coins: количество найденных монет на данном уровне
         (целочисленное значение).
"""
# ...
```

Документирование функций

Каждая общедоступная функция и метод должны снабжаться строками документирования. Эти строки должны следовать тому же шаблону, что и модули и классы. Первая строка должна состоять из одного предложения и содержать описание функции. Последующие абзацы должны содержать описание специфического поведения и аргументов функции. Также должны быть указаны возвращаемые значения и описаны исключения, которые вызывающий код должен обрабатывать как часть интерфейса функции.

Ниже приведен пример строки документирования функции.

```
def find_anagrams(word, dictionary):
"""Находит все анаграммы для слова.

Эта функция выполняется лишь со скоростью выполнения теста,
проверяющего членство в контейнере 'dictionary'.
Она будет работать медленно, если 'dictionary' - список,
и быстро, если 'dictionary' - множество.
```

```
Аргументы:
word: строка целевого слова.
dictionary: контейнер, содержащий все строки, о которых
известно, что они представляют собой действительные слова.

Возвращаемые значения:
Список найденных анаграмм. Если ни одно слово не найдено,
этот список пустой.
"""
# ...
```

Кроме того, при написании строк документирования для функций встречаются особые случаи, о которых необходимо знать.

- Если ваша функция не имеет аргументов, а возвращаемое значение — простое, то вполне допустимо описать ее одним предложением.
- Если ваша функция ничего не возвращает, то лучше вообще не упоминать о возвращаемом значении, чем писать "возвращает `None`".
- Если вы не ожидаете, что ваша функция может сгенерировать исключение в процессе нормальной работы, то не пишите об этом.
- Если ваша функция принимает переменное количество аргументов (см. раздел "Рекомендация 18. Снижайте визуальный шум с помощью переменного количества позиционных аргументов") или именованные аргументы (см. "Рекомендация 19. Обеспечивайте опциональное поведение с помощью именованных аргументов"), то используйте выражения `*args` и `**kwargs` в документированном списке аргументов для описания их назначения.
- Если ваша функция имеет аргументы с заданными по умолчанию значениями, то эти значения по умолчанию должны быть указаны (см. раздел "Рекомендация 20. Используйте значение `None` и средство `Docstrings` при задании динамических значений по умолчанию для аргументов").
- Если ваша функция — генератор (см. раздел "Рекомендация 16. Не упускайте возможность использовать генераторы вместо возврата списков"), то строки документирования должны описывать, что именно возвращает генератор в процессе итерирования.
- Если ваша функция — сопрограмма (см. раздел "Рекомендация 40. Используйте сопрограммы для одновременного выполнения нескольких функций"), то строки документирования должны содержать описание того, что возвращает сопрограмма, что она ожида-

ет получить из выражения `yield` и когда останавливается итерационный процесс.

Примечание

После того как вы запишете строки документирования для своих модулей, важно следить за обновлением документации. Встроенный модуль `doctest` упрощает выполнение контрольных примеров, включаемых в строки документирования для того, чтобы можно было убедиться в том, что между исходным кодом и документацией со временем не возникают расхождения.

Что следует запомнить

- Пишите документацию для каждого модуля, класса и функции, используя строки документирования кода. Обновляйте документацию по мере внесения изменений в код.
- Для модулей: кратко описывайте содержимое модуля, выделяя важные классы функции, о которых должны знать пользователи.
- Для классов: документируйте поведение, важные атрибуты и поведение подклассов в строках документирования, помещая их вслед за инструкциями `class`.
- Для функций и методов: документируйте каждый аргумент, возвращаемое значение, генерируемое исключение и другие аспекты поведения в строках документирования, помещая их вслед за инструкцией `def`.

Рекомендация 50. Используйте пакеты для организации модулей и предоставления стабильных API

Вполне естественно, что по мере роста кодовой базы программ вы сталкиваетесь с необходимостью реорганизации ее структуры. С этой целью вы разбиваете крупные функции на более мелкие, преобразуете структуры данных во вспомогательные классы (см. раздел "Рекомендация 22. Отдавайте предпочтение структуризации данных с помощью классов, а не словарей или кортежей") и разделяете функциональность между различными модулями, которые зависят друг от друга.

В какой-то момент вы обнаружите, что число модулей настолько возросло, что для сохранения ясности структуры кода требуется вводить дополнительный программный слой. В Python эта задача решается с помощью *пакетов* — модулей, содержащих другие модули.

В большинстве случаев пакеты определяют, помещая в каталог пустой файл с именем *__init__.py*. После этого любой файл Python, находящийся в том же каталоге, что и файл *__init__.py*, становится доступным для импорта с использованием пути относительно данного каталога. Предположим, что ваша программа имеет показанную ниже структуру каталогов.

```
main.py
mypackage/__init__.py
mypackage/models.py
mypackage/utils.py
```

Чтобы импортировать модуль `utils`, вы используете абсолютное имя модуля, которое включает имя каталога пакета.

```
# main.py
from mypackage import utils
```

Эта же схема остается в силе и в тех случаях, когда каталоги пакетов располагаются в других пакетах (например, `mypackage.foo.bar`).

Примечание

В Python 3.4 введен более гибкий способ определения пакетов: пространства имен. Пространства имен могут включать модули из совершенно разных каталогов, zip-архивов и даже удаленных систем. За более подробными сведениями о пространствах имен обратитесь к документу PEP 420 (`http://www.python.org/dev/peps/pep-0420/`).

Предоставляемая пакетами функциональность преследует в программах на языке Python две основные цели, которые описаны ниже.

Пространства имен

Прежде всего, пакеты предназначены для того, чтобы облегчить группирование модулей по различным пространствам имен. Это позволяет иметь много модулей с одним и тем же именем, но различными путями доступа, которые являются уникальными. Например, ниже приведен фрагмент программы, которая импортирует атрибуты из двух модулей с одним и тем же именем `utils.py`. Это возможно, поскольку к модулям можно обращаться, используя их абсолютные пути.

```
# main.py
from analysis.utils import log_base2_bucket
from frontend.utils import stringify

bucket = stringify(log_base2_bucket(33))
```

Однако такой подход перестает работать, если функции, классы или подмодули определены в одноименных пакетах. Допустим, вы хотите использовать функцию `inspect()`, которая имеется как в модуле `analysis.utils`, так и в модуле `frontend.utils`. Непосредственное импортирование этих атрибутов не сработает, поскольку вторая инструкция `import` перекроет значение `inspect` в текущей области видимости.

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect # Перекрывает!
```

Решение заключается в использовании предложения `as` инструкции `import` для переименования объектов, импортируемых в текущую область видимости.

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect

value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Равенство!')
```

Предложение `as` можно использовать для переименования всего, что вы получаете с помощью инструкции `import`, включая целые модули. Это упрощает доступ к коду в пространствах имен и делает его идентификацию понятной в процессе его использования.

Примечание

Другой подход, позволяющий избежать конфликта между импортированными именами, заключается в том, чтобы при обращении к именам всегда использовать уникальное имя модуля с наивысшим положением в иерархии.

Применительно к приведенному выше примеру вы прежде всего должны импортировать модули `analysis.utils` и `frontend.utils` с помощью инструкций `import`. После этого вы должны обращаться к функциям `inspect()`, используя полные пути доступа к ним: `analysis.utils.inspect()` и `frontend.utils.inspect()`.

Такой подход избавляет от необходимости использовать предложение `as`. Он также делает для новых читателей совершенно очевидным то, где именно определена каждая функция.

Стабильные API

Второе назначение пакетов в Python — это предоставление строго определенных, стабильных интерфейсов прикладного программирования (API) внешним потребителям.

Когда вы пишете API для широкого круга потребителей, например в виде пакета с открытым исходным кодом (см. раздел "Рекомендация 48: Знайте, где искать модули, разработанные сообществом Python"), вы хотите предоставить стабильную функциональность, которая не изменяется от выпуска к выпуску. Для этого очень важно скрывать организацию внутреннего кода от внешних пользователей. Это позволяет выполнять рефакторинг и улучшать внутренние модули пакета, не создавая при этом препятствий для их эксплуатации существующими пользователями.

Python может ограничивать открытость API для потребителей с помощью специального атрибута `__all__` модуля или пакета. Значением `__all__` является список всех имен, экспортируемых из модуля как часть его общедоступного API. Когда в коде, использующем модуль, выполняется инструкция `from foo import *`, из модуля `foo` импортируются лишь атрибуты, фигурирующие в списке `foo.__all__`. В случае отсутствия свойства `__all__` у модуля `foo` импортируются лишь общедоступные атрибуты — те, имя которых не начинается с символа подчеркивания (см. раздел "Рекомендация 27. Предпочитайте общедоступные атрибуты закрытым").

Допустим, вы хотите предоставить пакет для расчета столкновений между движущимися частицами. Определим модуль `models` из пакета `mypackage`, который будет содержать представление частиц.

```
# models.py
__all__ = ['Projectile']

class Projectile(object):
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity
```

Мы также определим модуль `utils` в пакете `mypackage`, который будет выполнять различные операции над экземплярами `Projectile`, такие как имитация столкновений между ними.

```
# utils.py
from . models import Projectile

__all__ = ['simulate_collision']

def _dot_product(a, b):
    # ...

def simulate_collision(a, b):
    # ...
```

Теперь мне хотелось бы предоставить все общедоступные части этого API в виде набора атрибутов, доступных в модуле `mypackage`. Это позволит потребителям модуля всегда выполнять импорт непосредственно из пакета `mypackage` вместо того, чтобы импортировать из пакетов `mypackage.models` или `mypackage.utils`. Кроме того, это будет гарантировать нормальную работу кода, использующего API, даже если внутренняя организация пакета `mypackage` изменится (например, будет удален файл *models.py*).

Чтобы реализовать такой подход, потребуется изменить файл *__init__.py* в каталоге `mypackage`. Когда этот файл импортируется, он фактически становится содержимым модуля `mypackage`. Следовательно, вы можете указать явный API для пакета `mypackage`, налагая ограничения на то, что допускается импортировать в файл *__init__.py*. Поскольку все наши внутренние модули уже указывают `__all__`, можно представить общедоступный интерфейс `mypackage`, осуществляя весь импорт из внутренних модулей и соответственно обновляя список `__all__`.

```
# __init__.py
__all__ = []
from . models import *
__all__ += models.__all__
from . utils import *
__all__ += utils.__all__
```

Ниже приведен код, использующий этот API, который выполняет импорт непосредственно из пакета `mypackage`, а не обращается к внутренним модулям.

```
# api_consumer.py
from mypackage import *

a = Projectile(1.5, 3)
b = Projectile(4, 1.7)
after_a, after_b = simulate_collision(a, b)
```

Самое главное то, что функции, предназначенные исключительно для внутреннего использования, такие как `mypackage.utils._dot_product()`, не будут доступны потребителям API через пакет `mypackage`, поскольку они не были включены в список `__all__`. В силу этого они не импортируются инструкцией `from mypackage import *`. Имена, предназначенные для внутреннего использования, эффективно скрываются.

В целом такой подход отлично работает в ситуациях, когда важно предоставить явный, стабильный API. Но если вы создаете API, предназначенный для использования с собственными модулями, то в функциональности, предоставляемой атрибутом `__all__`, вероятно, нет необхо-

димости, и ее следует избегать. Обычно механизма пространства имен, обеспечиваемого пакетами, вполне достаточно для того, чтобы команда программистов могла коллективно работать с большими объемами кода, который они контролируют, при сохранении разумных границ интерфейсов.

Остерегайтесь использовать инструкцию `import *`

Инструкции импорта типа `from x import y` совершенно понятны, поскольку в качестве источника `y` в них явно указывается пакет или модуль `x`. Импорт с использованием групповых символов, как в инструкции `from foo import *`, также может быть полезен, особенно в интерактивных сеансах работы с Python. Однако использование групповых символов затрудняет понимание кода.

- Инструкция `from foo import *` скрывает источник имен от тех, кто впервые читает код. Если в модуле имеется несколько инструкций `import *`, то для того, чтобы разобраться, где определено то или иное имя, приходится просматривать все модули, на которые имеются ссылки.
- Имена из инструкций `from import *` будут перекрывать любые конфликтующие имена в пределах вмещающего модуля. Это может приводить к странным ошибкам, обусловленным случайным взаимодействием между вашим кодом и перекрывающимися именами из нескольких инструкций `import *`.

Наиболее безопасный подход заключается в том, чтобы избегать использования инструкций `import *` в своем коде и явно импортировать имена с помощью инструкций типа `from x import y`.

Что следует запомнить

- ✦ В языке Python пакеты — это модули, которые содержат другие модули. Пакеты позволяют организовать код в виде отдельных пространств имен с уникальными абсолютными именами модулей, исключающими возможность возникновения конфликтов имен.
- ✦ Простые пакеты определяются путем добавления файла *__init__.py* в каталог, содержащий другие файлы с исходным кодом. Эти файлы становятся дочерними модулями пакета данного каталога. Каталоги пакетов могут содержать другие пакеты.
- ✦ Вы можете предоставить явный API для модуля, перечислив его общедоступные имена в специальном атрибуте `__all__`.

- Вы можете скрыть внутреннюю реализацию пакета, импортируя в файле *__init__.py* пакета лишь общедоступные имена или используя имена внутренних членов, начинающиеся с символа подчеркивания.
- В случае коллективной работы в рамках одной команды или над одной кодовой базой в использовании атрибута `__all__` для определения явных API, вероятно, нет необходимости.

Рекомендация 51. Изолируйте вызывающий код от API, определяя базовое исключение `Exception`

В процессе определения API модуля генерируемые вами исключения являются точно такой же частью вашего интерфейса, как и определяемые вами функции и классы (см. раздел "Рекомендация 14. Использование исключений предпочтительнее возврата значения `None`").

В Python имеется встроенная иерархия исключений для языка и стандартной библиотеки. Наблюдается тенденция использовать встроенные типы исключений для вывода сообщений об ошибках вместо определения собственных новых типов. Например, вы можете генерировать исключение `ValueError` всякий раз, когда функции передается некорректный параметр.

```
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Плотность должна иметь положительное
↳ значение')
    # ...
```

В некоторых случаях в использовании исключения `ValueError` есть смысл, но для API намного эффективнее определять собственную иерархию исключений. Это можно сделать, предоставляя корневой объект `Exception` в своем модуле. Все остальные исключения, генерируемые в данном модуле, могут наследовать от данного корневого исключения.

```
# my_module.py
class Error(Exception):
    """Базовый класс для всех исключений, генерируемых
       данным модулем."""

class InvalidDensityError(Error):
    """Возникла проблема с предоставлением значения density."""
```

Определение корневого исключения в модуле упрощает коду, использующему ваш API, перехват исключений, которые генерируются вами

намеренно. Например, приведенный ниже код, использующий ваш API, вызывает функцию с помощью инструкции try/except, перехватывающей корневое исключение.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error as e:
    logging.error('Неожиданная ошибка: %s', e)
```

Данная инструкция try/except не дает исключениям, возникающим в вашем API, всплывать слишком высоко и тем самым нарушать работу вызывающей программы. Она изолирует вызывающий код от вашего API. Подобная изоляция обладает тремя преимуществами.

Во-первых, корневые исключения позволяют вызывающему коду распознавать ситуации, в которых проблема связана с использованием API. При соответствующем использовании вашего API вызывающий код может перехватывать и обрабатывать различные исключения, которые вы генерируете намеренно. Необработанные исключения распространяются вверх до изолирующего блока except, который перехватывает корневое исключение вашего модуля. Этот блок может известить потребителя API об исключении, предоставляя ему возможность добавить обработку исключений данного типа.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Ошибка в вызывающем коде: %s', e)
```

Вторым преимуществом использования корневых исключений является то, что это облегчает нахождение ошибок в коде модуля API. Если ваш код намеренно генерирует лишь те исключения, которые вы определили в иерархии модуля, то это означает, что все остальные типы исключений, сгенерированных в вашем модуле, относятся к категории тех, которые вы не намеревались генерировать, и именно они указывают на ошибки в вашем коде.

Использование инструкции try/except, как в приведенном выше примере, не будет изолировать потребителей API от ошибок в коде вашего API. Чтобы обеспечить такую изоляцию, вызывающий код должен добавить еще один блок except, который перехватывает исключения, соответствующие базовому классу Exception Python. Это позволит потребителям API обнаруживать в реализации модуля ошибки, подлежащие устранению.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Ошибка в вызывающем коде: %s', e)
except Exception as e:
    logging.error('Ошибка в коде API: %s', e)
    raise
```

Третье преимущество использования корневых исключений касается возможностей будущих проверок вашего API. Со временем у вас может возникнуть потребность в расширении API с целью предоставления более специфических исключений в определенных ситуациях. Например, вы можете добавить подкласс `Exception`, индицирующий состояние ошибки в случае предоставления отрицательных значений параметра `density`.

```
# my_module.py
class NegativeDensityError(InvalidDensityError):
    """Было предоставлено отрицательное значение
       параметра density."""

def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError
```

Вызывающий код будет работать в точности так, как и прежде, поскольку он уже перехватывает исключения `InvalidDensityError` (родственный по отношению к исключению `NegativeDensityError` класс). В будущем вызывающий код может выделить новый тип исключения в качестве специального случая и соответственно изменить свое поведение.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError as e:
    raise ValueError('Должно быть предоставлено неотрицательное
↳ значение плотности') from e
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Ошибка в вызывающем коде: %s', e)
except Exception as e:
    logging.error('Ошибка в коде API: %s', e)
raise
```

Возможности организации проверок в коде в будущем можно расширить, предоставляя более широкий набор исключений непосредственно под корневым исключением. Допустим, у вас было предусмотрено несколько наборов ошибок, один из которых связан с расчетом веса (`weight`), другой — с расчетом объема (`volume`) и третий — с расчетом плотности (`density`).

```
# my_module.py
class WeightError(Error):
    """Базовый класс для ошибок при расчете веса."""

class VolumeError(Error):
    """Базовый класс для ошибок при расчете объема."""

class DensityError(Error):
    """Базовый класс для ошибок при расчете плотности."""
```

Специфические исключения будут наследовать от этих общих исключений. Каждое промежуточное исключение действует в качестве разновидности корневого исключения. Такой подход упрощает изолирование слоев вызывающего кода от кода API, основанного на широкой функциональности. Это гораздо лучше, чем заставлять вызывающий код каждый раз перехватывать длинный список весьма специфических исключений, соответствующих подклассам `Exception`.

Что следует запомнить

- Определение корневых исключений для модулей позволяет потребителям ваших API изолировать себя от них.
- Перехват корневых исключений может облегчить нахождение ошибок в коде, использующем ваш API.
- Перехват исключений базового класса `Exception` Python может облегчить нахождение ошибок в ваших реализациях API.
- Промежуточные корневые исключения позволяют добавлять специфические типы исключений в будущем, не нарушая работы кода, использующего ваш API.

Рекомендация 52. Знайте, как устранять циклические зависимости

В процессе коллективной разработки вы неизбежно столкнетесь с возникновением зависимостей между модулями. Это может происходить даже в тех случаях, когда вы самостоятельно разрабатываете различные части одной программы.

Предположим, вы хотите, чтобы ваше приложение с графическим пользовательским интерфейсом отображало диалоговое окно, в котором можно выбрать папку для сохранения документа. Отображаемые в диалоговом окне данные могут задаваться посредством аргументов, передаваемых вашим обработчикам событий. Однако для того, чтобы определить, как именно следует визуализировать информацию, диалоговое окно должно считывать также глобальное состояние, например информацию о предпочтениях пользователя.

Определим диалоговое окно, которое извлекает информацию о заданном по умолчанию месте сохранения файла из глобальных установок.

```
# dialog.py
import app
class Dialog(object):
    def __init__(self, save_dir):
        self.save_dir = save_dir
    # ...

save_dialog = Dialog(app.prefs.get('save_dir'))

def show():
    # ...
```

Проблема в том, что модуль `app`, который содержит объект `prefs`, импортирует также класс `dialog`, необходимый для отображения диалогового окна при запуске программы.

```
# app.py
import dialog

class Prefs(object):
    # ...
    def get(self, name):
        # ...

prefs = Prefs()
dialog.show()
```

Здесь налицо циклическая зависимость. Если вы попытаетесь использовать модуль `app` в своей основной программе, то при его импортировании возникнет исключение.

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    import app
  File "app.py", line 4, in <module>
    import dialog
```

```
  File "dialog.py", line 16, in <module>
    save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: 'module' object has no attribute 'prefs'
```

Чтобы понять, что здесь происходит, вам необходимо более детально ознакомиться с работой механизма импорта Python. Ниже перечислены операции (начиная с наиболее низкоуровневых), которые выполняет Python при импортировании модуля:

1. выполняет поиск модуля в местах, указанных в `sys.path`;
2. загружает код из модуля и убеждается в его компиляции;
3. создает объект соответствующего модуля;
4. вставляет модуль в `sys.modules`;
5. выполняет код в объекте модуля для определения его содержимого.

Суть проблемы циклических зависимостей в том, что атрибуты модуля не определены до тех пор, пока не выполнится код для этих атрибутов (шаг 5). Однако модуль может быть загружен инструкцией `import` сразу же после того, как он вставлен в `sys.modules` (шаг 4).

В приведенном выше примере модуль `app` импортирует `dialog`, прежде чем что-либо определять, а затем модуль `dialog` импортирует модуль `app`. Поскольку `app` все еще не закончил выполнение (в данный момент он импортирует модуль `dialog`), он представляет собой всего лишь пустую оболочку (после шага 4). Генерируется исключение `AttributeError` (для модуля `dialog` на шаге 5), поскольку код, который определяет `prefs`, пока что не выполнялся (шаг 5 для `app` еще не завершился).

Наилучшее решение этой проблемы — выполнить рефакторинг кода, чтобы структура данных `prefs` оказалась в самом низу дерева зависимостей. Далее как `app`, так и `dialog` могут импортировать один и тот же модуль `utility` и избежать циклических зависимостей. Но такое четкое разделение не всегда возможно или же требует выполнения такого большого объема работы по рефакторингу кода, что затраченные усилия себя не оправдают.

Существуют еще три способа разрыва циклических зависимостей.

Изменение порядка импортирования модулей

Один из вышеупомянутых способов — это изменение порядка следования инструкций `import`. Например, если вы перенесете операцию импортирования модуля `dialog` в конец модуля `app`, чтобы дать возможность выполниться содержимому последнего, то исключение `AttributeError` не возникнет.

```
# app.py
class Prefs(object):
    # ...

prefs = Prefs()

import dialog # Перемещен
dialog.show()
```

Этот код работает, поскольку в случае поздней загрузки модуля `dialog` при попытке выполнения в нем рекурсивного импорта модуля `app` будет обнаружено, что объект `app.prefs` уже определен (шаг 5 для `app` в основном уже выполнен).

Несмотря на то что такой подход позволяет избежать исключений `AttributeError`, он не соответствует рекомендациям по стилю PEP 8 (см. раздел "Рекомендация 2. Руководствуйтесь правилами стилевого оформления программ, изложенными в документе PEP 8"). В соответствии с этими правилами инструкции `import` всегда должны располагаться в самом начале файлов Python. Благодаря этому тем, кто впервые читает ваш код, будет легче разобраться в зависимостях, существующих между модулями. Это также гарантирует, что любой модуль, от которого зависит ваш код, будет доступен всему коду вашего модуля.

Размещение инструкций `import` в конце файла повышает хрупкость кода и может увеличивать вероятность того, что даже небольшие изменения в порядке следования вашего кода сделают его полностью неработоспособным.

Таким образом, следует избегать изменения порядка импортирования модулей для разрешения проблем, связанных с циклическими зависимостями.

Импортирование, конфигурирование, выполнение

Другое решение проблемы циклического импорта заключается в сведении к минимуму вероятности возникновения побочных эффектов во время выполнения импорта. Старайтесь писать код так, чтобы ваши модули лишь определяли функции, классы и константы. Избегайте фактического выполнения каких-либо функций при импорте. Необходимо также предусматривать, чтобы каждый модуль предоставлял функцию `configure()`, которую вы вызываете, по завершении импорта всех других модулей. Назначением этой функции является подготовка состояния каждого модуля путем обращения к атрибутам других модулей. Вы выполняете функцию после того, как все модули были уже импортированы (завершен шаг 5), и поэтому все атрибуты должны быть определены.

Переопределим модуль dialog таким образом, чтобы доступ к объекту prefs осуществлялся лишь при вызове функции configure().

```
# dialog.py
import app

class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    # ...

def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

Мы также переопределим модуль app таким образом, чтобы во время импорта в нем не выполнялись никакие существенные действия.

```
# app.py
import dialog

class Prefs(object):
    # ...

prefs = Prefs()

def configure():
    # ...
```

Наконец, в модуле main имеются три различные фазы выполнения: импортирование всего необходимого, конфигурирование всего необходимого и выполнение первого действия.

```
# main.py
import app
import dialog

app.configure()
dialog.configure()

dialog.show()
```

Описанный подход годится для многих ситуаций и позволяет использовать такие шаблоны проектирования, как *внедрение зависимостей*. Но иногда трудно структурировать код так, чтобы настройку конфигурации модуля можно было выделить в отдельный этап. Кроме того, наличие

двух отдельных фаз в модуле может затруднить чтение кода, поскольку это отделяет определение объектов от настройки их конфигураций.

Динамический импорт

Третье — и часто простейшее — решение проблемы циклического импорта состоит в том, чтобы использовать инструкцию `import` в функции или методе. Это называется *динамическим импортом*, поскольку модуль импортируется во время выполнения программы, а не при первоначальном запуске программы и инициализации ее модулей.

Ниже мы переопределим модуль `dialog` так, чтобы использовать динамический импорт. Здесь вместо импортирования модуля `app` модулем `dialog` во время инициализации он импортируется функцией `dialog.show()` во время выполнения.

```
# dialog.py
class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    import app # Dynamic import
    save_dialog.save_dir = app.prefs.get('save_dir')
    # ...
```

Теперь модуль `app` может быть использован в том же виде, что и в исходном примере: модуль `dialog` импортируется в начале модуля `app`, а функция `dialog.show()` вызывается в конце.

```
# app.py
import dialog

class Prefs(object):
    # ...
prefs = Prefs()

dialog.show()
```

При таком подходе результирующий эффект на стадиях импортирования, конфигурирования и выполнения модулей аналогичен рассмотренному выше. Отличие в том, что это не требует внесения каких-либо структурных изменений, касающихся способа определения и импортирования модулей. Вы просто откладываете циклический импорт до тех пор, когда вам действительно не потребуется доступ к другому модулю. К этому моменту вы можете быть достаточно уверены в том,

что все другие модули уже инициализированы (шаг 5 уже завершен для всех модулей).

В целом лучше избегать динамического импорта. Накладными расходами операций импорта нельзя пренебрегать, особенно в случае интенсивных циклов. Кроме того, откладывая выполнение, динамический импорт может преподнести вам сюрпризы на этапе выполнения, такие как исключения `SyntaxError`, возникающие спустя длительное время после запуска программы (о том, как избежать подобных ситуаций, см. в разделе "Рекомендация 56. Тестируйте любой код с помощью модуля `unittest`"). Однако даже с учетом указанных недостатков данный способ часто оказывается лучше альтернативы в виде реструктуризации всей программы.

Что следует запомнить

- Циклические зависимости возникают в тех случаях, когда два модуля должны вызывать друг друга на стадии импорта. Это может привести к краху программы уже при запуске.
- Наилучшим способом разрыва циклических зависимостей является рефакторинг кода для вынесения взаимных зависимостей в отдельные модули в нижней части дерева зависимостей.
- Динамический импорт — простейшее решение, позволяющее устранить циклические зависимости между модулями с минимальными усилиями по выполнению рефакторинга кода.

Рекомендация 53. Используйте виртуальные среды для изолированных и воспроизводимых зависимостей

Процесс создания крупных программ повышенной сложности часто приводит к тому, что приходится опираться на различные пакеты, разработанные сообществом пользователей Python (см. раздел "Рекомендация 48. Знайте, где искать модули, разработанные сообществом Python"). При этом вы будете устанавливать такие пакеты, как `pytz`, `numpy` и многие другие, используя систему управления пакетами `pip`.

Проблема в том, что по умолчанию `pip` устанавливает новые пакеты в глобальных расположениях. В результате этого вновь установленные модули будут оказывать воздействие на все программы на языке Python, имеющиеся в вашей системе. Теоретически это не должно вызывать никаких проблем. Если вы устанавливаете пакет, но не импортируете его, то как он может влиять на ваши программы?

Проблема коренится в транзитивных зависимостях — пакетах, от которых зависят устанавливаемые вами пакеты. Например, вы можете увидеть зависимости пакета Sphinx после его установки, запросив об этом pip.

```
$ pip3 show Sphinx
---
Name: Sphinx
Version: 1.2.2
Location: /usr/local/lib/python3.4/site-packages
Requires: docutils, Jinja2, Pygments
```

Если вы установите другой пакет, например flask, то увидите, что он также зависит от пакета Jinja2.

```
$ pip3 show flask
---
Name: Flask
Version: 0.10.1
Location: /usr/local/lib/python3.4/site-packages
Requires: Werkzeug, Jinja2, itsdangerous
```

Причиной конфликта может стать возникновение расхождений между пакетами Sphinx и flask с течением времени. Возможно, сейчас им обоим требуется одна и та же версия пакета Jinja2, и все работает нормально. Но спустя полгода или год может быть выпущена новая версия Jinja2, которая заставляет пользователей библиотеки вносить изменения в свой код. Если вы обновите свою глобальную версию пакета Jinja2 с помощью команды pip install --upgrade, то может оказаться, что пакет Sphinx не сможет работать, тогда как на пакете flask это никак не отразится.

Такие нарушения работы кода возникают потому, что Python позволяет иметь в каждый момент времени лишь одну установленную глобальную версию модуля. Если один из установленных вами пакетов должен использовать новую версию, а другой — старую, то ваша система не будет работать так, как надо.

Подобные программные сбои могут случаться даже тогда, когда в процессе сопровождения кода делается все возможное для того, чтобы сохранить совместимость API при переходе от одной версии к другой (см. раздел “Рекомендация 50. Используйте пакеты для организации модулей и предоставления стабильных API”). Новые версии библиотеки могут незначительно изменять поведение модулей, на которое опирается код, использующий данный API. Пользователи системы могут обновить один пакет до новой версии, но не другие, ибо это может привести

к разрыву зависимостей. Таким образом, существует постоянная опасность того, что вы почувствуете, как земля уходит из-под ваших ног.

Эти трудности только увеличиваются в случае коллективной разработки, когда ваши коллеги работают на других компьютерах. Разумно предположить, что версии Python и глобальные пакеты, которые они установили на своих машинах, будут незначительно отличаться от ваших. Это может породить крайне неприятную ситуацию, когда кодовая база прекрасно работает на компьютере одного программиста, но отказывается работать на компьютере другого.

Ключом к решению любых проблем подобного рода является инструмент `pyvenv`, предоставляющий виртуальные среды. Начиная с версии Python 3.4 утилита командной строки `pyvenv` доступна по умолчанию вместе с установкой Python (она также доступна по команде `python -m venv`). Предыдущие версии Python требуют установки отдельного пакета (с помощью команды `pip install virtualenv`) и использования утилиты командной строки под названием `virtualenv`.

Средство `pyvenv` позволяет создавать изолированные версии среды Python. Используя `pyvenv`, вы сможете иметь одновременно много версий одного и того же пакета, установленных в одной системе, которые не конфликтуют между собой. Эта дает возможность работать со многими проектами и использовать множество различных инструментов на одном и том же компьютере.

Средство `pyvenv` обеспечивает это путем установки версий пакетов и их зависимостей в полностью независимые структуры каталогов. Это дает возможность воспроизводить ту среду Python, относительно которой у вас есть уверенность, что она будет работать с вашим кодом. Это надежный способ, позволяющий избежать неожиданной потери работоспособности кода.

Команда pyvenv

Рассмотрим пример, который поможет вам научиться эффективно использовать средство `pyvenv`. Прежде чем использовать этот инструмент, важно, чтобы вам был понятен смысл команды `python3`, которую вы вводите в командной строке своей системы. На моем компьютере `python3` располагается в каталоге `/usr/local/bin` и запускает версию 3.4.2 (см. раздел "Рекомендация 1. Следите за тем, какую версию Python вы используете").

```
$ which python3
/usr/local/bin/python3
$ python3 --version
Python 3.4.2
```

Чтобы продемонстрировать настройку моей среды, я могу проверить, что выполнение команды для импортирования модуля pytz не приведет к ошибке. Это работает, поскольку пакет pytz уже установлен у меня в качестве глобального модуля.

```
$ python3 -c 'import pytz'
$
```

Далее я использую pyvenv для создания новой виртуальной среды под названием myproject.

Каждая виртуальная среда должна существовать в собственном уникальном каталоге. Результатом выполнения команды является создание дерева каталогов и файлов.

```
$ pyvenv /tmp/myproject
$ cd /tmp/myproject
$ ls
bin    include    lib    pyvenv.cfg
```

Чтобы начать использовать виртуальную среду, я применю команду source моей оболочки к сценарию bin/activate. Сценарий activate изменяет все переменные моей среды таким образом, чтобы они соответствовали виртуальной среде. Кроме того, он изменяет приглашение моей командной строки, включая в него имя виртуальной среды ('myproject'), чтобы было предельно ясно, в какой именно среде я работаю.

```
$ source bin/activate
(myproject)$
```

После того как среда активизирована, вы сможете увидеть, что путь к командной строке python3 переместился в каталог виртуальной среды.

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ ls -l /tmp/myproject/bin/python3
... -> /tmp/myproject/bin/python3.4
(myproject)$ ls -l /tmp/myproject/bin/python3.4
... -> /usr/local/bin/python3.4
```

Это гарантирует, что изменения во внешней системе не будут влиять на виртуальную среду. Даже если внешняя система обновляет свою версию по умолчанию, вызываемую командой python3, до версии 3.5, моя виртуальная среда будет по-прежнему явно указывать на версию 3.4.

Среда, которую я создал с помощью средства pyvenv, запускается без каких-либо установленных пакетов, не считая pip и setuptools. Попытка использовать пакет pytz, который был установлен в качестве гло-

бального модуля во внешней системе, окажется неудачной, поскольку он неизвестен виртуальной среде.

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'pytz'
```

Я могу использовать систему управления пакетами `pip` для установки модуля `pytz` в мою виртуальную среду.

```
(myproject)$ pip3 install pytz
```

Как только модуль `pytz` установлен, я могу убедиться в том, что он работает, используя ту же тестовую команду импорта.

```
(myproject)$ python3 -c 'import pytz'
(myproject)$
```

Чтобы вернуться в свою заданную по умолчанию систему по окончании работы в виртуальной среде, используйте команду `deactivate`. Это восстановит вашу среду до заданной в системе по умолчанию, включая расположение средства командной строки `python3`.

```
(myproject)$ deactivate
$ which python3
/usr/local/bin/python3
```

Если вам когда-либо вновь понадобится работать в виртуальной среде, достаточно будет, как и прежде, выполнить команду `source bin/activate` в соответствующем каталоге.

Воспроизведение зависимостей

Организовав виртуальную среду, вы можете устанавливать в ней пакеты с помощью `pip` по мере необходимости. В конечном счете вы можете скопировать свою среду в другое место. Допустим, вы хотите воспроизвести свою среду разработки на производственном сервере. Или же вы хотите воспроизвести среду другого разработчика на своем компьютере, чтобы иметь возможность выполнять его код.

Средство `pyvenv` позволяет легко справляться с подобными ситуациями. Используя команду `pip freeze`, сохраните все явные зависимости своих пакетов в файле. В соответствии с принятым соглашением этот файл называется *requirements.txt*.

```
(myproject)$ pip3 freeze > requirements.txt
(myproject)$ cat requirements.txt
numpy==1.8.2
```

```
pytz==2014.4
requests==2.3.0
```

Допустим, что вы хотите установить другую виртуальную среду, которая соответствует среде `myproject`. Вы можете создать новый каталог и активизировать среду, как мы делали до этого, используя средства `pyvenv` и `activate`.

```
$ pyvenv /tmp/otherproject
$ cd /tmp/otherproject
$ source bin/activate
(otherproject)$
```

В новой среде не будут установлены никакие дополнительные пакеты.

```
(otherproject)$ pip3 list
pip (1.5.6)
setuptools (2.1)
```

Можно установить все пакеты из первой среды, выполнив команду `pip install` для файла *requirements.txt*, который был сгенерирован с помощью команды `pip freeze`.

```
(otherproject)$ pip3 install -r /tmp/myproject/requirements.txt
```

Эта команда немного потрудится, пока не извлечет и не установит все пакеты, необходимые для воспроизведения первой среды. Как только эта работа будет выполнена, перечисление набора установленных пакетов во второй виртуальной среде приведет к тому же списку зависимостей, что и в случае первой виртуальной среды.

```
(otherproject)$ pip list
numpy (1.8.2)
pip (1.5.6)
pytz (2014.4)
requests (2.3.0)
setuptools (2.1)
```

Использование файла *requirements.txt* идеально подходит для организации коллективной работы посредством системы управления версиями. Вы можете принять внесенные в код изменения одновременно с обновлением списка пакетных зависимостей с гарантией того, что это будет сделано синхронно.

Неприятным моментом при работе с виртуальными средами является то, что их перемещение разрушает все связи, поскольку все пути (например, путь к `python3`) жестко закодированы в каталоге установки среды. Но это не имеет значения. Единственная цель использования виртуальных сред состоит в том, чтобы упростить воспроизведение одной

и той же установки. Вместо перемещения каталога виртуальной среды достаточно вывести в файл дамп старого каталога, создать новый каталог в другом месте и переустановить все из файла *requirements.txt*.

Что следует запомнить

- Виртуальные среды позволяют использовать средство `pip` для установки нескольких версий одного и того же пакета на одной и той же машине так, что между ними не будут возникать конфликты.
- Виртуальные среды создаются с помощью команды `pyvenv`, активизируются с помощью команды `source bin/activate` и отключаются с помощью команды `deactivate`.
- Вы можете вывести дамп всех требований к среде с помощью команды `pip freeze`. Для воспроизведения среды следует предоставить команде `pip install -r` файл *requirements.txt*.
- В версиях Python ниже 3.4 средство `pyvenv` должно загружаться и устанавливаться отдельно. Средство командной строки называется не `pyvenv`, а `virtualenv`.

8 Производство

Ввод программы на языке Python в эксплуатацию требует ее переноса из среды разработки в производственную среду. Поддержка столь несопоставимых сред может быть затруднительной. Создание программ, надежно работающих в различных ситуациях, столь же важно, как и создание программ, обеспечивающих корректную функциональность.

Вашей целью является выпуск программ на Python для массового использования с гарантией устойчивой работы. В Python имеются встроенные модули, назначение которых — стабилизация работы ваших программ. Он предоставляет средства, обеспечивающие возможности отладки, оптимизации и тестирования программ для улучшения их качества и повышения производительности на этапе выполнения.

Рекомендация 54. Используйте код с областью видимости модуля для конфигурирования сред развертывания

Среда развертывания — это конфигурация системы, в которой выполняется ваша программа. Любая программа имеет по крайней мере одну среду развертывания — *производственную*. Первоочередной задачей написания программы является ее запуск в производственной среде и получение некоторых результатов.

При написании программы и внесении в нее изменений требуется, чтобы ее можно было выполнять на компьютере, который вы используете для разработки. Конфигурация вашей среды разработки может значительно отличаться от конфигурации производственной среды. Например, вы можете писать программу для суперкомпьютеров, используя рабочую станцию Linux.

Инструменты наподобие `pyvenv` (см. раздел "Рекомендация 53. Используйте виртуальные среды для изолированных и воспроизводимых зависимостей") позволяют очень просто добиться того, чтобы во всех

средах были установлены одни и те же пакеты Python. Проблема в том, что относительно производственной среды часто приходится делать множество предположений, которые трудно воспроизвести в среде разработки.

Предположим, вы хотите выполнить свою программу в контейнере веб-сервера и предоставить ей доступ к базе данных. Это означает, что каждый раз, когда вы захотите изменить программный код, вам придется запускать контейнер сервера, настраивать базу данных и проходить парольную проверку. Связанные с этим накладные расходы слишком высоки, если все, что вам надо, — это убедиться в работоспособности программы после изменения всего лишь одной строки кода.

Лучший способ решения подобных проблем — предоставлять различную функциональность в зависимости от среды развертывания, переопределяя части программы на стадии запуска. Например, у вас могут быть два различных файла *__main__* — один для производственной среды, а другой для разработки.

```
# dev_main.py
TESTING = True
import db_connection
db = db_connection.Database()

# prod_main.py
TESTING = False
import db_connection
db = db_connection.Database()
```

Единственным отличием этих файлов является значение константы `TESTING`. Другие модули вашей программы могут импортировать модуль `__main__` и использовать значение `TESTING` для принятия решения о том, как они должны определять собственные атрибуты.

```
# db_connection.py
import __main__

class TestingDatabase(object):
    # ...

class RealDatabase(object):
    # ...

if __main__.TESTING:
    Database = TestingDatabase
else:
    Database = RealDatabase
```

Основной момент, на который следует обратить внимание, — это то, что код, выполняющийся в области видимости модуля (не в теле функции или метода), является обычным кодом Python. Для принятия решения о том, как модуль должен определять имена, можно использовать инструкцию `if` на уровне модуля. Это упрощает настройку модуля для различных сред развертывания. Вы можете избежать повторения нескольких вариантов кода, соответствующих различным дорогостоящим предположениям, например относительно конфигурации базы данных, когда в них нет необходимости. Допускается внедрять фиктивные реализации, которые упрощают интерактивную разработку и тестирование программ (раздел "Рекомендация 56. Тестируйте любой код с помощью модуля `unittest`").

Примечание

Как только ваши среды развертывания начинают усложняться, стоит подумать о том, чтобы вынести их из констант Python (наподобие `TESTING`) в отдельные конфигурационные файлы. Такие инструменты, как встроенный модуль `configparser`, позволяют поддерживать производственные конфигурации отдельно от кода — особенность, имеющая ключевое значение для сотрудничества с оперативным персоналом.

Этот подход может быть использован не только для решения проблем, связанных с необходимостью учета различий в конфигурациях сред развертывания. Например, если вы знаете, что программа должна работать с использованием разных конфигураций на хост-платформе, то можете проверять модуль `sys`, прежде чем определять конструкции верхнего уровня в модуле.

```
# db_connection.py
import sys
class Win32Database(object):
# ...
class PosixDatabase(object):
# ...
if sys.platform.startswith('win32'):
Database = Win32Database
else:
Database = PosixDatabase
```

Точно так же для управления определениями модуля могут быть использованы переменные среды из списка `os.environ`.

Что следует запомнить

- Программы часто должны выполняться во многих средах развертывания, каждая из которых требует выдвижения отдельных предположений и использования отдельных конфигураций.
- Вы можете адаптировать содержимое модуля к различным средам развертывания, используя обычные инструкции Python в области видимости модуля.
- Содержимое модуля может быть продуктом любых внешних условий, включая те, которые определяются путем интроспекции с помощью модулей `sys` и `os`.

Рекомендация 55. Используйте строки `repr` для вывода отладочной информации

В процессе отладки программ на Python функция `print()` (или вывод посредством встроенного модуля `logging`) может предоставить удивительно много информации. Во многих случаях к внутренним функциям Python можно легко получить доступ с помощью простых атрибутов (см. раздел "Рекомендация 27. Предпочитайте общедоступные атрибуты закрытым"). Все, что вам остается сделать, — вывести на печать изменения в состоянии вашей программы на этапе выполнения и посмотреть, что идет не так.

Функция `print()` выводит информацию о переданном ей объекте в удобном для чтения строковом представлении. Например, в случае печати простой строки ее содержимое будет выведено без кавычек, в которые она заключена.

```
print('foo bar')
>>>
foo bar
```

Это эквивалентно использованию строки форматирования `'%s'` и оператора `%`.

```
print('%s' % 'foo bar')
>>>
foo bar
```

Проблема в том, что строка значения в удобочитаемом виде ничего не говорит о фактическом типе этого значения. Например, вывод в такой форме не позволяет отличить число `5` от строки `'5'`.

```
print(5)
print('5')
```

```
>>>
5
5
```

Если функция `print()` используется в целях отладки, то подобное отличие типов имеет значение. Что вам действительно потребуется в процессе отладки, так это представление объекта, обеспечиваемое функцией `repr()`. Встроенная функция `repr()` возвращает *отображаемое представление* (printable representation) объекта, являющееся его наиболее понятным строковым представлением. Строка, возвращаемая функцией `repr()` для встроенных типов, является действительным выражением Python.

```
a = '\x07'
print(repr(a))

>>>
'\x07'
```

Передав значение, возвращенное функцией `repr()`, встроенной функции `eval()`, вы получите тот же объект `Python`, с которого начали (разумеется, на практике функцией `eval()` следует пользоваться с предельной осторожностью).

```
b = eval(repr(a))
assert a == b
```

Выполняя отладку с помощью функции `print()`, вы всегда должны предварительно применить к выводимому значению функцию `repr()`, чтобы устранить возможные неоднозначности в определении типа.

```
print(repr(5))
print(repr('5'))

>>>
5
'5'
```

Это эквивалентно использованию строки форматирования `'%r'` и оператора `%`.

```
print('%r' % 5)
print('%r' % '5')

>>>
5
'5'
```

Для динамических объектов Python установленное по умолчанию строковое представление совпадает с представлением, обеспечиваемым функцией `repr()`. Это означает, что передача динамического объекта функции `print()` приведет к получению правильного результата, и в явном использовании функции `repr()` нет необходимости. К сожалению, значение, выводимое по умолчанию функцией `repr()` для экземпляров объектов, не очень выручает. Например, ниже мы определим простой класс, а затем выведем его значение с помощью функции `print()`.

```
class OpaqueClass(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = OpaqueClass(1, 2)
print(obj)

>>>
<__main__.OpaqueClass object at 0x107880ba8>
```

Этот вывод не может быть передан функции `eval()` и ничего не сообщает о полях экземпляра объекта.

Существуют два решения этой проблемы. Если у вас есть контроль над классом, определите собственный специальный метод `__repr__()`, возвращающий строку, которая содержит выражение Python, воссоздающее объект.

Ниже мы определим такую функцию для класса `OpaqueClass`.

```
class BetterClass(object):
    def __init__(self, x, y):
        # ...

    def __repr__(self):
        return 'BetterClass(%d, %d)' % (self.x, self.y)
```

Теперь выводимое значение гораздо полезнее.

```
obj = BetterClass(1, 2)
print(obj)

>>>
BetterClass(1, 2)
```

В случае отсутствия контроля над определением класса вы можете обратиться к словарю экземпляра объекта, который хранится в атрибуте `__dict__`. Ниже мы выведем содержимое экземпляра класса `OpaqueClass`.

```
obj = OpaqueClass(4, 5)
print(obj.__dict__)

>>>
{'y': 5, 'x': 4}
```

Что следует запомнить

- Вызывая функцию `print()` для встроенных типов Python, вы получаете удобное для чтения человеком строковое представление значения, которое, однако, скрывает информацию о типе.
- Вызывая функцию `repr()` для встроенных типов Python, вы получаете отображаемое строковое представление значения. Такие строки можно передавать встроенной функции `eval()` для получения исходного значения.
- В форматированных строках символы `%s` обеспечивают получение удобных для чтения человеком строк типа `str`. Символы `%r` обеспечивают получение отображаемых строк типа `repr`.
- Вы можете скорректировать отображаемое представление класса и предоставить более подробную отладочную информацию, определив метод `__repr__()`.
- Вы можете просмотреть внутреннее содержимое любого объекта с помощью атрибута `__dict__`.

Рекомендация 56. Тестируйте любой код с помощью модуля `unittest`

В языке Python статические проверки типа не предусмотрены. В компиляторе отсутствуют какие-либо средства, гарантирующие, что ваша программа будет работать, когда вы ее запустите.

В Python вы не знаете, будут ли функции, которые вызывает ваша программа, определены во время выполнения, даже если их существование очевидно из исходного кода. Такое динамическое поведение является одновременно и благом, и проклятием.

Многие из тех, кто программирует на Python, сказали бы, что динамическое поведение себя оправдывает, поскольку краткость и простота обеспечивают выигрыш в производительности. Однако большинству людей приходилось слышать хотя бы одну ужасающую историю о том, как причину таинственной ошибки, возникающей в программе на Python во время выполнения, так и не удалось обнаружить. В одном из известных мне случаев подобного рода возникновение исключения

`SyntaxError` уже в условиях эксплуатации являлось побочным эффектом динамического импорта (см. раздел "Рекомендация 52. Знайте, как устранять циклические зависимости"). Столкнувшись с такой ситуацией, один мой знакомый программист вообще зарекся когда-либо использовать Python.

Меня только удивляет, почему программу не протестировали самым тщательным образом, прежде чем запускать в производство? Безопасность типов — это еще не все. Следует всегда тестировать код, на каком бы языке он ни был написан. Тем не менее я согласен, что огромная разница между Python и другими языками программирования состоит в том, что единственным способом быть уверенным в том, что программа на языке Python работает правильно, является написание тестов. Никаких способов статической проверки, которые давали бы вам основания чувствовать себя в этом смысле уверенно, не существует.

К счастью, то же самое динамическое поведение, которое делает невозможной статическую проверку типов в Python, одновременно чрезвычайно упрощает написание тестов для кода. Вы можете задействовать динамическую природу Python и легкость переопределения поведения для реализации тестов и получения доказательств того, что программа работает так, как ожидалось.

Вы должны относиться к тестам как к своеобразной страховке своего кода. Хорошо продуманные тесты дают уверенность в корректности кода. Если вы выполнили рефакторинг кода или расширили его, то с помощью тестов легче обнаружить, как изменилось поведение программы. Как бы странно это ни звучало, но дополнительные усилия по разработке надежных тестов облегчают внесение изменений в код Python, а не усложняют этот процесс.

Проще всего подготавливать тесты, используя встроенный модуль `unittest`. Предположим, у вас имеется следующая функция `utility()`, определенная в файле *utils.py*.

```
# utils.py
def to_str(data):
    if isinstance(data, str):
        return data
    elif isinstance(data, bytes):
        return data.decode('utf-8')
    else:
        raise TypeError('Ожидается тип str или bytes, '
                        'фактически: %r' % data)
```

Чтобы определить тесты, я создаю еще один файл с именем *test_utils.py* или *utils_test.py*, который содержит тесты для каждого из ожидаемых вариантов поведения.

```
# utils_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsTestCase(TestCase):
    def test_to_str_bytes(self):
        self.assertEqual('hello', to_str(b'hello'))

    def test_to_str_str(self):
        self.assertEqual('hello', to_str('hello'))

    def test_to_str_bad(self):
        self.assertRaises(TypeError, to_str, object())

if __name__ == '__main__':
    main()
```

Тесты организуются в классы `TestCase`. Каждый тест представляет собой метод, имя которого начинается со слова `test`. Если выполнение тестового метода не сопровождается возбуждением исключений любого рода (включая исключение `AssertionError`, генерируемое инструкциями `assert`), то считается, что тест пройден.

Класс `TestCase` предоставляет вспомогательные методы для создания в тесте таких утверждений, как `assertEqual` для проверки равенства, `assertTrue` для проверки булевых выражений и `assertRaises` для проверки того, что в соответствующих случаях генерируются исключения (для получения более подробной информации см. `help(TestCase)`). Для выполнения более удобочитаемых тестов можно определить собственные вспомогательные методы в подклассах `TestCase`; нужно лишь следить за тем, чтобы имена ваших методов не начинались со слова `test`.

Примечание

Другая распространенная практика при написании тестов — это использование фиктивных функций и классов для создание заглушек. С этой целью Python 3 предоставляет встроенный модуль `unittest.mock`, также доступный для Python 2 в виде пакета с открытым исходным кодом.

Иногда ваши классы `TestCase` могут нуждаться в настройке тестовой среды до выполнения тестовых методов. Для этого можно переопределить методы `setUp()` и `tearDown()`. Вызовы данных методов соответственно предшествуют вызовам каждого из тестовых методов и следуют за ними, и они позволяют вам быть уверенным в том, что каждый тест выполняется в изоляции (важный принцип лучшей практики тестирования). Например, ниже я определяю класс `TestCase`, который создает

временный каталог перед каждым тестом и удаляет его содержимое, когда тест завершается.

```
class MyTest(TestCase):
    def setUp(self):
        self.test_dir = TemporaryDirectory()
    def tearDown(self):
        self.test_dir.cleanup()
    # Далее следуют тестовые методы
    # ...
```

Обычно я определяю по одному классу `TestCase` для каждого набора родственных тестов. Иногда у меня имеется по одному классу `TestCase` для каждой функции, имеющей многочисленные граничные случаи. Бывает и так, что один класс `TestCase` охватывает все функции одного модуля. Я также создаю один класс `TestCase` для тестирования единственного класса и всех его методов.

По мере усложнения программы вам понадобятся дополнительные тесты для проверки взаимодействия между модулями, а не только для тестирования изолированного кода. В этом и состоит суть различия между модульным и интеграционным тестированием. В Python очень важно писать оба типа тестов по той же причине: вы не сможете гарантировать фактическую работоспособность модулей, не получив доказательств этого.

Примечание

В зависимости от проекта, удобно определять тесты, управляемые данными, или объединять тесты в различные наборы для проверки родственных видов функциональности. Особенно полезными для этих целей могут быть отчеты о покрытии кода и другие продвинутые варианты использования, а также пакеты с открытым исходным кодом `nose` (`http://nose.readthedocs.org/`) и `pytest` (`http://pytest.org/`).

Что следует запомнить

- Уверенность в работоспособности программ на Python может дать только написание и выполнение тестов.
- Встроенный модуль `unittest` предоставляет большинство средств, которые нужны вам для написания эффективных тестов.
- Вы можете определять тесты, создавая подклассы класса `TestCase` и определяя по одному методу для каждого варианта поведения, который вы хотели бы тестировать. Имена тестовых методов подклассов `TestCase` должны начинаться со слова `test`.

- Важно писать как модульные тесты (для изолированной функциональности), так и интеграционные тесты (для взаимодействующих модулей).

Рекомендация 57. Используйте интерактивную отладку с помощью пакета pdb

Избежать появления ошибок в коде в процессе разработки программ еще никому не удавалось. Функция `print()` может помочь вам в нахождении источников многих проблем в коде (см. раздел "Рекомендация 55. Используйте строки `repr` для вывода отладочной информации"). Другим замечательным способом изоляции проблем является написание тестов (см. предыдущую рекомендацию).

Однако этих инструментов недостаточно для выявления всех глубинных проблем. Когда вам потребуется нечто более мощное, обратитесь к встроенному *интерактивному отладчику* Python. Этот отладчик позволяет просматривать состояние программы, выводить значения локальных переменных и выполнять программы по одной инструкции за один раз.

В большинстве других языков программирования отладчики используют для того, чтобы указать, на какой строке кода следует приостановить, а затем продолжить выполнение программы. В отличие от этого, простейший способ использования отладчика Python — это изменение программы для непосредственной инициации отладчика перед участком кода, который вы считаете проблемным. Между выполнением программы на Python под управлением отладчика и обычным выполнением нет никакой разницы.

Все, что требуется сделать для того, чтобы инициировать отладчик, — это импортировать встроенный модуль `pdb` и запустить его функцию `set_trace()`. Вы часто можете увидеть, как это делается в одной строке, которую программисту легко "закомментировать" с помощью единственного символа #.

```
def complex_func(a, b, c):
    # ...
    import pdb; pdb.set_trace()
```

Как только встретится эта инструкция, выполнение программы приостановится. Терминал, в котором вы запустили эту программу, перейдет в интерактивный режим оболочки Python.

```
> complex_func()
(Pdb)
```

В ответ на приглашение (Pdb) можно вводить имена локальных переменных для просмотра их значений. Вы также можете вывести список всех локальных переменных, вызвав встроенную функцию `locals()`. Допускается импортировать модули, просматривать глобальное состояние, конструировать новые объекты, выполнять встроенную функцию `help()` и даже изменять части программы, т.е. делать все то, что может помочь вам в отладке программы. Кроме того, есть три команды, упрощающие изучение состояния выполняющейся программы.

- `bt`. Выводит текущий стек вызовов. Это позволяет определить, какая строка кода выполняется в данный момент и где вы находитесь в стеке вызовов.
- `up`. Перемещает область видимости вверх по стеку вызовов к более раннему вызову. Это позволяет просматривать локальные переменные на более высоких уровнях стека вызовов.
- `down`. Перемещает область видимости вниз по стеку вызовов на один уровень.

Закончив изучение текущего состояния, можно использовать команды отладчика для возобновления выполнения программы под вашим точным контролем.

- `step`. Выполняет программу до следующей выполняемой строки, а затем возвращает управление отладчику. Если следующая строка включает вызов функции, то отладчик приостанавливает выполнение в вызванной функции.
- `next`. Выполняет программу до следующей строки в текущей функции, а затем возвращает управление отладчику. Если следующая строка включает вызов функции, то отладчик не останавливается, пока не будет выполнен возврат из этой функции.
- `return`. Выполняет программу, пока не будет выполнен возврат из текущей функции, после чего управление возвращается отладчику.
- `continue`. Продолжает выполнение программы до следующей точки останова (или вызова функции `set_trace()`).

Что следует запомнить

- Вы можете инициировать интерактивный отладчик Python в интересующей вас точке непосредственно в программе с помощью инструкций `import pdb; pdb.set_trace()`.
- Приглашение отладчика Python предлагает вам полноценную командную оболочку, которая позволяет просматривать и изменять состояние выполняющейся программы.

- Команды оболочки pdb обеспечивают точный контроль над выполнением программы, позволяя вам переключаться между просмотром состояния программы и ее выполнением.

Рекомендация 58. Сначала — профилирование, затем — оптимизация

Динамическая природа Python может преподносить сюрпризы в поведении программ на этапе их выполнения. Операции, которые, как вы считали, должны выполняться медленно, на самом деле выполняются очень быстро (манипуляции со строками, генераторы). Средства языка, которые вы могли считать быстрыми, в действительности работают очень медленно (доступ к атрибутам, вызовы функций). Истинные причины медлительности программ в Python могут быть далеко не очевидными.

Лучше всего игнорировать то, что подсказывает вам интуиция, и непосредственно измерять производительность программы, прежде чем оптимизировать ее. Python предоставляет встроенный *профилировщик*, с помощью которого вы сможете определить, как долго выполняются отдельные части программы. Это позволяет вам сфокусировать свои усилия по оптимизации кода на самых крупных источниках проблем и игнорировать те части программы, которые не влияют на производительность.

Предположим, вы хотите выяснить причину медленной работы алгоритма, который используется в вашей программе. Ниже мы определим функцию, которая сортирует список элементов данных методом сортировки вставками.

```
def insertion_sort(data):
    result = []
    for value in data:
        insert_value(result, value)
    return result
```

В основе механизма сортировки вставками лежит функция, которая находит точку вставки для каждого элемента данных. Ниже мы определим крайне неэффективную версию функции `insert_value()`, использующую линейный просмотр элементов входного массива.

```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
            return
    array.append(value)
```

С целью профилирования `insertion_sort()` и `insert_value()` мы создадим набор данных в виде случайных чисел и определим функцию `test()` для передачи профилировщику.

```
from random import randint

max_size = 10**4
data = [randint(0, max_size) for _ in range(max_size)]
test = lambda: insertion_sort(data)
```

Python предоставляет два встроенных профилировщика, один из которых — это модуль Python (`profile`), а второй — модуль C-расширения (`cProfile`). Встроенный модуль `cProfile` лучше из-за его минимального влияния на производительность программы в процессе профилирования. В то же время с профилировщиком `profile` связаны слишком большие накладные расходы, которые искажают результаты.

Примечание

В процессе профилирования программы следите за тем, чтобы измерения относились именно к вашему коду, а не к внешним системам. Особое внимание обращайте на функции, которые получают доступ к сети или ресурсам на диске. Их влияние на длительность выполнения программы может быть существенным из-за медленной работы соответствующих базовых систем. Если ваша программа использует кеширование, маскирующее задержки, возникающие при загрузке внешних ресурсов наподобие указанных, вам дополнительно следует убедиться в достаточном наполнении кеша, прежде чем приступать к профилированию.

Ниже мы создадим экземпляр объекта `Profile` из модуля `cProfile` и выполним с его помощью тестовую функцию, используя метод `runcall()`.

```
profiler = Profile()
profiler.runcall(test)
```

Как только функция `test()` завершит выполнение, мы можем извлечь статистические данные о полученных результатах, используя встроенный модуль `pstats` и его класс `Stats`. Различные методы объекта `Stats` позволяют так настраивать отбор и сортировку профилирующей информации, чтобы отображались лишь те данные, которые вас интересуют.

```
stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
```

Вывод представляет собой таблицу данных, организованных с разбивкой по функциям. Отбор данных осуществляется лишь в то время,

когда активен профилировщик, в процессе выполнения приведенного выше метода `runcall()`.

```
>>>
       20003 function calls in 1.812 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
        1    0.000    0.000    1.812    1.812  main.py:34(<lambda>)
        1    0.003    0.003    1.812    1.812  main.py:10(insertion_sort)
    10000    1.797    0.000    1.810    0.000  main.py:20(insert_value)
     9992    0.013    0.000    0.013    0.000  {method 'insert' of 'list'
↳ objects}
        8    0.000    0.000    0.000    0.000  method 'append' of 'list'
↳ objects}
        1    0.000    0.000    0.003    0.000  {method 'disable' of
↳ '_lsprof.Profiler' objects}
```

Смысл значений, приведенных в таблице статистики профилировщика, описан ниже.

- `ncalls`. Количество вызовов функции за все время профилирования.
- `tottime`. Количество секунд, в течение которых выполнялась данная функции, исключая время выполнения вызываемых ею функций.
- `tottime percall`. Среднее количество секунд, в течение которых выполнялся один вызов данной функции, исключая время выполнения вызываемых ею функций. Это значение получается делением `tottime` на `ncalls`.
- `cumtime`. Суммарное количество секунд, в течение которых выполнялась данная функции, включая время выполнения вызываемых ею функций.
- `cumtime percall`. Среднее количество секунд, в течение которых выполнялся один вызов данной функции, включая время выполнения вызываемых ею функций. Это значение получается делением `cumtime` на `ncalls`.

Проанализировав статистические данные, приведенные в таблице профилирования, можно сделать вывод, что наиболее интенсивному использованию CPU, о чем можно судить по величине суммарного времени, в нашем тесте соответствует функция `insert_value()`. Ниже мы переопределим эту функцию для использования встроенного модуля `bisect` (см. раздел "Рекомендация 46. Используйте встроенные алгоритмы и структуры данных").

```
from bisect import bisect_left

def insert_value(array, value):
    i = bisect_left(array, value)
    array.insert(i, value)
```

Мы можем вновь выполнить профилировщик и сгенерировать новую таблицу статистических данных профилировщика. Новая функция работает значительно быстрее, причем суммарное время выполнения оказалось примерно в 100 раз меньше, чем для функции `insert_value()`.

```
>>>
       30003 function calls in 0.028 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
        1    0.000    0.000    0.028    0.028  main.py:34(<lambda>)
        1    0.002    0.002    0.028    0.028  main.py:10(insertion_sort)
    10000    0.005    0.000    0.026    0.000  main.py:112(insert_value)
    10000    0.014    0.000    0.014    0.000  {method 'insert' of 'list'
↳ objects}
    10000    0.007    0.007    0.000    0.000  {built-in method
↳ bisect_left}
        1    0.000    0.000    0.003    0.000  {method 'disable' of
↳ '_lsprof.Profiler' objects}
```

Иногда, когда вы профилируете всю программу, может обнаружиться, что за большую часть времени выполнения ответственна общая служебная функция. Вывод профилировщика, предусмотренный по умолчанию, затрудняет выявление таких ситуаций, поскольку он не отображает, как служебная функция вызывается различными частями вашей программы.

Например, приведенная ниже функция `my_utility()` повторно вызывается двумя различными функциями в программе.

```
def my_utility(a, b):
    # ...

def first_func():
    for _ in range(1000):
        my_utility(4, 5)

def second_func():
    for _ in range(10):
        my_utility(1, 3)

def my_program():
```

```
    for _ in range(20):
        first_func()
        second_func()
```

Результаты профилирования этого кода, полученные с использованием заданных по умолчанию установок для вывода на печать с помощью функции `print_stats()`, могут сбивать с толку.

```
>>>
       20242 function calls in 0.208 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
        1    0.000    0.000    0.208    0.208  main.py:176(my_program)
       20    0.005    0.000    0.206    0.010  main.py:168(first_func)
    20200    0.203    0.000    0.203    0.000  main.py:161(my_utility)
       20    0.000    0.000    0.002    0.000  main.py:172(second_func)
        1    0.000    0.000    0.000    0.000  {method 'disable' of
↳ '_lsprof.Profiler' objects}
```

Совершенно очевидно, что большая часть времени выполнения выпадает на функцию `my_utility()`, однако вовсе не очевидно, почему эта функция вызывается так часто. Если вы просмотрите программный код, то обнаружите несколько мест, где вызывается данная функция, но это вовсе не прояснит ситуацию.

Чтобы можно было справляться с проблемами подобного рода, профилировщик Python предоставляет возможность оценить, какой вызывающий код внес вклад в данные профилирования каждой функции.

```
stats.print_callers()
```

Ниже приведена таблица статистических данных профилировщика, в которой слева отображаются вызываемые функции, а справа — функции, ответственные за вызов. Как следует из данных таблицы, функция `my_utility()` в основном используется функцией `first_func()`.

```
>>>
   Ordered by: cumulative time

Function                 was called by...
                             ncalls  tottime  cumtime
main.py:176(my_program)  <-
main.py:168(first_func)  <-     20    0.005    0.206  main.py:176(my_program)
main.py:161(my_utility)  <- 20000    0.202    0.202  main.py:168(first_func)
                              200    0.002    0.002  main.py:172(second_func)
main.py:172(second_func) <-     20    0.000    0.002  main.py:176(my_program)
```

Что следует запомнить

- Очень важно сначала профилировать программы и лишь затем оптимизировать, поскольку причины медленной работы программ часто далеко не очевидны.
- Используйте модуль `cProfile` вместо модуля `profile`, поскольку он предоставляет более полные результаты профилирования.
- Метод `runcall()` объекта `Profile` предоставляет все, что вам нужно для профилирования дерева вызова функций в изоляции.
- Объект `Stats` позволяет выбрать и вывести на печать подмножество данных профилирования, необходимых для того, чтобы выяснить факторы, влияющие на производительность вашей программы.

Рекомендация 59. Используйте модуль `tracemalloc` для контроля памяти и предотвращения ее утечки

В стандартной реализации Python, CPython, система управления памятью использует счетчики ссылок на объекты. Это гарантирует, что объект, ссылки на который в программе уже не используются, будет удален из памяти. Кроме того, в CPython предусмотрен встроенный детектор циклических ссылок, гарантирующий, что объекты, ссылающиеся сами на себя, в конечном счете также будут удалены из памяти сборщиком мусора.

Теоретически это означает, что большинству программистов на Python не стоит беспокоиться о распределении памяти в своих программах. Об этом автоматически позаботятся языковые средства и среда выполнения CPython.

Однако на практике бывает так, что программа в конечном счете перерасходует память из-за задержанных ссылок. Определить, действительно ли программа расходует память по назначению или имеет место утечка памяти, довольно затруднительно.

Суть первого способа контроля расходования памяти, который мы рассмотрим, состоит в том, чтобы запросить список всех объектов, известных в данный момент системе сборки мусора, с помощью встроенного модуля `gc`. Несмотря на то что этот инструмент довольно грубый, он позволяет быстро оценить эффективность использования памяти вашей программой.

Ниже приведен пример выполнения программы, расходующей память из-за удерживаемых ссылок. Программа выводит информацию

относительно количества объектов, созданных во время выполнения, а также информацию о распределении памяти для небольшой выборки объектов .

```
# using_gc.py
import gc
found_objects = gc.get_objects()
print('%d объектов до' % len(found_objects))

import waste_memory
x = waste_memory.run()
found_objects = gc.get_objects()
print('%d объектов после' % len(found_objects))
for obj in found_objects[:3]:
    print(repr(obj)[:100])

>>>
4756 объектов до
14873 объектов после
<waste_memory.MyObject object at 0x1063f6940>
<waste_memory.MyObject object at 0x1063f6978>
<waste_memory.MyObject object at 0x1063f69b0>
```

Недостатком метода `gc.get_objects()` является то, что он ничего не говорит о том, как именно были размещены объекты в памяти. В сложных программах конкретный тип объектов может быть размещен в памяти многими способами.

Однако главное не столько в общем количестве объектов, сколько в выяснении того, какой код отвечает за распределение памяти для объектов, обусловливающих утечку.

В Python 3.4 включен новый встроенный модуль `tracemalloc`, с помощью которого можно решить эту задачу. Модуль `tracemalloc` позволяет трассировать память с целью определения особенностей распределения памяти для объектов. Ниже мы выведем информацию о трех основных потребителях памяти в программе, используя `tracemalloc`.

```
# top_n.py
import tracemalloc
tracemalloc.start(10) # Сохранять до 10 фреймов стека

time1 = tracemalloc.take_snapshot()
import waste_memory
x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'lineno')
```

```
for stat in stats[:3]:
    print(stat)
>>>
waste_memory.py:6: size=2235 KiB (+2235 KiB),
↳count=29981 (+29981), average=76 B
waste_memory.py:7: size=869 KiB (+869 KiB),
↳count=10000 (+10000), average=89 B
waste_memory.py:12: size=547 KiB (+547 KiB),
↳count=10000 (+10000), average=56 B
```

Здесь сразу можно увидеть, какие объекты играют главную роль в потреблении памяти и в каких строках располагается исходный код, выделяющий память для них.

Кроме того, модуль `tracemalloc` обеспечивает вывод полной информации о трассировке стека для каждого блока (вплоть до количества фреймов, переданных методу `start()`).

Выведем данные трассировки стека для основного потребителя памяти в программе.

```
# with_trace.py
# ...
stats = time2.compare_to(time1, 'traceback')
top = stats[0]
print('\n'.join(top.traceback.format()))

>>>
File "waste_memory.py", line 6
  self.x = os.urandom(100)
File "waste_memory.py", line 12
  obj = MyObject()
File "waste_memory.py", line 19
  deep_values.append(get_data())
File "with_trace.py", line 10
    x = waste_memory.run()
```

Результаты трассировки стека, подобные этим, представляют наибольшую ценность для выяснения того, какие конкретно функции общего назначения ответственны за потребление памяти в программе.

К сожалению, в Python 2 встроенный модуль `tracemalloc` не предоставляется. Существуют пакеты с открытым исходным кодом, обеспечивающие трассировку памяти в Python 2 (такие, как `heapy`), однако они не воспроизводят в полной мере функциональность, предоставляемую встроенным модулем `tracemalloc`.

Что следует запомнить

- Контроль использования памяти и предотвращения ее утечки в программах на Python может быть затруднен.
- Модуль gc может помочь в выяснении того, какие объекты существуют в программе, но он не предоставляет информацию относительно распределения памяти для этих объектов.
- Встроенный модуль `tracemalloc` предоставляет мощные инструменты, позволяющие определять расход памяти.
- Встроенный модель `tracemalloc` доступен лишь в Python 3.4 и более поздних версиях.

Предметный указатель

Д

И

К

Л

М

О

П

Р

С

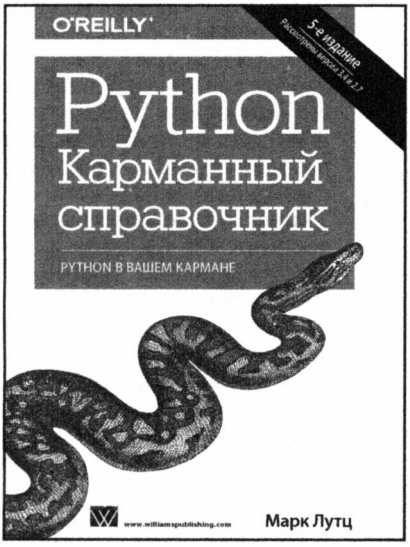
Т

У

Ф

PYTHON
КАРМАННЫЙ СПРАВОЧНИК
ПЯТОЕ ИЗДАНИЕ

Марк Лутц



www.williamspublishing.com

Этот краткий справочник по Python составлен с учетом версий 3.4 и 2.7 и очень удобен для наведения быстрых справок при разработке программ на Python. В лаконичной форме здесь представлены все необходимые сведения о типах данных и операторах Python, специальных методах перегрузки операторов, встроенных функциях и исключениях, наиболее употребительных стандартных библиотечных модулях и других примечательных языковых средствах Python, в том числе и для объектно-ориентированного программирования. Справочник рассчитан на широкий круг читателей, интересующихся программированием на Python.

ISBN 978-5-8459-1912-0 **в продаже**

PYTHON
СОЗДАНИЕ ПРИЛОЖЕНИЙ
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТРЕТЬЕ ИЗДАНИЕ

Уэсли Чан

www.williamspublishing.com

Книга содержит всю необходимую информацию для создания реальных приложений на языке Python. В ней описаны профессиональные приемы программирования на языке Python, методы создания клиентов и серверов с помощью протоколов TCP, UDP и XML-RPC, работа с высокоуровневыми библиотеками SocketServer и Twisted, изложены основы разработки GUI-приложений с помощью библиотеки Tkinter, продемонстрированы расширения на языке C/C++ и использование многопоточности, описана работа с реляционными базами данных, ORM и MongoDB, изложены основы веб-программирования, регулярных выражений, программирования COM-клиентов веб-разработки с помощью каркаса Django и облачных вычислений на платформе Google App Engine, а также программирование на языке Java в среде Jython и способы установления соединений с социальными сетями Twitter и Google+. Книга представляет собой ценный источник знаний по языку Python и предназначена для программистов всех уровней.

ISBN 978-5-8459-1793-5

в продаже

Программирование/Python

"Каждая рекомендация в книге Секреты Python *— это самостоятельный урок с наглядными примерами исходного кода, что позволяет читать главы в любом порядке. Я буду рекомендовать эту книгу студентам в качестве компактного сборника ценных наставлений по широкому кругу вопросов для программистов среднего и профессионального уровня, работающих с языком Python".*

Брэндон Роудс, инженер компании Dropbox, председатель конференции PyCon 2016-2017

Секреты PYTHON

Язык Python завоевал популярность благодаря тому, что позволяет новичкам почти сразу же браться за написание кода. Однако достигнуть цельного понимания уникальных возможностей Python чрезвычайно трудно, особенно если учесть, что на этом пути вас подстерегает множество скрытых ловушек.

Книга приобщит вас к стилю программирования, выдержанному в истинном "духе Python", и поможет научиться писать исключительно надежный и высокопроизводительный код. Используя сжатый стиль изложения, пионером которого был Скотт Мейерс, автор приводит 59 описаний лучших методик программирования, дает советы и показывает кратчайшие пути решения различных задач программирования на Python, дополняя их реалистичными примерами кода.

Опираясь на свой многолетний опыт создания инфраструктурных проектов для компании Google, автор раскрывает секреты малоизвестных аспектов и идиом Python, радикально влияющих на поведение и производительность кода. Вы ознакомитесь с наилучшими способами решения ключевых задач, что облегчит понимание, сопровождение и усовершенствование вашего кода.

Основные темы книги:

- Рекомендации по основным аспектам разработки ПО с использованием версий Python 3.x и 2.x, дополненные подробными описаниями и примерами.
- Лучшие методики написания функций, снижающие вероятность появления ошибок в коде.
- Точное описание вариантов поведения с помощью классов и объектов.
- Рекомендации относительно того, как избежать скрытых ошибок с помощью метаклассов и динамических атрибутов.
- Эффективные подходы к решению проблем, связанных с одновременным и параллельным выполнением множества операций.
- Усовершенствованные приемы работы со встроенными модулями Python.
- Инструментальные средства и лучшие методики коллективной разработки.
- Решения по отладке, тестированию и оптимизации кода.

Бретт Слаткин — ведущий инженер-разработчик компании Google и соучредитель компании Google Consumer Surveys. До этого разрабатывал инфраструктуру Python для Google App Engine, адаптировал Python для управления огромным серверным парком Google и использовал Python в качестве инструмента реализации системы Google для протокола PubSubHubbub. Получил степень бакалавра в области компьютерной инженерии в Колумбийском университете в Нью-Йорке.

Изображение на обложке:

ВИЛЬЯМС

www.williamsp

Addison-Wesley

ISBN: 978-5-8459-2078-2
16037
9 785845 920782

PEARSON