EXPERT INSIGHT

python™

Python лучшие практики и инструменты

Четвертое издание

Михал Яворски
Тарек Зиаде

Packt›

Python. Лучшие практики и инструменты

Четвертое издание

Михал Яворски

Тарек Зиаде

ПИТЕР®

Санкт-Петербург • Москва • Минск

2024

ББК 32.973.2-018.1
УДК 004.43
Я22

Яворски Михал, Зиаде Тарек

Я22 Python. Лучшие практики и инструменты. 4-е изд. — СПб.: Питер, 2024. — 592 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-2064-2

Python — простой, но мощный язык, поэтому он используется в самых разных областях. Написать код на Python легко, но сделать его удобочитаемым и пригодным для повторного использования и сопровождения может оказаться проблемой. Четвертое издание этой книги дополнено лучшими практиками, полезными инструментами и стандартами, которые применяют профессиональные разработчики, что поможет вам не только преодолеть эти затруднения, но и освоить новейшие возможности и расширенные концепции языка.

Книга начинается с легкой разминки, которая познакомит вас с последними улучшениями Python, элементами синтаксиса и полезными инструментами, делающими разработку эффективнее. Кроме того, начальные главы помогут программистам с опытом работы на других языках успешно влиться в экосистему Python.

Следующие главы посвящены распространенным паттернам проектирования и методологиям программирования — таким как событийно-ориентированное программирование, параллелизм и метапрограммирование. Также вы разберете сложные примеры кода и будете решать содержательные задачи, связывая Python с C и C++ и создавая расширения, сочетающие сильные стороны разных языков. В заключительных главах рассматривается полный жизненный цикл приложения после ввода в эксплуатацию.

К концу книги вы освоите разработку эффективного и простого в сопровождении кода на Python.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1801071109 англ.

ISBN 978-5-4461-2064-2

Краткое содержание

Оглавление

Об авторах

Михал Яворски (Michał Jaworski) более 10 лет профессионально разрабатывает программное обеспечение на разных языках. Бóльшую часть этого времени Михал занимался высокопроизводительными распределенными бэкенд-сервисами для веб-приложений. Он работал на разных должностях в нескольких компаниях: от рядового разработчика до ведущего специалиста по архитектуре. Python всегда был и остается его любимым языком.

Хочу поблагодарить свою жену за неустанную поддержку. Оливия, ты единственная с самого начала знала, что я обманывал себя, когда говорил, что это легкий проект, который не отнимет много моего (нашего) времени. Не знаю почему, но ты все равно вдохновила меня на то, чтобы им заняться.

Тарек Зиаде (Tarek Ziadé) — программист из Бургундии (Франция). Он трудится в Elastic и создает инструменты для разработчиков. До Elastic 10 лет проработал в Mozilla и основал AFPy — французскую группу пользователей Python. Тарек также написал ряд статей о Python для разных журналов и несколько книг на французском и английском языках.

Спасибо Фрее, Суки, Мило и Амине за ту поддержку, которую они оказывают во всех моих проектах по написанию книг.

О научном редакторе

Тал Эйнат (Tal Einat) занимается разработкой программного обеспечения более 20 лет, и Python всегда был его основным языком. Тал входит в группу главных разработчиков языка Python с 2010 года. Он получил диплом бакалавра по математике и физике в Тель-Авивском университете. Тал увлекается пешим туризмом, компьютерными играми и философской научной фантастикой, а также любит проводить время с семьей.

В последние 8 лет Тал занимался образовательными технологиями: сначала в Compedia, где создал группу разработки обучающих приложений с использованием виртуальной и дополненной реальности, а потом в стартапе FullProof, где он является одним из основателей.

Сейчас Тал работает в стартапе Rhino Health, который разрабатывает и внедряет модели искусственного интеллекта в области медицины. Эти модели используют данные пациентов со всего мира, обеспечивая их конфиденциальность.

Посвящаю свою часть работы над книгой моему деду Якову «Янеку» Фридману, который недавно скончался. Ты продолжаешь жить в нашей памяти, придаешь нам силы и даришь хорошее настроение. Также хочу поблагодарить свою жену, детей, братьев, родителей и многих других родственников за поддержку и за радость, которой они наполняют мою жизнь.

Предисловие

Python — это круто!

Начиная с первой версии в конце 1980-х годов и до текущей версии 3.9[1], Python развивается по одной и той же концепции: как мультипарадигменный язык программирования с особым вниманием к удобочитаемости и эффективности кода.

Поначалу Python рассматривался как еще один язык сценариев. Многие не верили, что на нем можно разрабатывать большие и сложные системы. Но за прошедшие годы, в том числе благодаря компаниям-первопроходцам, стало очевидно, что с помощью Python можно создавать практически любые программы.

Писать код на Python — просто. Сложнее добиться, чтобы код было удобно читать, повторно использовать и сопровождать. Этого можно достичь только мастерством и хорошей техникой программирования, которые формируются по мере того, как вы учитесь и накапливаете опыт.

В этой книге мы постарались отразить многолетний профессиональный опыт создания разнообразных приложений на Python — от небольших системных сценариев, которые пишутся за пару часов, до крупных приложений, над которыми десятки разработчиков трудятся по несколько лет.

Книга делится на три части:

1. **Владенне ннструментамн**: в главах 1–4 представлены основные элементы инструментария программистов на Python: от средств новышения производительности и современных сред разработки до новейших элементов синтаксиса, которые появились в последних релизах Python. Также эта часть станет удобной отправной точкой для программистов с опытом работы на других языках, которые только начинают изучать расширенные возможности Python.

[1] На момент подготовки русского издания книги последняя версия — Python 3.11.2. — *Примеч. ред.*

2. **Созданне приложений на Python**: главы 5–9 посвящены паттернам, парадигмам программирования и приемам метапрограммирования. Мы попробуем разработать несколько небольших, но полезных программ и будем часто углубляться в особенности архитектуры приложений. Также мы заглянем за пределы Python и посмотрим, как интегрировать код, написанный на Python, с кодом на других языках программирования.
3. **Сопровожденне приложений на Python**: в главах 10–13 рассматривается жизненный цикл приложения после запуска в эксплуатацию. Здесь представлены инструменты и приемы, которые упрощают сопровождение, и показаны подходы к решению типичных проблем при упаковке, развертывании, мониторинге и оптимизации производительности.

Для кого написана эта книга

Книга предназначена для опытных программистов, которые хотят больше узнать о продвинутых концепциях и новейших возможностях Python.

Она написана для тех, кому интересно оттачивать мастерство владения Python, и прежде всего для профессиональных разработчиков, которые зарабатывают на жизнь программированием на Python. Из этих соображений книга ориентирована прежде всего на инструменты и нриемы, необходимые для создания производительного, надежного и простого в сопровождении кода на Python.

Однако это не значит, что для программистов-любителей здесь нет ничего интересного. Книга прекрасно подойдет всем, кто хочет изучить продвинутые концепции языка Python. Чтобы освоить материал книги, достаточно владеть базовыми навыками Python, хотя менее опытным программистам, возможно, понадобится приложить дополнительные усилия. Книга также послужит неплохим введением в новейшие возможности Python для тех, кто немного отстал и пока еще использует старые версии языка.

Обзор содержания книги

В главе 1 «*Python сегодняшнего дня*» рассказано, что сейчас происходит в языке Python и в его профессиональном сообществе. Вы увидите, что Python постоянно изменяется, и поймете почему. Вы узнаете, что делать со старым кодом на Python 2 и как оставаться в курсе всего, что происходит в сообществе Python.

В главе 2 «*Современные среды разработки для Python*» описано, как современные программисты налаживают воспроизводимые и целостные среды разработки на Python. Вы узнаете, чем отличается изоляция на уровне приложения от изоляции на уровне системы. Основное внимание уделено двум популярным инструментам

изоляции — средам типа virtualenv и контейнерам Docker, но также рассматриваются и другие альтернативы. В конце главы представлены популярные средства повышения производительности, которые особенно полезны при разработке.

В главе 3 «*Новые возможности Python*» продемонстрированы последние нововведения в языке. Мы рассмотрим важнейшие изменения в синтаксисе Python за последние четыре выпуска. Также представлены интересные новшества, запланированные на следующий основной релиз — Python 3.10.

Глава 4 «*Python в сравнении с другими языками*» показывает важные сходства и отличия Python от других языков. Вы узнаете, что такое идиомы программирования и как распознать их в коде. Мы глубже рассмотрим ключевые элементы объектно-ориентированной модели Python и ее отличия от прочих объектно-ориентированных языков, а также обсудим другие популярные средства языка, такие как дескрипторы, декораторы и классы данных. Эта глава позволит программистам с опытом работы на других языках успешно влиться в экосистему Python.

В главе 5 «*Интерфейсы, паттерны и модульность*» обсуждаются элементы Python, которые позволяют реализовывать разные паттерны проектирования, пригодные для повторного использования. Основное внимание уделяется концепции интерфейсов классов и их реализации в Python. Речь идет также об инверсии управления и внедрении зависимостей — двух приемах программирования, которые чрезвычайно полезны, но порой незаслуженно упускаются из внимания.

Глава 6 «*Конкурентное выполнение*» объясняет, как реализовать конкурентность в Python с помощью разных подходов и библиотек. В главе представлены три основные модели конкурентного выполнения: многоточность, многопроцессорность и асинхронное программирование. В этой главе вы узнаете ключевые различия этих моделей и научитесь эффективно ими пользоваться.

Глава 7 «*Событийно-ориентированное программирование*» рассказывает о том, что такое событийно-ориентированное программирование и как оно связано с асинхронным программированием и различными моделями конкурентного выполнения. Также представлено несколько подходов к событийному программированию и некоторые полезные библиотеки.

В главе 8 «*Элементы метапрограммирования*» приведена сводка стандартных приемов метапрограммирования, доступных для разработчиков на Python. Вы узнаете о таких стандартных средствах метапрограммирования, как декораторы, метаклассы и паттерны генерирования кода.

Глава 9 «*Интеграция Python с C и C++*» объясняет, как интегрировать код, написанный на других языках, с приложениями на Python. Вы узнаете, когда расширения на C бывают полезны и как их создавать.

Глава 10 «*Автоматизация тестирования и контроля качества*» рассказывает, как автоматизировать тестирование и контроль качества. В ней вы узнаете о но-

пулярном фреймворке тестирования Pytest и о многих полезных приемах тестирования. Также представлены инструменты, с помощью которых можно получать метрики контроля качества и гармонизировать стиль написания кода в полностью автоматическом режиме.

В главе 11 «*Упаковка и распространение кода Python*» описано, как устроены пакеты Python и как создавать пакеты для распространения с открытым исходным кодом в PyPI (Python Package Index). Мы также рассмотрим упаковку приложений для веб-разработки и создание автономных исполняемых файлов Python для настольных приложений.

В главе 12 «*Наблюдение за поведением и быстродействием приложений*» обсуждается мониторинг действующих приложений. Вы узнаете о системах журналирования Python, научитесь отслеживать метрики приложений и выполнять распределенную трассировку транзакций. Вы также ознакомитесь с тем, как простые приемы мониторинга масштабируются для крупных распределенных систем.

В главе 13 «*Оптимизация кода*» представлены базовые правила оптимизации, которые должен знать каждый разработчик. Вы научитесь выявлять узкие места производительности приложений и пользоваться популярными средствами профилирования. Кроме того, вы узнаете, какие распространенные методы и стратегии оптимизации стоит применять к обнаруженным узким местам.

Как работать с книгой наиболее эффективно

Эта книга подходит для программистов, работающих в любой операционной системе, где доступен Python 3.

Книга не рассчитана на начинающих, так что мы предполагаем, что в вашей среде уже установлен Python или вы по крайней мере знаете, как его установить. При этом в книге учитывается тот факт, что не каждый читатель в курсе всех новейших возможностей Python или официально рекомендуемых инструментов. Поэтому в главе 2 «Современные среды разработки для Python» приведен обзор рекомендуемых приемов и инструментов (таких, как виртуальные среды и `pip`) для налаживания сред разработки.

Загрузка файлов с кодом примеров

Код примеров из этой книги также доступен на GitHub по адресу **https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition**. Если код обновляется, изменения отражаются в существующем репозитории GitHub.

В нашем обширном каталоге книг и видеокурсов по адресу https://github.com/PacktPublishing/ есть и другие сборники кода, которые заслуживают вашего внимания.

Условные обозначения

В этой книге используются следующие условные обозначения.

Элементы кода в тексте. Такой шрифт применяется внутри абзацев для обозначения таких элементов, как команды, переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова. Например: «Любая попытка запустить код с такими проблемами заставит интерпретатор завершить работу, выдав исключение `Syntax Error exception`».

Листинги выделяются следующим образом:

```
print("hello world")
```

Ввод и вывод командной строки записываются так:

```
$ python3 script.py
```

Некоторые примеры кода представляют собой ввод в оболочке. Вы узнаете их по символам приглашения командной строки:

- `>>>` для интерактивной оболочки Python;
- `$` для Bash (macOS и Linux);
- `>` для CMD или PowerShell (Windows).

Некоторые фрагменты кода или примеры ввода командной строки требуют подстановки ваших собственных данных. Для указания на это используются угловые скобки:

```
$ python3 script.py
```

Так помечаются предупреждения и важные примечания.

А так — советы и секреты.

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Python сегодняшнего дня

Python — поразительный язык. Он поражает прежде всего тем, что в течение многих лет остается актуальным и продолжает развиваться.

С давних пор одним из важнейших достоинств Python была операционная совместимость, благодаря которой было неважно, что за операционная система у вас или у ваших клиентов. Если для этой ОС существовал интерпретатор Python, то ваши программы на Python в ней работали. И что еще важнее, программы работали всегда одинаково в разных ОС. Однако в наше время это качество уже стало обычным и для других современных языков программирования, которые обеспечивают аналогичные возможности операционной совместимости. Кроме того, с развитием облачных служб, приложений на основе веб-технологий и надежных программных средств виртуализации уже не столь важно, чтобы язык программирования работал во многих ОС.

Похоже, в наши дни на первый план выходит продуктивность работы программистов. В постоянном стремлении к новшествам часто бывает важно сделать прототип, который можно опробовать в полевых условиях на реальных пользователях, а затем без проволочек развивать его до стадии полноценного продукта. Python позволяет программистам проходить этот путь очень быстро. Официальный каталог пакетов Python Package Index — огромная подборка программных библиотек и фреймворков, которые вы можете легко использовать в своих проектах. Благодаря этому каталогу вы тратите на работу гораздо меньше времени и усилий. Широчайшая доступность библиотек, созданных сообществом, в сочетании с четким и лаконичным синтаксисом, ориентированным на удобочитаемость кода, облегчает создание и сопровождение программных продуктов. Поэтому Python показывает выдающиеся результаты по части продуктивности работы программистов.

Python так долго остается востребованным потому, что он непрерывно развивался и продолжает развиваться. В этой главе мы обзорно рассмотрим текущее состояние дел в Python и разберемся, как оставаться в курсе последних изменений в экосистеме и сообществе Python.

Эта глава посвящена следующим темам:

- Где мы находимся и куда направляемся?
- Что делать с Python 2?
- Как оставаться в курсе происходящего?

Для начала вспомним историю развития Python и посмотрим, что происходит с ним сейчас.

Где мы находимся и куда направляемся

Python нельзя назвать молодым языком. Его история начинается в конце 1980-х, а первая официальная версия 1.0 была выпущена в 1994 году. Здесь можно было бы привести всю хронологию выпуска основных версий Python, но на самом деле важны лишь несколько дат:

- 16 октября 2000-го: официальный выпуск Python 2.0;
- 3 декабря 2008-го: официальный выпуск Python 3.0;
- 9 ноября 2011-го: объявление об отмене выпуска Python 2.8;
- 1 января 2020-го: официальное прекращение поддержки Python 2.

Таким образом, на момент написания книги версия Python 3 примерно вдвое младше, чем Python в целом. Кроме того, в этой версии эпоха активной разработки новых возможностей языка уже продолжается дольше, чем она происходила в Python 2.

Хотя Python 3 существует уже достаточно давно, его внедрение шло настолько медленно, что изначально запланированное прекращение поддержки Python 2 было отложено на 5 лет. В основном это объяснялось многочисленными проблемами обратной совместимости, из-за которых не всегда был возможен простой и прямолинейный переход. Кроме того, популярность Python в значительной мере обусловлена огромным количеством бесплатных библиотек. Следовательно, миграция на Python 3 дополнительно усложнялась, если ваша программа критически зависела от стороннего пакета, который еще не был совместим с Python 3.

К счастью, Python 2 наконец-то остался в прошлом, и сообщество программистов может вздохнуть с облегчением. Многие разработчики библиотек Python с от-

крытым кодом уже давно перестали обеспечивать совместимость с Python 2. Кроме того, официальное прекращение поддержки Python 2 стимулировало переход на Python 3 в корпоративных средах, где к подобным инициативам обычно относятся без энтузиазма. В основном это объясняется тем, что для Python 2 больше не планируется никаких патчей безопасности.

Что делать с Python 2

Итак, Python 2 больше не поддерживается разработчиками языка. Ввиду отсутствия патчей безопасности его следует считать уязвимым. Но можно ли назвать его мертвым?

Пожалуй, нет. Даже относительно популярные библиотеки с открытым кодом со временем могут стать неинтересными для своих создателей и соавторов. Также бывает, что появляются более совершенные альтернативы, так что разработка исходной библиотеки попросту теряет смысл. Такие библиотеки часто перестают сопровождаться, и никто не обновляет их ради совместимости с Python 3.

Одной из главных причин, по которым Python 3 долго не получал широкого распространения, были нерасторопные процессы выпуска многих дистрибутивов Linux. Специалисты по сопровождению дистрибутивов обычно не спешат принимать новые версии языков, особенно если эти версии нарушают обратную совместимость и требуют вносить исправления в другое ПО. Многие программисты на Python работают под Linux, и если новейшая версия языка недоступна в системном репозитории пакетов, это снижает вероятность того, что они проведут обновление самостоятельно.

Но существует область, где Python 2 определенно задержится еще на несколько лет: это корпоративные среды. Когда речь идет о денежных затратах, обычно трудно убедить стейкхолдеров, что ранее внедренные системы нужно пересмотреть просто ради того, чтобы не отставать от предстоящих изменений, — особенно если ПО работает.

Скорее всего, код на Python 2 больше не будет встречаться в ключевых компонентах активно разрабатываемых продуктов, но он все еще попадается в служебных сценариях, внутренних инструментах или службах, которые давно не развивались.

Если вам приходится сопровождать программный код, написанный на Python 2, лучше всего подумать о том, как поскорее перейти на Python 3. Обычно это не делается в одночасье, и часто приходится сначала убеждать других, прежде чем можно будет приступать к миграции.

Оказавшись в подобной ситуации, попробуйте такую стратегию:

1. Определите, что нужно переносить и почему. В зависимости от ситуации вы будете иметь дело с разными фрагментами кода, которые используются по-разному. Не все они заслуживают миграции. Если код на самом деле нигде не используется, нет смысла его обновлять.
2. Выясните, что мешает миграции. Иногда вы работаете с кодом, зависимости которого не удается легко перевести на новую версию. Это усложнит миграцию, и стоит выявить такие ситуации заранее, чтобы продумать эффективный план перехода.
3. Убедите стейкхолдеров. Если вы разрабатываете библиотеку с открытым кодом, вам придется убедить коллег-соавторов, которые будут помогать вам со сменой версии. Если вы работаете в компании по разработке ПО, то нридется убедить руководство, которое платит вам зарплату, что миграция важнее, чем, скажем, работа над новой функциональностью.

Как правило, труднее всего получить согласие стейкхолдеров, особенно если вы профессиопально пишете код и необходимо как-то втиснуть миграцию в график текущей разработки новой функциональности. Прежде чем выносить этот вопрос на обсуждепие, надо хорошо подготовиться, именно поэтому первые два шага так важны. Они позволяют оценить, какие ресурсы будут затрачены на переход, и сформулировать убедительное обоснование для этого проекта. По сути, чтобы кого-то убедить, лучше всего составить список преимуществ миграции. Ниже перечислены распространенные аргументы в ее пользу:

- **Можно нспользовать более новые н качественные библиотекн.** С тех пор как поддержка Python 2 официально прекращена, крайне маловероятно, что новые (и возможно, лучшие) библиотеки будут совместимы с этой версией.
- **Синжаются затраты на владенне кодом.** Если в других проектах и компонентах команда использует более повую версию Python, то будет выгоднее привести разные проекты к одной и той же версии, потому что при этом уменьшится общая сложность.
- **Упрощается сонровождеине.** Встроенная поддержка Python 2 постепенно исчезает из разных сред исполнения и операционных систем. Использование более новой версии Python сократит операционные затраты на развертывание нрограммного продукта.
- **Облегчается адаптация иовых сотрудинков.** Переход на единую версию Python облегчает адаптацию новых членов команды, потому что им будет проще с самого начала работать со всей кодовой базой.

Чтобы убедить стейкхолдеров, можно также разъяснить риски, которые возникают при отказе от миграции на новую версию Python:

- **Серьезная угроза безопасности.** После официального прекращения поддержки Python 2 выход штатных патчей безопасности не гарантирован. Правда, этот риск остается умозрительным, пока не обнаружатся конкретные уязвимости. Тем не менее если вы используете Python 2, то уже ограничиваете себе доступ к обновлениям сторонних библиотек и многих проектов с открытым кодом, которые прекратили поддержку Python 2 несколько лет назад.
- **Высокие затраты, связанные с безопасностью.** Хотя теоретически можно создать ветку проекта с открытым кодом и вносить исправления безопасности своими силами, постоянные затраты на это обычно намного больше, чем стоимость миграции на новую версию языка.
- **Проблемы с наймом персонала.** Этот риск проявляется независимо от опыта программистов. Молодые разработчики могут быть знакомы с Python 2 гораздо хуже, чем с Python 3. Им будет сложнее войти в курс дела, зато проще наделать примитивных ошибок с потенциально катастрофическими последствиями. С другой стороны, опытные программисты не всегда охотно соглашаются работать в среде, которая опирается на устаревшую технологию.

Чтобы конструктивно обсуждать тему миграции, вам понадобится продумать работоспособный тактический план, в котором сбалансированы перспективы выигрышей, риски и способность достичь целей команды в разумном темпе. Например, инвестиции в сокращение будущих затрат на сопровождение обычно окупаются только после продолжительного времени. Поэтому такие вложения тоже стоит распределять во времени. С другой стороны, известным и незакрытым уязвимостям безопасности всегда следует уделять первоочередное внимание.

Как оставаться в курсе происходящего

Технологии непрерывно развиваются. Люди постоянно ищут новые инструменты, которые позволят решать задачи быстрее, чем раньше. Каждые несколько месяцев невесть откуда возникает совершенно новый язык или в одном из общепризнанных языков появляется совершенно новый элемент синтаксиса. Это относится и к Python. Важнейшие из недавних нововведений в Python рассматриваются в главе 3 «Новые возможности Python».

Новые языки или языковые возможности стимулируют создание новых библиотек и фреймворков. Они, в свою очередь, открывают путь новым парадигмам программирования и паттернам проектирования. Успех таких паттернов или

парадигм в экосистеме одного языка часто способствует тому, что программисты внедряют их в других экосистемах. Так новые идеи переходят из языка в язык.

Это относится и к Python. В главе 4 «Python в сравнении с другими языками» мы увидим, как разные на первый взгляд языки программирования используют общие возможности и концепции.

Языки постоянно эволюционируют. Этот процесс неизбежен, и похоже, он только ускоряется. Начиная с Python 3.9, вышедшего 5 октября 2020-го, новые основные версии Python выпускаются ежегодно. Как правило, каждый выпуск порождает лавину новых библиотек и фреймворков, в которых обкатываются свежие нововведения. Для сообщества Python это хорошо, потому что таким образом постоянно стимулируются инновации. С другой стороны, при этом непросто успеть уследить за всем, что происходит в мире Python, хотя для каждого профессионала в высшей степени важно уметь оставаться на технологической передовой.

В ближайших разделах мы обсудим некоторые источники информации об актуальном положении дел в Python. Они помогут вам лучше адаптироваться к изменениям в языке и окружающем его сообществе, не отставать от последних передовых приемов и своевременно осваивать новые полезные инструменты.

Документы PEP

В сообществе Python сложился общепринятый подход к изменениям. Гипотетические идеи по развитию языка в основном обсуждаются в специализированном списке рассылки (python-ideas@python.org), но никакие серьезные изменения не вносятся без появления нового документа, который называется «предложением по расширению Python» (**PEP**, **Python Enhancement Proposal**).

На python-ideas@python.org и другие рассылки можно подписаться по адресу https://mail.python.org/mailman3/lists/.

PEP — формализованный документ, в котором подробно описано предложение по внесению изменений в Python. Он становится отправной точкой для обсуждения в сообществе. Некоторые темы этой главы подробно описаны в соответствующих документах PEP:

- PEP 373 — график выпусков Python 2.7;
- PEP 404 — сообщение об отмене выпуска Python 2.8;
- PEP 602 — ежегодный цикл выпусков Python.

Назначение, формат и процесс прохождения этих документов тоже были стандартизированы в форме отдельного документа PEP, а именно PEP 1.

Документы PEP играют очень важную роль для Python и в зависимости от темы служат разным целям:

- Информирование. Они содержат сводную информацию, необходимую ключевым разработчикам языка Python, и оповещают о графиках выпуска Python.
- Стандартизация. Они определяют правила по стилю кодирования, предоставляют документацию или другие рекомендации.
- Проектирование. Они онисывают предлагаемую функциональность.

Список всех предложенных PEP доступен в динамическом (то есть непрерывно обновляемом) документе PEP 0. Это отличный источник ипформации, если вам интересно, в каком направлении движется Python, но у вас нет времени отслеживать все обсуждения в списках рассылки. В PEP 0 указано, какие идеи были приняты, какие реализовапы, а какие все еще находятся в процессе рассмотрения.

Адреса документов PEP в интернете имеют формат:

http://www.python.org/dev/peps/pep-XXXX

XXXX — номер PEP, состоящий из четырех цифр с начальными нулями. Документы также легко ищутся в поисковых системах по запросу «Python PEP XXX».

Документ PEP 0 доступен по адресу https://www.python.org/dev/peps/.

Каталог всех официально обсуждаемых предложений выполняет еще одну задачу. Например, люди часто интересуются:

- Почему функциопальность A работает именно так?
- Почему в Python нет функциональности B?

В большинстве таких случаев ответ уже содержится в том докумепте PEP, где обсуждалась эта функциональпость. Вместо того чтобы снова и снова обсуждать добавление той или иной функциональности, можно нереадресовать интересующихся к конкретному PEP. Многие из документов PEP описывают языковые возможности Python, которые предлагались, но не были приняты.

Оставаться в курсе текущих событий в мире Python в первую очередь помогают открытые PEP с описапием идей, которые пока находятся в активном обсуждении. Нанример, вот некоторые интересные PEP, которые значились открытыми на момент написания книги:

- PEP 603 — Добавление типа frozenmap в коллекции.
- PEP 634 — Структурное сопоставление с шаблоном: спецификация.

- PEP 638 – Синтаксические макросы.
- PEP 640 – Синтаксис для неиспользуемых переменных.

Эти предложения лежат в диапазоне от относительно небольших расширений существующей стандартной библиотеки (как PEP 603) до совершенно новых сложных синтаксических средств (как PEP 638). Если вас интересует, какую функциональность сообщество Python планирует включить в будущие выпуски языка, то открытые документы PEP – лучший источник информации.

Активные сообщества

За языком Python стоит некоммерческая организация PSF (Python Software Foundation). Она является правообладателем интеллектуальной собственности на Python и управляет его лицензированием. Вот краткая выдержка из текста миссии PSF:

> *Миссия Python Software Foundation – развивать, защищать и продвигать язык программирования Python, а также поддерживать и стимулировать рост разностороннего международного сообщества программистов на Python.*

Полный текст миссии PSF доступен по адресу https://www.python.org/psf/mission/.

Для миссии PSF важно поддерживать сообщество программистов на Python, потому что в конечном итоге именно оно определяет путь развития Python. Для этого сообщество не только расширяет язык прозрачными и открытыми методами (как описано в разделе, посвященном документам PEP), но также пополняет и сопровождает богатую экосистему сторонних пакетов и фреймворков. Таким образом, один из лучших способов быть в курсе того, что происходит с Python, – это контактировать с сообществом.

Как свойственно любому языку программирования, существует много независимых интернет-сообществ, посвященных Python. Обычно они сосредоточены на конкретных фреймворках или областях разработки – веб-нрограммировании, анализе и обработке данных, машинном обучении и т. д.

Логично было бы предположить, что в интернете есть хотя бы одна центральная площадка, где ведутся все важные обсуждения на тему устройства языка и его интернретатора. К сожалению, все не так нросто.

По многим причинам, в том числе историческим, пространство официальных списков рассылки и форумов Python может показаться крайне занутанным.

Существует множество официальных и полуофициальных каналов коммуникации, где обитают разработчики Python. Особенная неразбериха связана со списками рассылки, потому что официальные рассылки распространяются через две разные службы, и поэтому существуют два отдельных архива рассылок:

- **Mailman 2**: более старый и меньший по объему архив, доступный по адресу https://mail.python.org/mailman/listinfo. Традиционно все архивы рассылок python.org были доступны в архиве Mailman 2, но теперь многие из них перенесены в Mailman 3. Впрочем, еще остаются рассылки с активными обсуждениями, которые управляются через Mailman 2.
- **Mailman 3**: более новый архив, доступный по адресу https://mail.python.org/archives. Сейчас это основной архив рассылок python.org, а также база для других активных рассылок. У него более современный и удобный интерфейс, но в нем нет рассылок, которые еще не мигрировали из Mailman 2.

Что касается самих рассылок, то их достаточно много, хотя, к сожалению, большинство уже неактивны. Некоторые рассылки посвящены конкретным проектам (например, scikit-image@python.org) или отдельным предметным областям (скажем, code-quality@python.org). Помимо рассылок с предельно узкой тематикой, есть несколько общих рассылок, которые представляют интерес для каждого программиста на Python:

- python-ideas@python.org: базовый список рассылки Python, где обсуждают широкий круг идей, связанных с этим языком. Многие документы PEP начинаются с умозрительных идей, выдвинутых в этой рассылке. Это отличное место, чтобы обсуждать потенциальные «что, если» и узнавать, какую функциональность подписчики хотели бы видеть в ближайшем будущем.
- python-dev@python.org: список рассылки, посвященный разработке ядра Python (в первую очередь интерпретатора CPython). Также здесь обсуждаются первые черновики новых PEP, прежде чем они будут официально анонсированы в других каналах. Это не то место, куда стоит обращаться за помощью общего характера с Python, но это ключевой ресурс для тех, кто хочет испытать свои силы в исправлении известных ошибок в интерпретаторе CPython или стандартной библиотеке Python.
- python-announce-list@python.org: список рассылки для разных объявлений. Здесь публикуются анонсы конференций и встреч, а также оповещения о выходе новых версий ваших любимых пакетов или фреймворков или новых PEP. Из рассылки также можно узнать о новых увлекательных проектах.

Кроме классических списков рассылки, существует официальный интернет-форум на платформе Discourse по адресу https://discuss.python.org/. Это относительно новая площадка в пространстве официальных обсуждений Python. По своему предназначению форум отчасти пересекается со многими традиционными списками рассылки, потому что на нем созданы категории для обсуж-

дения новых идей, PEP и разработки ядра Python. У этого форума более низкий входной порог для тех, кто не знаком с концепцией списков рассылки, и намного более современный пользовательский интерфейс.

К сожалению, не все новые обсуждения происходят на discuss.python.org, и если вы хотите быть в курсе всего, что творится в мире разработки Python, то вам придется отслеживать как форумы, так и рассылки. Хочется надеяться, что когда-нибудь все эти источники соединятся в одном месте.

Кроме официальных форумов и рассылок, существует несколько публичных сообществ Python на основе популярных коммуникационных платформ. Вот самые заметные сообщества:

- **Рабочее пространство PySlackers в Slack** (pyslackers.com). Большое сообщество энтузиастов Python на базе корпоративного мессенджера Slack.
- **Сервер Python на Discord** (pythondiscord.com). Другое публичное сообщество Python на платформе Discord.
- **«Саб» /r/python в Reddit** (www.reddit.com/r/Python/). Подфорум на платформе Reddit, посвященный Python.

Эти три сообщества являются открытыми в том смысле, что к ним можно свободно присоединиться, если вы зарегистрированы на соответствующей платформе (причем все они, конечно, бесплатны). Вероятно, ваш выбор будет определяться тем, на какой платформе вы предпочитаете обмениваться сообщениями или вести обсуждения. Не исключено, что вы или кто-нибудь из ваших друзей уже пользуется чем-то из перечисленного.

Бесспорное преимущество подобных открытых сообществ в том, что в них очень много участников, так что вы почти всегда найдете, с кем пообщаться. Это открывает простор для свободных неформальных обсуждений на различные темы, связанные с Python, и позволяет быстро получить помощь, когда возникают простые проблемы с программированием.

Обратная сторона медали в том, что невозможно отслеживать абсолютно все обсуждения в сообществе. К счастью, в сообществах часто организованы системы отдельных подканалов или тегов, на которые можно подписаться, чтобы получать оповещения на интересующие вас темы. Обратите внимание, что эти каналы не поддерживаются официально и не курируются PSF. В результате информация на Reddit или в других сообществах иногда оказывается предвзятой или неточной.

Другие ресурсы

Читать все новые документы PEP, отслеживать списки рассылки и участвовать в сообществах — лучший способ оставаться в курсе того, что происходит в мире

Python. К сожалению, все это требует много времени и усилий, потому что приходится отфильтровывать огромные объемы информации. Кроме того, в таких источниках, как списки рассылки, форумы и платформы обмена сообщениями, часто бушуют человеческие эмоции. Некоторые технические обсуждения на неоднозначные темы оказываются настолько жаркими, что мало чем отличаются от драм в социальных сетях.

Если у вас мало времени или вы быстро устаете от общения с незнакомыми людьми в Сети, есть другой вариант. Вместо того чтобы фильтровать входную информацию самостоятельно, можно обратиться к специально отобранному контенту — блогам, новостным рассылкам и так называемым спискам классных ресурсов.

Рассылки особенно хороши, если вы хотите оставаться в курсе дела. Вот пара интересных рассылок, на которые стоит подписаться:

- **Python Weekly** (http://www.pythonweekly.com/) — популярная рассылка, которая еженедельно предлагает подписчикам десятки новых интересных пакетов и других ресурсов Python.
- **PyCoder Weekly** (https://pycoders.com) — другая популярная еженедельная рассылка со сводкой новых пакетов и интересных статей.

Эти рассылки будут держать вас в курсе дела о самых важных событиях в мире Python. Они также помогут находить новые блоги или интересные обсуждения на других платформах (таких, как Reddit или Hacker News). Содержимое многих рассылок общего характера часто пересекается, так что вряд ли стоит подписываться сразу на все.

Совершенно по-иному организована информация в «списках классных ресурсов» (awesome lists). Это перечни специально отобранных ссылок на полезные и важные ресурсы по конкретным темам, которые обычно поддерживаются в виде репозиториев Git на GitHub. Такие списки часто получаются очень длинными и разбиваются на категории.

Вот примеры популярных списков классных ресурсов про Python, которые составляют различные пользователи GitHub:

- **awesome-python от vinta** (https://github.com/vinta/awesome-python): многочисленные ссылки на интересные проекты (в основном размещенные на GitHub) и стандартные библиотечные модули. Ссылки разделены на 80 тематических категорий: от базовых концепций программирования (кэширование, аутентификация, отладка и т. д.) до целых областей IT, в которых часто применяется Python: веб-программирование, анализ и обработка

данных, робототехника, кибербезопасность. Помимо проектов, в список входят ссылки на новостные рассылки, подкасты, книги и обучающие руководства.

- **pycrumbs от kirang89** (https://github.com/kirang89/pycrumbs): в этом списке основное внимание уделяется интересным и ценным статьям. Статьи разделены более чем на 100 категорий, посвященных конкретным возможностям Python, общим вопросам программирования и саморазвитию.
- **pythonidae от svaksha** (https://github.com/svaksha/pythonidae): этот список систематизирован в соответствии с областями науки и техники, где часто применяется Python: математика, биология, химия, веб-программирование, физика, обработка изображений и многие другие. У списка древовидная структура. Список на главной странице содержит свыше 20 основных категорий, которые, в свою очередь, разделяются на более узкие подкатегории с перечнями библиотек и ресурсов.

Со временем «списки классных ресурсов» обычно разрастаются до невероятных размеров, и из них становится неудобно получать актуальную информацию. Дело в том, что это просто «моментальные снимки» того, что составители считали «классным» на момент включения в список. Тем не менее если вам нужно погрузиться в совершенно новую область (допустим, искусственный интеллект), такие списки будут хорошей отправной точкой для дальнейших изысканий.

Итоги

В этой главе мы рассмотрели текущее состояние Python и ход изменений, прослеживающихся в истории этого языка. Вы узнали, почему Python меняется и почему важно следить за этими переменами.

Поддерживать знания и навыки в актуальном состоянии — одно из самых серьезных испытаний, с которыми сталкиваются профессиональные программисты независимо от языка программирования. С учетом 30-летней истории Python и его постоянно расширяющегося сообщества не всегда ясно, что делать, чтобы не отставать от изменений в экосистеме Python. Вот почему в этой главе перечислены ресурсы, с помощью которых можно следить за важными обсуждениями будущего Python.

Вместе с языком меняются и средства разработки, призванные упростить и усовершенствовать процессы разработки. В следующей главе мы продолжим тему изменений и рассмотрим современные среды разработки. Вы узнаете, как налаживать воспроизводимые и целостные среды исполнения — и для разработки, и для эксплуатации. Вы также познакомитесь с различными средствами повышения производительности, которые предлагает сообщество Python.

2

Современные среды разработки для Python

Знать язык программирования на глубоком уровне — самое важное качество пастоящего профессионала. Тем не мепее тяжело эффективно разрабатывать хорошие программные продукты, не разбираясь в том, какие популярные инструменты и практики приняты в сообществе этого языка. В Python нет ничего такого, что нельзя было бы найти в каком-то еще языке. Если сравнивать синтаксис, выразительность или быстродействие, то всегда найдется решение, которое превосходит Python по одному или нескольким критериям. Однако кое в чем Python все-таки выделяется на фоне конкурентов: его уникальный атрибут — целая экосистема, сформировавшаяся вокруг языка. Вот уже много лет сообщество Python совершенствует стандартпые методы разработки и библиотеки, которые помогают создавать качественное ПО за более короткое время.

Писать новые программы — дорогостоящий и долгий процесс. Впрочем, когда есть возможность повторно использовать существующий код вместо того, чтобы «изобретать велосипед», это радикально сокращает время и затраты на разработку. В некоторых комнаниях это единственная нричина, по которой их проекты оказываются экономически оправданными. Поэтому важнейшей частью экосистемы является богатая коллекция пакетов для решения всевозможных задач. Огромное количество таких пакетов с открытым кодом доступно в каталоге пакетов Python (PyPI, Python Package Index).

Поскольку сообщество, опирающееся на открытый исходный код, играет важную роль в развитии Python, разработчики языка приложили значительные усилия, чтобы создать инструменты и стандарты для работы с чужими пакетами Python. Среди этих инструментов — виртуальные изолированные среды, улучшенные интерактивные оболочки, отладчики, а также программы, которые номогают ориентироваться в огромной коллекции пакетов, доступных в PyPI.

В этой главе рассматриваются следующие темы:

- Обзор экосистемы пакетов Python.
- Изоляция среды выполнения.
- Использование venv.
- Изоляция среды на уровне системы.
- Популярные средства повышения производительности.

Но прежде чем изучать конкретные элементы экосистемы Python, начнем с технических требований.

Технические требования

Бесплатные средства системной виртуализации, о которых идет речь в этой главе, можно загрузить на соответствующих сайтах:

- Vagrant: https://www.vagrantup.com
- Docker: https://www.docker.com
- VirtualBox: https://www.virtualbox.org

В главе также упоминаются пакеты Python, которые можно загрузить из каталога PyPI:

- `poetry`
- `flask`
- `wait-for-it`
- `watchdog`
- `ipython`
- `ipdb`

О том, как устанавливать пакеты, рассказано в разделе «Установка пакетов Python с помощью pip».

Файлы с кодом из этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%202.

Экосистема пакетов Python

В экосистеме пакетов Python центральное место занимает каталог Python Packaging Index (PyPI). Это огромный общедоступный репозиторий преиму-

щественно бесплатных проектов на Python, который на момент написания книги содержал почти 3,5 миллиона дистрибутивов для более чем 250 000 пакетов. Это не самый крупный репозиторий пакетов (объем npm в 2019 году превысил миллион пакетов), но с такими показателями Python все равно остается среди лидирующих экосистем пакетов.

У такой громадной экосистемы есть свои минусы. В современных приложениях часто используется по несколько пакетов из PyPI, у которых бывают собственные зависимости, а у тех — свои зависимости. В больших приложениях такие цепочки зависимостей могут чрезвычайно растягиваться. А если учесть, что некоторые пакеты требуют конкретных версий других пакетов, то можно быстро оказаться в «аду зависимостей», то есть в ситуации, когда разрешить конфликт версий вручную практически невозможно.

Поэтому так важно разбираться в инструментах, которые помогают работать с пакетами из PyPI.

Установка пакетов Python с помощью pip

В наше время Python является стандартным компонентом многих операционных систем. В большинстве дистрибутивов Linux и систем семейства UNIX (таких, как FreeBSD, NetBSD, OpenBSD и macOS) Python либо установлен по умолчанию, либо доступен в репозиториях системных пакетов. Многие из этих систем даже используют Python в составе базовых компонентов: на его основе работают программы установки Ubuntu (Ubiquity), Red Hat Linux (Anaconda) и Fedora (снова Anaconda). К сожалению, с операционными системами часто поставляется Python не самой свежей версии.

Ввиду популярности Python как компонента ОС многие пакеты из PyPI также доступны в формате портированных пакетов, совместимых со средствами управления пакетами той или иной системы, такими как `apt-get` (Debian, Ubuntu), `rpm` (Red Hat Linux) или `emerge` (Gentoo).

Однако набор доступных библиотек часто ограничен, и они по большей части оказываются устаревшими по сравнению с аналогичными пакетами из PyPI. Иногда они распространяются с исправлениями под соответствующую платформу, чтобы гарантировать совместимость с другими системными компонентами.

В свете всего сказанного при разработке новых приложений всегда следует полагаться на дистрибутивы пакетов, доступные в PyPI. Для установки пакетов группа специалистов по сопровождению стандартных пакетных инструментов **PyPA** (Python Packaging Authority) рекомендует использовать `pip`. Эта программа командной строки позволяет устанавливать пакеты прямо из PyPI. Начиная с версий CPython 2.7.9 и 3.4, в каждый выпуск Python вклю-

чается служебный модуль **ensurepip**, который гарантирует, что **pip** будет установлен в вашей среде независимо от того, был ли он включен в дистрибутив. Установку **pip** можно инициировать с помощью **ensurepip**, как в этом примере:

```
$ python3 -m ensurepip

Looking in links: /var/folders/t6/n6lw_s3j4nsd8qhsl1jhgd4w0000gn/T/
tmpouvorgu0
Requirement already satisfied: setuptools in ./.venv/lib/python3.9/
site-packages (49.2.1)
Processing /private/var/folders/t6/n6lw_s3j4nsd8qhsl1jhgd4w0000gn/T/
tmpouvorgu0/pip-20.2.3-py2.py3-none-any.whl
Installing collected packages: pip
Successfully installed pip-20.2.3
```

Когда **pip** доступен, новые пакеты можпо устанавливать простой командой:

```
$ pip install <имя_пакета>
```

Таким образом, чтобы установить пакет с именем **django**, достаточно выполнить команду:

```
$ pip install <имя_пакета>
```

Среди прочего, **pip** позволяет устанавливать конкретные версии пакетов (команда **pip install <имя_пакета>==<версия>**) или обповлять пакеты до иоследней доступной версии (комапда **pip install --upgrade <имя_пакета>**).

Возможности **pip** не ограничиваются установкой пакетов. Помимо **install**, поддерживаются дополнительные команды, с номощью которых можно получать информацию о пакетах, выполпять поиск но PyPI или собирать собственные дистрибутивы пакетов. Список всех доступных комапд выводится командой

```
$ pip install <имя_пакета>
```

Опа выдает следующий результат[1]:

```
Использование:
  pip <command> [options]

Команды:
  install                 Установить пакеты.
  download                Загрузить пакеты.
```

[1] Вывод команды будет на английском языке, ниже приводится неревод. — *Примеч. ред.*

```
  uninstall              Удалить пакеты.
  freeze                 Вывести список установленных пакетов в формате
                             требований.
  list                   Вывести список установленных пакетов.
  show                   Вывести информацию об установленных пакетах.
  check                  Проверить, что у установленных пакетов есть
                             совместимые зависимости.
  config                 Управлять локальной и глобальной конфигурацией.
  search                 Искать пакеты в PyPI.
  cache                  Работать с кэшем установочных комплектов pip.
  wheel                  Собрать пакеты по вашим требованиям.
  hash                   Вычислить хеш-сумму архивированных пакетов.
  completion             Настроить автозавершение команд в командной строке.
  debug                  Показать информацию для отладки.
  help                   Показать справку по командам.
(...)
```

Самая свежая информация о том, как установить `pip` для более старых версий Python, доступна по адресу https://pip.pypa.io/en/stable/installing/.

Изоляция среды выполнения

Когда вы с помощью `pip` устанавливаете новый пакет из PyPI, он будет установлен в один из доступных **каталогов пакетов**. Точное местонахождение этих каталогов зависит от операционной системы. Чтобы просмотреть пути к каталогам, в которых Python будет искать модули и пакеты, используйте модуль `site` как команду:

```
$ python3 -m site
```

Пример вывода команды `python3 -m site` в macOS:

```
sys.path = [
    '/Users/swistakm',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python39.
zip',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/
lib-dynload',
    '/Users/swistakm/Library/Python/3.9/lib/python/site-packages',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/
site-packages',
]
USER_BASE: '/Users/swistakm/Library/Python/3.9' (exists)
USER_SITE: '/Users/swistakm/Library/Python/3.9/lib/python/site-
packages' (exists)
ENABLE_USER_SITE: True
```

В этом выводе переменная `sys.path` содержит список каталогов, в которых выполняется поиск модулей. Это места, откуда Python пытается загружать модули. На первом месте всегда стоит текущий рабочий каталог (в данном случае /users/swistakm), а на последнем — глобальный каталог пакетов.

`USER_SITE` в этом выводе — путь к локальному каталогу пакетов пользователя, от имени которого запущен текущий интерпретатор Python. У пакетов, установленных в пользовательском каталоге, более высокий приоритет, чем у пакетов в глобальном каталоге. Чтобы узнать путь к глобальному каталогу пакетов, можно также вызвать функцию `sys.getsitepackages()`. В следующем примере она используется в интерактивной оболочке:

```
>>> import site
>>> site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages']
```

Пути к пользовательским каталогам пакетов можно также получить с помощью функции `sys.getusersitepackages()`:

```
>>> import site
>>> site.getusersitepackages()
/Users/swistakm/Library/Python/3.9/lib/python/site-packages
```

При выполнении команды `pip install` пакеты будут установлены либо в локальный, либо в глобальный каталог пакетов в зависимости от нескольких условий, которые проверяются в таком порядке:

1. **Пользовательский каталог**, если указан параметр `--user`.
2. **Глобальный каталог**, если пользователю, запустившему `pip`, разрешена запись в этот каталог.
3. **Пользовательский каталог** во всех остальных случаях.

Эти условия попросту означают, что `pip` без параметра `--user` всегда пытается устанавливать пакеты в глобальный каталог пакетов, и только если это невозможно, переключается на пользовательский каталог. В большинстве операционных систем, где Python доступен по умолчанию (macOS и многие дистрибутивы Linux), глобальный каталог пакетов системного дистрибутива Python защищен от записи со стороны непривилегированных пользователей. А значит, чтобы установить пакет в глобальный каталог с помощью системного дистрибутива Python, нужно использовать команду, которая дает привилегии суперпользователя (например, `sudo`). В системах семейства UNIX и Linux запуск `pip` с правами суперпользователя будет выглядеть так:

```
$ sudo -H pip install <имя_пакета>
```

В Windows не нужны привилегии суперпользователя, чтобы устанавливать пакеты Python на уровне системы, потому что интерпретатор Python не устанавливается в этой ОС по умолчанию. Кроме того, в некоторых других операционных системах (таких, как macOS) Python устанавливается так, что глобальный каталог пакетов будет доступен для записи обычным пользователям (если использовать установщик с сайта python.org).

Можно устанавливать пакеты непосредственно из PyPI в глобальный каталог пакетов, причем в некоторых средах это происходит по умолчанию, однако, как правило, так поступать не рекомендуется. Следует помнить, что `pip` устанавливает в каталог только одну версию пакета. Если в каталоге уже есть более старая версия, то новая установка ее заменит. Это может создать проблемы, особенно если вы планируете разрабатывать на Python более одного приложения. Совет не устанавливать ничего в глобальный каталог пакетов может показаться странным, потому что такое поведение более или менее стандартно для `pip`, но для этого есть веские причины.

Как упоминалось ранее, Python является важным компонентом многих пакетов, доступных в репозиториях пакетов операционной системы, и иногда он обеспечивает работу важных служб. Специалисты по сопровождению системных дистрибутивов прикладывают значительные усилия, чтобы выбрать правильные версии пакетов в соответствии с их многочисленными зависимостями.

Очень часто пакеты Python, доступные в системном репозитории (например, `apt`, `yum` или `rpm`), содержат специальные исправления или намеренно не обновляются до новых версий, чтобы поддерживать совместимость с другими системными компонентами. Если с помощью `pip` принудительно обновить такой пакет до версии, нарушающей обратную совместимость, это может привести к ошибкам в критических системных службах.

И наконец, если вы работаете параллельно над несколькими проектами, то заметите, что практически невозможно поддерживать единый список версий пакетов, которые подходят для всех проектов. Пакеты развиваются быстро, и не все изменения сохраняют обратную совместимость. Рано или поздно вы столкнетесь с ситуацией, когда одному из ваших новых проектов непременно нужна новейшая версия некоторой библиотеки, но другой проект не может использовать эту версию из-за какого-то изменения, нарушившего обратную совместимость. Если пакет установлен в глобальный каталог, то всем проектам будет доступна только одна версия этого пакета.

К счастью, у этой проблемы есть простое решение — изоляция среды. Существует много разных инструментов, которые изолируют среду выполнения Python на разных уровнях системной абстракции. Главная идея заключается в том, чтобы изолировать зависимости проекта от пакетов, которые

нужны другим проектам и/или системным службам. Преимущества такого нодхода:

- Он решает дилемму «проект X зависит от пакета 1.x, но проекту Y нужен пакет 4.x». У разработчика появляется возможность работать над разными проектами с разными зависимостями, которые могут хоть нолностью противоречить друг другу, но при этом никак не влиять друг на друга.
- Проект больше не ограпичивается версиями пакетов, которые предоставляются в репозиториях системного дистрибутива разработчика (`apt`, `yum`, `rpm` и т. д.).
- Исчезает риск нарушепия работоспособности других системпых служб, которые зависят от определепных версий пакетов, потому что новые версии пакетов доступны только в изолировапной среде.
- Можпо легко зафиксировать список пакетов, которые образуют зависимости проекта. При этом обычно фиксируются точные версии всех пакетов со всеми цепочками зависимостей, чтобы такую среду можпо было легко воспроизвести на другом компьютере.

Если вы работаете пад несколькими проектами параллельно, то быстро выясняется, что их зависимостями невозможно управлять без изоляции того или иного рода.

В следующем разделе вы узнаете, чем изоляция на уровне приложепия отличается от изоляции на уровне системы.

Изоляция на уровне приложения и на уровне системы

Самый простой подход к изоляции, требующий наименьших затрат ресурсов, — изоляция на уровне приложения с помощью виртуальных сред. В Python есть встроенпый модуль `venv`, который заметно упрощает создание и использование таких сред.

Основная задача виртуальных сред — изолировать интерпретатор Python и доступные в нем пакеты. Такие среды легко настраивать, но они непереносимы — в основпом потому, что зависят от абсолютных системных путей. Это знпчит, что их нельзя легко скопировать с одного компьютера на другой, сохранив работоспособность. Их даже нельзя перемещать между каталогами в одной файловой системе. Тем не менее возможностей таких сред хватает, чтобы обеспечивать необходимую изоляцию при разработке небольших проектов и пакетов. Благодаря встроенпой поддержке в дистрибутивах Python ваши коллеги также могут легко их воспроизводить.

Виртуальных сред обычно достаточно, чтобы разрабатывать снециализированные библиотеки, не зависящие от операционной системы, или проекты невысокой сложности, у которых нет многочисленных внешних зависимостей. Кроме того, если вы нишете программы, которые должны выполняться только на вашем собственном компьютере, то виртуальные среды обычно обеспечивают необходимый уровень изоляции и воспроизводимости.

К сожалению, в некоторых случаях такой нодход не гарантирует должной логической целостности и воспроизводимости. Хотя программы, написанные на Python, обычно хорошо переносимы, не каждый пакет ведет себя одинаково во всех операционных системах. Это справедливо в первую очередь для пакетов, которые зависят от сторонних совместно используемых библиотек (DLL в Windows, `.so` в Linux, `.dylib` в macOS) или интенсивно используют компилированные расширения Python, написанпые на C и C++. Проблемы также возпикают с библиотеками, написапными на чистом Python, которые обращаются к специфическим API для конкретной операционной системы.

В подобных случаях в нроцесс разработки имеет смысл добавить изоляцию на уровне системы. Такие решения обычно пытаются воспроизводить и изолировать целые операционные системы со всеми библиотеками и критическими системными компонентами: либо с помощью классических средств визуализации ОС (например, VMware, Parallels и VirtualBox), либо на основе контейнерных систем (например, Docker и Rocket). Некоторые из доступных решений, обеспечивающих изоляцию такого рода, подробнее описаны ниже в разделе «Изоляция сред на уровне системы».

В среде разработки стоит предпочесть изоляцию на уровне системы, если вы нишете программы не на том компьютере, на котором они будут выполняться. Если ваши программы работают на удаленных серверах, то стоит с самого начала использовать изоляцию на уровне системы, потому что это может предотвратить проблемы с переносимостью в будущем. Так следует поступать независимо от того, использует ли ваше приложение компилированный код (совместно используемые библиотеки, компилированные расширения). Об изоляции на уровне системы также стоит задуматься, если приложение активно обращается к впешним службам: базам данных, кэшам, поисковым системам и т. д. Многие инструменты изоляции па уровне системы позволяют также легко изолировать такие зависимости.

Так как оба подхода к изоляции сред находят свое место в современной разработке Python, мы рассмотрим их подробнее. Начнем с более простого — виртуальных сред, которые испольэуют модуль Python `venv`.

Изоляция сред на уровне приложений

В Python есть встроенный инструмент для создания виртуальных сред — модуль venv, который можно запустить непосредственно из системной оболочки. Чтобы создать новую виртуальную среду, введите следующую команду:

```
$ python3.9 -m venv <имя_среды>
```

Здесь имя_среды заменяется именем новой среды (вместо него можно указать абсолютный путь). Обратите внимание, что используется команда python3.9, а не просто python3. Дело в том, что в зависимости от среды команда python3 может быть нривязана к разным версиям интерпретатора, так что всегда лучше предельно ясно указать, какую конкретно версию Python использовать для создания новой виртуальной среды. Команда python3.9 -m venv создает новый каталог имя_среды в текущем рабочем каталоге. В созданном каталоге находятся несколько подкаталогов:

- bin/: здесь хранится новый исполняемый файл Python и сценарии или исполняемые файлы, предоставляемые другими пакетами.

ПРИМЕЧАНИЕ ДЛЯ ПОЛЬЗОВАТЕЛЕЙ WINDOWS

В Windows модуль venv придерживается других соглашений об именах во внутренней структуре каталогов. Вместо bin/, lib/ и include/ следует использовать Scripts/, Libs/ и Include/ в соответствии с соглашениями, принятыми в этой ОС. Команды для активации и деактивации среды тоже различаются: вместо source со сценариями activate и deactivate нужно вызывать ИМЯ_СРЕДЫ/Scripts/activate.bat и ИМЯ_СРЕДЫ/Scripts/deactivate.bat.

- lib/ и include/: в этих каталогах находятся вспомогательные библиотечные файлы для пового интерпретатора Python в виртуальной среде. Новые пакеты будут устанавливаться в каталог ИМЯ_СРЕДЫ/lib/pythonX.Y/site-packages/.

Разработчики часто хранят свои виртуальные среды вместе с исходным кодом и называют каталоги сред типовыми именами вида .venv или venv. Многие интегрированные среды разработки (IDE) для Python распознают эти имена и автоматически подгружают библиотеки для автодополнения кода. Типовые имена также позволяют автоматически исключать каталоги виртуальных сред из систем управления версиями, что обычно рекомендуется делать. Например, пользователи Git могут добавить эти имена в глобальный файл .gitignore с шаблонами путей, которые надо игнорировать при управлении версиями исходного кода.

После того как новая среда будет создана, ее надо активировать в текущем сеансе командной оболочки. Если вы используете оболочку Bash, виртуальную среду можно активировать командой `source`:

```
$ source имя_среды/bin/activate
```

Есть и более короткая версия команды, которая должна работать в любой POSIX-совместимой системе независимо от оболочки:

```
$ source имя_среды/bin/activate
```

Эта команда изменяет состояние текущего сеанса оболочки, затрагивая ее переменные окружения. Чтобы пользователь понимал, что виртуальная среда активирована, приглашение в командной строке изменяется: в его начало добавляется строка (`ИМЯ_СРЕДЫ`). Вот пример, в котором создается и активируется новая среда:

```
$ python3 -m venv example
$ source example/bin/activate
(example) $ which python
/home/swistakm/example/bin/python
(example) $ deactivate
$ which python
/usr/local/bin/python
```

Важно заметить, что `venv` не предоставляет никаких дополнительных возможностей, чтобы отслеживать, какие пакеты должны быть установлены в виртуальной среде. Кроме того, виртуальные среды непереносимы и их не следует перемещать в другую систему, на другой компьютер и даже в другое место этой же файловой системы. Это значит, что каждый раз, когда вы захотите установить свое приложение на новом хосте, придется создавать новую виртуальную среду.

Из-за этого пользователи `pip` выработали полезный принцип: хранить спецификацию всех зависимостей проекта в одном месте. Проще всего для этого создать файл требований с общепринятым именем `requirements.txt`, пример содержимого которого показан ниже:

```
# Строки, начинающиеся с решетки (#), считаются комментариями.

# фиксированные спецификаторы версий лучше всего для воспроизводимости
eventlet==0.17.4
graceful==0.1.1

# для проектов, тщательно протестированных с разными
# версиями зависимостей, допускаются диапазоны версий
falcon>=0.3.0,<0.5.0

# следует избегать пакетов без указания версии, кроме ситуации,
# когда всегда желателен/необходим последний выпуск
pytz
```

С таким файлом все зависимости можно легко установить за один шаг. Команда `pip install` поддерживает формат подобных файлов требований. Путь к файлу можно указать с флагом `-r`, как в примере:

```
$ pip install -r requirements.txt
```

Учтите, что в файлах требований перечисляются только те пакеты, которые надо установить, а не те, которые уже установлены в вашей среде. Если вы установите какой-то пакет вручную, он не будет автоматически отражен в файле требований. Таким образом, нужно специально заботиться о том, чтобы файл требований поддерживался в актуальном состоянии, особенно в больших и сложных проектах.

Команда `pip freeze` выводит список всех пакетов в текущей среде вместе с их версиями, однако использовать ее следует с осторожностью. В список также попадут зависимости ваших зависимостей, так что в больших проектах он очень быстро разрастается. Вам придется тщательно проверять, нет ли в списке пакетов, установленных по ошибке или случайно.

Если проект требует улучшенной воспроизводимости виртуальных сред и жесткого контроля за установленными зависимостями, то вам может понадобиться более мощный инструмент. Один из таких инструментов — Poetry — будет рассмотрен в следующем разделе.

Poetry кок система управления зависимостями

Poetry реализует относительно новый подход к управлению зависимостями и виртуальными средами в Python. Цель этого проекта с открытым кодом — предоставить более предсказуемую и удобную среду для работы с экосистемой пакетов Python.

Так как Poetry является пакетом PyPI, его можно установить командой `pip`:

```
$ pip install -r requirements.txt
```

Обратите внимание, что Poetry создает виртуальные среды Python, поэтому этот пакет не следует устанавливать внутри самой виртуальной среды. Poetry можно установить в пользовательский или глобальный каталог пакетов, хотя пользовательский каталог считается предпочтительным (см. раздел «Изоляция среды выполнения».

Как уже упоминалось в разделе «Установка пакетов Python с помощью pip», эта команда устанавливает Poetry в каталог пакетов. В зависимости от конфигурации системы это будет либо глобальный, либо пользовательский каталог. Что-

бы избежать этой неоднозначности, создатели Poetry рекомендуют использовать альтернативный метод инициализации.

В macOS, Linux и других POSIX-совместимых системах Poetry можно установить с помощью утилиты `curl`:

```
$ curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/
master/get-poetry.py | python
```

В Windows для установки можно использовать PowerShell:

```
> (Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-
poetry/poetry/master/get-poetry.py -UseBasicParsing).Content | python -
```

После установки Poetry можно применять для следующих целей:

- Создавать новые нроекты Python вместе с виртуальными средами.
- Инициализировать существующие проекты в виртуальной среде.
- Управлять зависимостями проектов.
- Упаковывать библиотеки.

Чтобы в Poetry создать совершенно новый нроект, занустите команду `poetry new`, как в этом нримере:

```
$ poetry new my-project
```

Эта команда создает новый каталог **my-project**, содержащий несколько исходных файлов. Структура каталога выглядит примерно так:

```
my-project/
├── README.rst
├── my_project
│   └── __init__.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_my_project.py
```

Как видите, создалось несколько файлов, которые можно использовать как заглушки для дальнейшей разработки. Чтобы инициализировать Poetry в уже существующем нроекте, запустите команду `poetry init` в каталоге проекта. Она не создаст новых файлов проекта, кроме файла **pyproject.toml**.

Этот файл — **pyproject.toml** — является центральным компонентом Poetry и описывает конфигурацию проекта. Для `my-project` из нредыдущего примера его содержимое может выглядеть так:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Michał Jaworski <swistakm@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Как видно из листинга, файл pyproject.toml делится на четыре раздела:

- `[tool.poetry]`: основные метаданные проекта: название, описание версии, автор и т. д. Эта информация необходима, если вы собираетесь опубликовать проект в виде пакета в PyPI.
- `[tool.poetry.dependencies]`: список зависимостей проекта. В новых проектах этот раздел содержит только версию Python, но в нем также могут перечисляться версии всех пакетов, которые обычно описываются в файле requirements.txt.
- `[tool.poetry.dev-dependencies]`: список зависимостей, необходимых для локальной разработки, например тестовые фреймворки или инструменты разработчика. На практике принято создавать отдельные списки таких зависимостей, потому что в средах реальной эксплуатации они обычно не нужны.
- `[build-system]`: описание Poetry как системы сборки, используемой для управления проектом.

Файл pyproject.toml определен в официальном стандарте Python (документ PEP 518). Подробнее ознакомиться с его структурой можно по адресу https://www.python.org/dev/peps/pep-0518/.

Если вы создаете новый проект или инициализируете существующий проект в Poetry, эта программа сможет в любой момент создать новую виртуальную среду в совместно используемом расположении. Ее можно активировать средствами Poetry, вместо того чтобы вызывать `source` для сценариев `activate`. Это удобнее, чем использовать модуль `venv`, потому что вам не придется запоминать, где фактически хранится виртуальная среда. Все, что вам нужно, — перейти в оболочке в любое место дерева исходного кода проекта и ввести команду `poetry shell`, как в этом примере:

```
$ cd my-project
$ poetry shell
```

С этого момента виртуальная среда Poetry будет активирована в текущем сеансе оболочки. Чтобы в этом убедиться, введите команду `which python` или `python -m site`.

С помощью Poetry также проще управлять зависимостями. Как уже упоминалось, файлы requirements.txt — очень примитивный механизм унравления зависимостями. Они онисывают, какие накеты надо установить, но не отслеживают автоматически, что было установлено в среде в нроцессе разработки. Если вы установите какой-то пакет с помощью `pip`, но забудете отразить это изменение в файле requirements.txt, то другие нрограммисты могут столкнуться с затруднениями, воспроизводя вашу среду.

С Poetry эта проблема исчезает. Добавлять зависимости в проект можно только одним способом, и этот способ — команда `poetry add <имя_пакета>`. Вот что она делает:

- Обрабатывает целые деревья зависимостей, если другие пакеты совместно используют зависимости.
- Устанавливает все пакеты из дерева зависимостей в виртуальной среде, связанной с проектом.
- Отражает изменения в файле pyproject.toml.

В следующем примере показан процесс установки фреймворка Flask в среде `my-project`:

```
$ poetry add flask
```

Результат выполнения команды выглядит так:

```
Using version ^1.1.2 for Flask

Updating dependencies
Resolving dependencies... (38.9s)

Writing lock file

Package operations: 15 installs, 0 updates, 0 removals

  • Installing markupsafe (1.1.1)
  • Installing pyparsing (2.4.7)
  • Installing six (1.15.0)
  • Installing attrs (20.3.0)
  • Installing click (7.1.2)
  • Installing itsdangerous (1.1.0)
```

```
  • Installing jinja2 (2.11.2)
  • Installing more-itertools (8.6.0)
  • Installing packaging (20.4)
  • Installing pluggy (0.13.1)
  • Installing py (1.9.0)
  • Installing wcwidth (0.2.5)
  • Installing werkzeug (1.0.1)
  • Installing flask (1.1.2)
  • Installing pytest (5.4.3)
```

А вот как выглядит файл pyproject.toml с выделенным изменением в зависимостях проекта:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Michał Jaworski <swistakm@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.9"
Flask = "^1.1.2"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
```

В предыдущем примере Poetry установил 15 накетов, хотя мы запросили всего одну зависимость. Дело в том, что у Flask есть свои собственные зависимости, а у этих зависимостей — тоже свои зависимости. Такие «зависимости зависимостей» называются **транзнтнвнымн завнснмостями**. Для библиотек часто иснользуются нестрогие снецификаторы версий вида `six >=1.0.0`, которые допускают широкий дианазон версий. Poetry реализует алгоритм разрешения зависимостей, чтобы составить набор версий, который удовлетворяет всем ограничениям транзитивных зависимостей.

Проблема транзитивных зависимостей — в том, что они могут меняться со временем. Всномните, что библиотеки могут задавать нестрогие спецификаторы версий для своих зависимостей. Может оказаться, что в двух средах, созданных в разное время, используются разные финальные версии пакетов. Невозможность воспроизвести точные версии всех транзитивных зависимостей может создать проблемы для крупных проектов, а вручную отслеживать все зависимости в файле requirements.txt — неблагодарное занятие.

Poetry решает проблему транзитивных зависимостей с помощью так называемых **файлов блокировкн завнсимостей**. Когда вы уверены, что в вашей среде сло-

жился работоспособный и протестированный набор версий пакетов, можно занустить такую команду:

```
$ poetry lock
```

Команда создает чрезвычайно пространный файл poetry.lock, в котором содержится полный снимок процесса разрешения зависимостей. Затем этот файл используется для подбора версий транзитивных зависимостей вместо обычного процесса разрешения зависимостей.

Каждый раз, когда с помощью команды `poetry add` добавляются новые пакеты, Poetry анализирует дерево зависимостей и обновляет файл poetry.lock. На сегодняшний день файл блокировки — лучший и самый надежный способ управлять транзитивными зависимостями в проекте.

Дополнительную информацию о расширенных возможностях Poetry можно найти в официальной документации по адресу https://python-poetry.org.

Изоляция сред на уровне системы

Ключевой фактор быстрых итераций при разработке ПО — повторное использование существующих программных компонентов. «Не новторяйтесь» (DRY, Don't Repeat Yourself) — расхожая мантра среди программистов. Включение сторонних пакетов и модулей в кодовую базу — лишь часть этой идеологии. Повторно используемыми компонентами также можно считать двоичные библиотеки, базы данных, системные службы, сторонние API и т. д. Даже целые операционные системы можно рассматривать как повторно используемые компоненты.

Серверная часть (backend) веб-приложений — отличный нример того, насколько сложными могут быть такие приложения. Простейший программный стек обычно состоит из нескольких уровней. Рассмотрим воображаемое нриложение, которое хранит некую информацию о своих пользователях и предоставляет доступ к ней через интернет по протоколу HTTP. На практике приложения такого рода содержат как минимум три уровня (начиная с низшего):

- База данных или другая разновидность хранилища.
- Код приложения, реализованный на Python.
- Сервер HTTP, работающий в режиме обратного прокси-сервера (например, Apache или NGINX).

Очень простые приложения могут быть одноуровневыми, но это обычно не относится к приложениям, которые сложно устроены или должны обрабатывать

интенсивный трафик. В реальности большие приложения иногда настолько сложны, что их нельзя представить в виде вертикального стека слоев, а можно только в виде «лоскутного одеяла» из взаимосвязанных служб. И маленькие и большие приложения могут использовать несколько разных баз данных, делиться на независимые процессы и задействовать другие системные службы для кэширования, управления очередями, журналирования, обнаружения служб и т. д. К сожалению, этим усложнениям нет предела.

Здесь действительно важно то, что не все элементы программного стека можно изолировать на уровне среды выполнения Python. HTTP-сервер (например, nginx), СУБД (например, PostgreSQL) или совместно используемая библиотека — все эти компоненты обычно не входят в дистрибутив Python или экосистему пакетов Python и не инкапсулируются в виртуальных средах Python. Поэтому они считаются внешними зависимостями программного продукта.

Проблема усугубляется тем, что внешние зависимости обычно доступны в разных версиях и модификациях под разными операционными системами. Например, если два разработчика используют совершенно разные дистрибутивы Linux (допустим, Debian и Gentoo), то вряд ли в каждый момент времени им будет доступна одна и та же версия nginx в репозиториях пакетов ОС. Более того, разные версии могут компилироваться с разными флагами времени компиляции (например, чтобы активировать специфические настройки для той или иной ОС), а также поставляться с особыми расширениями или исправлениями для конкретного дистрибутива.

Таким образом, без подходящих инструментов было бы трудно гарантировать, что все участники команды разработки используют одинаковые версии всех компонентов. Теоретически можно добиться, чтобы все, кто работает над одним и тем же проектом, использовали одинаковые версии служб на своих компьютерах. Однако все усилия окажутся напрасными, если они не работают под той же операционной системой, которая будет в среде эксплуатации. Не всегда можно заставить программиста работать в чем-то, кроме его любимой операционной системы.

Среда эксплуатации (production environment)[1] — реальная среда, в которой приложение устанавливается, чтобы работать по прямому назначению. Например, среда эксплуатации для настольного приложения — это настольный компьютер, на котором пользователь установил приложение. Для серверной части веб-приложения, доступного через интернет, средой эксплуатации обычно является удаленный сервер (иногда виртуальный), установленный в центре обработки данных.

[1] Иногда говорят просто «продакшен». — *Примеч. ред.*

Беда в том, что переносимость все еще остается большой проблемой. Не все службы функционируют в среде эксплуатации точно так же, как на компьютерах разработчиков, и эта ситуация вряд ли изменится. Даже Python может по-разному вести себя в разных системах, несмотря на то, сколько усилий было потрачено, чтобы сделать его кросс-платформенным. Для Python такие расхождения обычно хорошо документированы и встречаются только в точках прямого взаимодействия с ОС. Тем не менее это не очень надежная стратегия — полагаться на то, что программист сможет запомнить длинный список особенностей совместимости.

Распространенное решение этой проблемы — изолировать целые системы в качестве сред приложения. Обычно это делается с помощью разных средств виртуализации систем. Конечно, виртуализация может влиять на производительность, однако с современными процессорами, которые поддерживают виртуализацию на аппаратном уровне, потери производительности существенно снижаются. Зато список возможных преимуществ оказывается довольно длинным:

- Версии системы, служб и совместно используемых библиотек в среде разработки можно точно синхронизировать со средой эксплуатации, что помогает бороться с проблемами совместимости.
- Одни и те же определения для инструментов конфигурации системы, таких как Puppet, Chef или Ansible (если они используются), можно использовать при настройке как среды эксплуатации, так и среды разработки.
- Разработчики, недавно присоединившиеся к команде, могут легко подключиться к проекту, если создание таких сред автоматизировано.
- Разработчики могут напрямую оперировать низкоуровневыми системными средствами, которые недоступны в их привычной операционной системе. Например, **FUSE** (Filesystem in Userspace) — это возможность операционной системы Linux, которая не будет работать в Windows без виртуализации.

В следующем разделе мы рассмотрим два разных подхода к изоляции сред разработки на уровне системы.

Контейнеризация и виртуализация

Существуют два основных типа изоляции на уровне системы для целей разработки:

- **Полная виртуализация**, при которой эмулируется вся компьютерная система.
- **Виртуализация на уровне операционной системы**, также называемая **контейнеризацией**, при которой изолируются целые пользовательские пространства в пределах одной ОС.

Средства полной виртуализации эмулируют целые компьютерные системы внутри других систем. Их можно рассматривать как виртуальное оборудование, которое предоставляется с помощью программы, выполняющейся на вашем компьютере. Поскольку при этом полностью эмулируется аппаратное обеспечение, появляется возможность запустить на вашем хосте любую операционную систему. Эта технология заложена в основу инфраструктуры **VPS** (Virtual Private Server) и провайдеров облачных вычислений, так как она позволяет запускать несколько независимых и изолированных друг от друга операционных систем на одном физическом компьютере.

Также этот метод подходит, чтобы запускать разные операционные системы в целях разработки, потому что запуск новой ОС не требует перезагрузки компьютера. При этом можно просто избавиться от виртуальной машины, если она больше не нужна. В типичной мультизагрузочной конфигурации выполнить эти операции с такой же легкостью не получится.

Наоборот, виртуализация на уровне операционной системы не эмулирует оборудование. Она инкапсулирует среду пользовательского пространства (совместно используемые библиотеки, ограничения ресурсов, тома файловой системы, код и т. д.) в форме контейнеров, которые работают только в пределах четко ограниченной контейнерной среды. Все контейнеры выполняются на одном и том же ядре ОС, но не вмешиваются в работу друг друга, если только вы явно не разрешите им это делать.

Виртуализация на уровне операционной системы не подразумевает эмуляции оборудования. Тем не менее она может устанавливать ограничения на использование системных ресурсов: дискового пространства, процессорного времени, оперативной памяти или сети. Этими ограничениями управляет только системное ядро, так что дополнительные затраты производительности обычно меньше, чем при полной виртуализации. Поэтому виртуализация на уровне ОС часто называется **облегченной виртуализацией**.

В контейнере обычно содержится только код приложения и его зависимости системного уровня, прежде всего совместно используемые библиотеки или двоичные файлы времени выполнения (такие, как интерпретатор Python), хотя теоретически контейнеры могут быть сколь угодно большими. Образы контейнеров Linux часто базируются на дистрибутивах целых систем, таких как Debian, Ubuntu или Fedora. Для процессов, выполняющихся внутри контейнера, он выглядит как полностью изолированная системная среда.

Если речь идет об изоляции на уровне системы для целей разработки, оба способа обеспечивают сходные уровни изоляции и воспроизводимости. Тем не менее разработчики чаще предпочитают «облегченную» виртуализацию на уровне ОС, потому что среды такого рода обычно оказываются экономичнее, быстрее, практичнее, а также удобнее с точки зрения сборки и портируемости.

Это особенно полезно для программистов, которые работают над несколькими проектами параллельно или должны синхронизировать свою среду с другими программистами.

Вот два ведущих инструмента, которые обеспечивают изоляцию на уровне системы для сред разработки:

- Docker для виртуализации на уровне операционной системы.
- Vagrant для полной виртуализации.

В функциональности Docker и Vagrant много общего, но они предназначены для разных целей. Vagrant создавался в основном как инструмент для разработки. Он позволяет запустить целую виртуальную машину одной командой, но редко применяется, чтобы просто упаковать такую среду как завершенный продукт, который можно легко перенести в среду эксплуатации и выполнять «как есть». А вот Docker создавался именно для этого: чтобы готовить самодостаточные контейнеры, которые можно перенести и развернуть в рабочей среде как полноценные пакеты. При грамотной реализации это может значительно улучшить процесс развертывания продукта.

Из-за нюансов реализации среды, основанные на контейнерах, иногда ведут себя не так, как среды, основанные на виртуальных машинах. Кроме того, они не включают ядро операционной системы, из-за чего код, сильно привязанный к специфике ОС, на разных хостах может работать по-разному. А если вы решите применять контейнеры для разработки, но не использовать их в целевых средах эксплуатации, то частично лишитесь гарантий целостности, ради которых, собственно, устраивалась изоляция среды.

Но если контейнеры уже используются в целевой среде эксплуатации, вам всегда следует теми же средствами воспроизводить условия этой среды на стадии разработки. К счастью, в Docker, который сейчас считается самым популярным контейнерным решением, есть великолепный инструмент `docker-compose`, который значительно упрощает управление локальными контейнерными средами.

Контейнеры — отличная альтернатива полной виртуализации. Это облегченный метод виртуализации, в котором ядро и операционная система позволяют запускать несколько изолированных экземпляров пользовательских пространств. Если в вашей ОС есть встроенная поддержка контейнеров, то этот метод виртуализации будет менее ресурсоемким, чем полная виртуализация на уровне машины.

Виртуальные среды с Docker

Программные контейнеры во многом обязаны своей популярностью Docker — одной из доступных реализаций для операционной системы Linux.

Docker позволяет описать образ контейнера в форме простого текстового документа, который называется **Docker-файлом**. На основе таких файлов можно развертывать образы, которые затем сохраняются в репозиториях образов. Эти репозитории позволяют программистам повторно использовать существующие образы, а не создавать их самостоятельно. Docker также поддерживает инкрементные изменения, поэтому если в контейнер что-то добавляется, его не придется перестраивать с нуля.

Docker — метод виртуализации ОС для систем семейства Linux, а ядра Windows и macOS не поддерживают его напрямую. Однако это не значит, что Docker нельзя использовать в этих системах. В них Docker становится своего рода гибридом полной виртуализации и виртуализации на уровне ОС. Установка Docker в этих двух системах создает промежуточную виртуальную машину с ОС Linux, которая служит хостом для контейнеров. Демон Docker и средства командной строки позволяют прозрачно передавать весь трафик и образы между операционной системой и контейнерами, работающими на виртуальной машине.

Инструкции по установке Docker доступны по адресу https://www.docker.com/get-started.

Наличие промежуточной виртуальной машины означает, что в Windows и macOS Docker уже не будет таким легковесным, как в Linux. Тем не менее дополнительные затраты производительности не должны быть существенно выше, чем в случае других сред разработки, основанных исключительно на полной виртуализации.

Создание первого Docker-файла

Каждая среда на базе Docker опирается на Docker-файл — описание того, как создать образ Docker. Этот образ можно представлять себе примерно так же, как образ виртуальной машины: это отдельный файл с многоуровневой структурой, который инкапсулирует все системные библиотеки, файлы, исходный код и другие зависимости, которые нужны, чтобы запускать ваши приложения.

Каждый уровень образа Docker описывается в Docker-файле одной инструкцией такого формата:

```
ИНСТРУКЦИЯ аргументы
```

Docker поддерживает множество инструкций. Перечислим самые элементарные, которые понадобятся для начала работы:

- `FROM <имя_образа>`: описывает базовый образ, который послужит основой для вашего. Образы часто создаются на базе стандартных дистрибутивов Linux,

обычно с донолнительными библиотеками и программами. Стандартный репозиторий образов Docker называется Docker Hub и бесплатно доступен по адресу https://hub.docker.com/.

- `COPY <источник>... <приемник>`: копирует файлы из контекста локальной сборки (обычно файлы проекта) и добавляет их в файловую систему контейнера.
- `ADD <источник>... <приемник>`: работает аналогично `COPY`, но автоматически распаковывает архивы и позволяет использовать адреса URL в качестве источника.
- `RUN <команда>`: выполняет заданную команду поверх предыдущих уровней. После этого изменения, внесенные командой в файловую систему, фиксируются как новый уровень образа.
- `ENTRYPOINT ["<исполняемый_файл>", "<параметр>", ...]`: задает точку входа — команду по умолчанию, которая будет выполняться при запуске контейнера. Если точка входа не задана ни на одном уровне образа, Docker по умолчанию использует `/bin/sh -c`, то есть запускает стандартную командную оболочку (обычно Bash, но возможны и другие оболочки).
- `CMD ["<параметр>", ...]`: задает параметры по умолчанию для точек входа образа. С учетом того, что точка входа по умолчанию — `/bin/sh -c`, эта инструкция также может принимать форму `CMD ["<исполняемый_файл>", "<параметр>", ...]`. Рекомендуется задавать целевой исполняемый файл непосредственно в инструкции `ENTRYPOINT`, а `CMD` использовать только для аргументов по умолчанию.
- `WORKDIR <каталог>`: назначает текущий рабочий каталог для каждой из последующих инструкций — `RUN`, `CMD`, `ENTRYPOINT`, `COPY` и `ADD`.

Чтобы продемонстрировать типичную структуру Docker-файла, попробуем поместить в контейнер простое приложение на Python. Предположим, мы хотим создать эхо-сервер HTTP, который возвращает содержимое полученного запроса HTTP. Мы будем использовать Flask — популярный веб-микрофреймворк Python.

Flask не входит в стандартную библиотеку Python. Чтобы установить его в своей среде, используйте `pip`:

```
$ pip install flask
```

Дополнительную информацию о фреймворке Flask можно найти по адресу https://flask.palletsprojects.com/.

Код приложения, который будет сохранен в сценарии Python echo.py, может выглядеть так:

```
from flask import Flask, request
app = Flask(__name__)

@app.route('/')
def echo():
    return (
        f"METHOD: {request.method}\n"
        f"HEADERS:\n{request.headers}"
        f"BODY:\n{request.data.decode()}"
    )

if __name__ == '__main__':
    app.run(host="0.0.0.0")
```

В начале сценария импортируется класс `Flask` и объект `request`. Экземпляр класса `Flask` представляет наше веб-приложение. Специальный глобальный объект `request` всегда представляет контекст запроса HTTP, который обрабатывается в данный момент.

`echo()` — так называемая **функция представлеиия**, отвечающая за обработку входных запросов. `@app.route('/')` регистрирует фупкцию представления `echo()` с путем `/`. Это означает, что к этой функции будут перенаправляться только те запросы, которые соответствуют пути `/`. Внутри представления функция считывает компоненты входного запроса (метод, заголовки и тело) и возвращает их в текстовой форме. Flask включает этот текстовый вывод в тело ответа на запрос.

Сценарий завершается вызовом метода `app.run()`. Так запускается локальный сервер разработки пашего приложепия. Он не предназпачеп для использования в среде эксплуатации, но годится для целей разработки и сильно упрощает паш пример.

Если у вас установлеп пакет Flask, приложение можпо запустить такой командой:

```
$ python3 echo.py
```

Команда запускает сервер разработки Flask на порте 5000. Для доступа к серверу либо откройте адрес http://localhost:5000 в браузере, либо воспользуйтесь утилитой командной строки.

Вот пример ответа сервера для запроса GET с помощью `curl`:

```
$ curl localhost:5000
METHOD: GET
HEADERS:
Host: localhost:5000
User-Agent: curl/7.64.1
Accept: */*
BODY:
```

После того как мы убедились, что приложение возвращает содержимое полученного запроса HTTP, все почти готово к тому, чтобы поместить приложение в Docker. Файлы нашего проекта будут иметь такую структуру:

```
.
├── Dockerfile
├── echo.py
└── requirements.txt
```

Файл requirements.txt будет содержать только одну запись `flask==1.1.2`, чтобы гарантировать, что образ всегда будет использовать одну и ту же версию Flask. Прежде чем переходить к Docker-файлу, давайте решим, как должен работать образ. Мы хотим добиться следующих целей:

- Скрыть от пользователя техническую начинку, прежде всего тот факт, что мы используем Python и Flask.
- Упаковать исполняемый файл Python 3.9 со всеми зависимостями.
- Упаковать все зависимости проекта, определенные в файле requirements.txt.

Имея эти требования, можно приступать к написанию первого Docker-файла. Он выглядит так:

```
FROM python:3.9-slim
WORKDIR /app/

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY echo.py .
ENTRYPOINT ["python", "echo.py"]
```

Строка `FROM python:3.9-slim` задает базовый образ для нашего контейнерного образа. Это один из официальных образов, коллекцию которых Python поддерживает на Docker Hub. `3.9-slim` — тег образа, включающего Python 3.9 с минимальным набором системных пакетов, необходимых для запуска Python. Обычно это неплохая отправная точка для образов приложений на основе Python.

В следующем разделе вы узнаете, как создать образ Docker на основе приведенного выше Docker-файла и как запустить контейнер.

Запуск контейнеров

Прежде чем можно будет запустить контейнер, надо создать образ, определенный в Docker-файле. Это можно сделать такой командой:

```
$ docker build -t <имя> <путь>
```

Аргумент `-t <имя>` позволяет нрисвоить образу удобочитаемый идентификатор. Это совершенно необязательно, однако без идентификатора не получится легко ссылаться на вновь созданный образ. Аргумент `<путь>` задает нуть к каталогу, где находится Docker-файл. Допустим, мы запускаем команду из корневого каталога проекта, представленного в предыдущем разделе. При этом мы хотим назначить образу идентификатор `echo`. Команда `docker build` будет выглядеть так:

```
$ docker build -t echo .
```

Результат может получиться следующим:

```
Sending build context to Docker daemon   16.8MB
Step 1/6 : FROM python:3.9-slim
3.9-slim: Pulling from library/python
bb79b6b2107f: Pull complete
35e30c3f3e2b: Pull complete
b13c2c0e2577: Pull complete
263be93302fa: Pull complete
30e7021a7001: Pull complete
Digest: sha256:c13fda093489a1b699ee84240df4f5d0880112b9e09ac21c5d687500
3d1aa927
Status: Downloaded newer image for python:3.9-slim
 ---> a90139e6bc2f
Step 2/6 : WORKDIR /app/
 ---> Running in fd85d9ac44a6
Removing intermediate container fd85d9ac44a6
 ---> b781318cdec7
Step 3/6 : COPY requirements.txt .
 ---> 6d56980fedf6
Step 4/6 : RUN pip install -r requirements.txt
 ---> Running in 5cd9b86ac454
(...)
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1
click-7.1.2 flask-1.1.2 itsdangerous-1.1.0
Removing intermediate container 5cd9b86ac454
 ---> 0fbf85e8f6da
Step 5/6 : COPY echo.py .
 ---> a546d22e8c98
Step 6/6 : ENTRYPOINT [„python", „echo.py"]
 ---> Running in 0b4e57680ac4
Removing intermediate container 0b4e57680ac4
 ---> 0549d15959ef
Successfully built 0549d15959ef
Successfully tagged echo:latest
```

Список доступных образов можно просмотреть командой `docker images`:

```
$ docker images
REPOSITORY      TAG         IMAGE ID        CREATED              SIZE
echo            latest      0549d15959ef    About a minute ago   126MB
python          3.9-slim    a90139e6bc2f    10 days ago          115MB
```

ШОКИРУЮЩИЙ РАЗМЕР ОБРАЗОВ КОНТЕЙНЕРОВ

Размер нашего образа составляет 126 Мбайт, потому что в нем фактически заключен целый дистрибутив Linux, который необходим, чтобы выполнять приложение на Python. Казалось бы, это очень много, но на самом деле не о чем беспокоиться. Для краткости мы использовали базовый образ, с которым просто работать. Существуют и другие образы, созданные специально для минимизации размера, но они обычно предназначены для более опытных пользователей Docker. А если у вас много контейнеров, то благодаря многоуровневой структуре образов Docker базовые уровни могут кэшироваться и использоваться повторно, так что редко возникают проблемы с затратами дискового пространства. В приведенном примере оба образа вместе займут лишь 126 Мбайт, потому что образ `echo:latest` добавляет всего 11 Мбайт к образу python:3.9-slim.

После того как образ создан и помечен идентификатором, контейнер можно запустить командой `docker run`. Наш контейпер — это пример веб-службы, поэтому придется дополпительно сообщить Docker, что мы хотим афишировать порты контейнера, для чего выполнить их локальную привязку:

```
docker run -it --rm --publish 5000:5000 echo
```

Вот что делают аргументы этой команды:

- `-it`: на самом деле это два объединенных флага: `-i` и `-t`. Флаг `-i` (от слова interactive) сохраняет поток stdin открытым, даже если процесс контейнера отсоединяется, а `-t` (от аббревиатуры tty) выделяет контейнеру псевдотелетайп. Телетайп (TTY) в Linux и операционных системах семейства UNIX — это абстракция, которая связывает стандартный ввод и вывод нрограммы с некоторым терминалом. Короче говоря, благодаря этим двум флагам вы сможете в реальном времени видеть вывод ваших приложений и завершать процесс нрерыванием с клавиатуры. Процесс просто будет вести себя так же, как если бы вы занустили Python прямо из командной строки.
- `--rm`: заставляет Docker автоматически удалять контейнер при завершении. Без этого флага контейнер остается, и к нему можно повторно присоединиться, чтобы диагностировать его состояние. По умолчанию Docker не удаляет контейнеры, потому что это упрощает отладку. Они способны быстро загромождать диск, поэтому стоит привыкнуть использовать `--rm` по умолчанию, если только вам не требуется сохранить завершенный контейнер для последующего анализа.
- `--publish 5000:5000`: заставляет Docker афишировать норт контейнера 5000, вынолнив нривязку порта 5000 к интерфейсу хоста. Этот нараметр также можно использовать, чтобы переназначить порты приложения. Например,

если вы хотите предоставить локальный доступ к приложению `echo` на порте 8080, используйте аргумент `--publish 8080:5000`.

Создавать и запускать собственные образы командой `docker` — простые и прямолинейные операции, но через какое-то время они начинают казаться слишком громоздкими: приходится вводить довольно длинные команды и запоминать многочисленные идентификаторы. В более сложных средах это бывает неудобно. В следующем разделе мы посмотрим, как упростить рабочие процессы Docker с помощью программы Docker Compose.

Настройка сложных сред

В простых конфигурациях основные операции с Docker выполняются достаточно тривиально, однако когда вы начинаете использовать Docker в нескольких проектах, возрастает риск путаницы. Очень легко забыть о конкретных параметрах командной строки или о том, какие порты должны афишироваться для тех или иных образов.

Но ситуация действительно усложняется, когда одна из ваших служб должна взаимодействовать с другими. Дело в том, что в каждый контейнер Docker рекомендуется помещать только один работающий процесс.

Это значит, что в образ контейнера лучше не добавлять инструменты наблюдения за процессами (такие, как Supervisor или Circus). Вместо этого стоит создать несколько контейнеров, взаимодействующих друг с другом. Каждая служба может использовать свой собственный образ, работать с собственными параметрами конфигурации и предоставлять порты, которые могут перекрываться с портами других служб. Если вы хотите запускать несколько разных процессов, то каждый процесс должен быть в отдельном контейнере.

В больших средах эксплуатации со множеством развернутых контейнеров используются специализированные системы управления контейнерами (такие, как Kubernetes, Nomad или Docker Swarm), которые отслеживают все контейнеры и подробности их выполнения: образы, порты, тома, конфигурацию и т. д. Такие инструменты можно использовать и локально, но для целей разработки это излишне.

Лучший инструмент для управления контейнерами при разработке на локальном компьютере, который подходит как для простых, так и для сложных сценариев, — Docker Compose. Он обычно распространяется вместе с Docker, но в некоторых дистрибутивах Linux (например, Ubuntu) он недоступен по умолчанию. В таких случаях его нужно установить как отдельный пакет из репозитория системных пакетов. Docker Compose предоставляет мощную программу командной строки `docker-compose` и позволяет создавать описания многоконтейнерных приложений в синтаксисе YAML.

Compose предполагает, что в корневом каталоге проекта находится файл со специальным именем docker-compose.yml. В предыдущем проекте такой файл мог бы выглядеть так:

```
version: '3.8'

services:
  echo-server:
    # Заставляет Docker Compose построить образ
    # на базе локального каталога (.)
    build: .

    # Эквивалент параметра "-p" команды "docker run"
    ports:
    - "5000:5000"

    # Эквивалент параметра "-t" команды "docker run"
    tty: true
```

Если создать такой файл docker-compose.yml в вашем проекте, то всю среду приложения можно будет запускать и останавливать двумя простыми командами:

- `docker-compose up`: запускает все контейнеры, определенные в файле docker-compose.yml, и активно выводит их стандартный вывод.
- `docker-compose down`: останавливает все контейнеры, запущенные командой `docker-compose` в текущем каталоге проекта.

Docker Compose автоматически создает образ, если его не было раньше. Это отличный способ описать среду разработки в файле конфигурации. Если вы работаете с другими программистами, им можно передать один файл docker-compose.yml для вашего проекта. В результате развертывание полностью работоспособной локальной среды разработки сводится к одной команде `docker-compose up`. Если вы пользуетесь системой управления версиями, в нее определенно стоит включить файл docker-compose.yml наряду с остальным кодом.

Более того, если вашему приложению требуются дополнительные внешние службы, то их можно легко добавить в среду Docker Compose, вместо того чтобы устанавливать в хост-системе. Вот пример, в котором добавляется один экземпляр базы данных PostgreSQL и хранилище Redis с использованием официальных образов из Docker Hub:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
```

```
    - "5000:5000"
    tty: true

  database:
    image: postgres

  cache:
    image: redis
```

Docker Hub — официальный репозиторий образов Docker. Многие разработчики ПО с открытым кодом размещают здесь официальные образы своих проектов. Дополнительную информацию о Docker Hub можно найти по адресу https://hub.docker.com.

Все очень просто. Чтобы обеспечить лучшую воспроизводимость среды, всегда следует указывать теги версий внешних образов (например, `postgres:13.1` и `redis:6.0.9`). Тем самым вы гарантируете, что все, кто пользуется вашим файлом docker-compose.yml, будут применять одни и те же версии внешних служб. Благодаря Docker Compose можно одновременно использовать несколько версий одной службы: они не будут мешать друг другу, потому что разные среды Docker Compose по умолчанию изолируются на сетевом уровне.

Полезные рецепты Docker и Docker Compose для Python

Docker и контейнеры вообще — настолько необъятная тема, что ее невозможно охватить в одном коротком разделе книги. Благодаря Docker Compose можно легко начать работать с Docker, не погружаясь во внутреннее устройство этой системы. Однако если у вас еще нет опыта работы с Docker, то рано или поздно вам придется сделать паузу, взять документацию и тщательно прочитать ее.

Официальная документация Docker доступна по адресу https://docs.docker.com/.

Ниже приведены несколько популярных советов и рецептов, которые позволят отсрочить этот момент и решить многие распространенные проблемы, с которыми рано или поздно придется иметь дело.

Как уменьшить размер контейнеров

Общая проблема начинающих пользователей Docker — большой размер образов контейнеров. Контейнеры и правда занимают гораздо больше места, чем обычные пакеты Python, хотя их размеры, как правило, незначительны по сравнению с образами виртуальных машин. Тем не менее на одной виртуальной машине часто размещают много служб, а в случае контейнеров определенно стоит соз-

давать отдельный образ для каждой службы. Это означает, что при большом количестве служб издержки дискового нространства могут стать заметными.

Чтобы ограничить размер образов, применяйте два взаимодополняющих приема:

- **Исиользуйте базовый образ, создаииый сиециальио для этой цели**. Alpine Linux — пример компактпого дистрибутива Linux, специально адаптированного для создания очень пебольших и предельно облегчепных образов Docker. Размер базового образа — всего около 5 Мбайт, а элегантный менеджер пакетов поможет обеспечить комнактность образов.
- **Учитывайте особенности стековой файловой системы Docker**. Образы Docker состоят из уровней, каждый из которых инкапсулирует различия в корневой файловой системе между собой и предыдущим уровпем. После того как уровень зафиксирован, размер образа уже нельзя умепьшить. Это означает, что если системный пакет иснользуется как зависимость на этапе сборки и позднее может быть исключен из образа, то вместо множественных ипструкций `RUN` может иметь смысл сделать все в одной ипструкции `RUN` с цепочкой команд оболочки, чтобы предотвратить избыточную фиксацию уровней.

Эти два нриема продемонстрированы в следующем Docker-файле:

```
FROM alpine:3.13
WORKDIR /app/

RUN apk add --no-cache python3

COPY requirements.txt .
RUN apk add --no-cache py3-pip && \
    pip3 install -r requirements.txt && \
    apk del py3-pip

COPY echo.py .
CMD ["python", "echo.py"]
```

В этом примере использован базовый образ alpine:3.13, чтобы продемонстрировать, как избавляться от ненужных зависимостей перед фиксацией уровня. К сожалению, менеджер apk в дистрибутиве Alpine не позволяет точно контролировать, какая версия Python будет установлена. Поэтому рекомендуемые базовые образы Alpine берутся из официального репозитория Python. Для Python 3.9 это базовый образ python:3.9-alpine.

Флаг `--no-cache` программы `apk` (менеджера пакетов для Alpine) имеет два эффекта. Во-первых, он заставляет `apk` игнорировать существующий кэш списков пакетов, чтобы была установлена последняя версия пакета, официально доступпая в репозитории. Во-вторых, он не обновляет существующий кэш списков

пакетов, а значит, уровень, созданный этой инструкцией, будет чуть меньше, чем при использовании флага `--update-cache`, который был бы необходим, чтобы установить последнюю версию пакета. Различия не столь заметны (вероятно, около 2 Мбайт), но такие маленькие фрагменты кэша могут накапливаться в больших образах с вызовами `apk add` на многих уровнях. В других дистрибутивах Linux менеджеры пакетов обычно предоставляют похожие инструменты, чтобы отключить кэш списков пакетов.

Вторая инструкция `RUN` в этом примере учитывает устройство уровней в образах Docker. В Alpine пакет Python не включает установленную версию `pip`, поэтому ее приходится устанавливать вручную. Как правило, после установки всех необходимых пакетов Python `pip` становится ненужным и его можно удалить. Чтобы инициализировать `pip`, можно было воспользоваться модулем `ensurepip`, но тогда не будет очевидного способа удалить `pip`. Вместо этого мы построили длинную цепочку инструкций, которая использует `apk`, чтобы установить пакет `py3-pip` и удалить его после установки других пакетов Python. В Alpine 3.13 этот прием может сэкономить до 16 Мбайт.

Выполнив команду `docker images`, вы увидите, что размеры образов, основанных на базовых образах Alpine и python:slim, существенно различаются:

```
$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED              SIZE
echo-alpine   latest    e7e3a2bc7b71   About a minute ago   53.7MB
echo          latest    6b036d212e8f   40 minutes ago       126MB
```

Новый образ почти вдвое меньше того, который основан на образе `python:3.9-slim`. В основном это объясняется компактностью дистрибутива Alpine, размер которого составляет всего около 5 Мбайт. Если бы мы не удалили `pip` и не использовали флаг `--no-cache`, то размер образа, вероятно, был бы близок к 72 Мбайт (кэши списков пакетов занимают около 2 Мбайт, а `py3-pip` — около 16 Мбайт). В сумме это позволяет сэкономить почти 25 % размера образа. Эта экономия будет не столь ощутима для больших приложений со множеством зависимостей, где 18 Мбайт не играют особой роли. Тем не менее этот прием полезен и для работы с другими зависимостями на этапе сборки. Например, некоторым пакетам во время установки нужны дополнительные компиляторы, такие как `gcc` (GNU Compiler Collection), и дополнительные заголовочные файлы. В такой ситуации можно действовать по тому же принципу, чтобы в итоговом образе не было полного набора GNU Compiler Collection. Это уже может заметно повлиять на размер образа.

Как работать со службами в среде Docker Compose

Сложные приложения часто состоят из многочисленных служб, взаимодействующих друг с другом. Compose позволяет легко описывать такие приложения.

Ниже нриведен пример файла docker-compose.yml, в котором приложение онределено как комбинация двух служб:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true

  database:
    image: postgres
    restart: always
```

В этой конфигурации указаны две службы:

- `echo-server`: контейнер службы эхо-вывода с образом, созданным на основе локального Docker-файла.
- `database`: контейнер базы данных PostgreSQL из официального Docker-образа `postgres`.

Будем считать, что `echo-server` собирается взаимодействовать со службой `database` по сети. Чтобы настроить такое взаимодействие, нужно знать IP-адрес или имя хоста службы: эти сведения используются в конфигурации приложения. К счастью, Docker Compose создавался именно для таких сценариев, поэтому он существенпо упрощает задачу.

Каждый раз, когда вы запускаете среду командой `docker-compose up`, Docker Compose по умолчанию создает выделенную сеть Docker и регистрирует все службы в этой сети, используя их имена как имена хостов. Это означает, что служба `echo-server` может использовать адрес `database:5432`, чтобы взаимодействовать с `database` (5432 — порт PostgreSQL по умолчанию), а любая другая служба в среде Docker Compose сможет обратиться к конечной точке HTTP службы `echo-server` по адресу http://echo-server:80.

Хотя имена хостов служб в Docker Compose легко предсказуемы, не рекомендуется жестко фиксировать адреса в коде приложепия. Лучше объявить их как переменные окружения, которые приложение может считывать при запуске. Следующий пример показывает, как определить произвольные переменные окружения для каждой службы в файле docker-compose.yml:

```
version: '3.8'

services:
  echo-server:
    build: .
```

```
    ports:
    - "5000:5000"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432
      - DATABASE_PASSWORD=password

  database:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: password
```

В выделенных строках объявляются переменные окружения, которые сообщают эхо-серверу имя хоста и порт базы данных. Переменные окружения — самый нредпочтительный сиособ передавать контейнерам параметры конфигурации.

Контейнеры Docker эфемерны. Это означает, что когда контейнер удаляется (обычно при выходе), внутренние изменения его файловой системы теряются. Для баз данных это означает, что если вы не хотите потерять информацию в БД, которая запущена в контейнере, то следует смонтировать том внутри контейнера в каталоге, где должны храниться данные. Специалисты по сопровождению образов Docker для баз данных обычно обеспечивают документацию по монтированию таких томов, поэтому чтобы защитить информацию в базе данных, всегда обращайтесь к документации используемого образа Docker. Пример того, как тома Docker применяются для несколько иных целей, приведен в разделе «Как добавить оперативную перезагрузку для абсолютно любого кода».

Взаимадействие между средами Dacker Campose

Если вы создаете систему, состоящую из нескольких независимых служб и/или нриложений, то вам с большой долей вероятности понадобится хранить их код в нескольких отдельных репозиториях кода (проектах). Файлы docker-compose.yml для каждого приложепия Docker Compose обычно хранятся в том же реноситории, что и код приложения. Сеть по умолчанию, которую Compose создает для приложения, изолирована от сетей других нриложений. Что же делать, если вам внезапно попадобилось, чтобы несколько независимых приложений взаимодействовали друг с другом?

К счастью, эта задача тоже чрезвычайно легко решается в Compose. Синтаксис файла docker-compose.yml позволяет задать именованную внешнюю сеть Docker как сеть по умолчанию для всех служб, определенных в этой конфигурации.

В этом примере конфигурации задается внешняя сеть с именем `my-interservice-network`:

```
version: '3.8'

networks:
  default:
    external:
      name: my-interservice-network

services:
  webserver:
    build: .
    ports:
    - "80:80"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432
      - DATABASE_PASSWORD=password

  database:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: password
```

Такие внешние сети не находятся под управлением Docker Compose, поэтому их придется создавать вручную командой `docker network create`:

```
$ docker network create my-interservice-network
```

После этого внешнюю сеть можно использовать в других файлах docker-compose.yml для всех приложений, чьи службы должны быть зарегистрированы в той же сети. Ниже приведен пример конфигурации других приложений, которые смогут взаимодействовать со службами `database` и `webserver` по сети `my-interservice-network`, несмотря на то что они не определены в том же файле docker-compose.yml:

```
version: '3.8'

networks:
  default:
    external:
      name: my-interservice-network

services:
  other-service:
    build: .
    ports:
    - "80:80"
```

```
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432
      - ECHO_SERVER_ADDRESS=http://echo-server:80
```

Такой подход позволяет запустить две независимые среды Docker Compose в разных оболочках. Все службы смогут взаимодействовать друг с другом через общую сеть Docker.

Как отложить запуск кода до открытия портов служб

По команде `docker-compose up` все службы запускаются одновременно. Последовательность запуска можно до определенной степени регулировать с помощью ключа `depends_on` в объявлении службы, как в этом примере:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    depends_on:
      - database

  database:
    image: postgres
    environment:
      POSTGRES_PASSWORD: password
```

Эта конфигурация гарантирует, что эхо-сервер будет запущен после службы базы данных. К сожалению, не всегда можно обойтись тем, чтобы просто задать нужный порядок запуска служб в среде разработки.

Предположим, что служба `echo-server` должна прочитать что-то из базы данных непосредственно после запуска. Docker Compose позаботится о том, чтобы службы запускались по порядку, но не гарантирует, что PostgreSQL будет готов принимать подключения от эхо-сервера. Дело в том, что инициализация PostgreSQL может занять пару секунд.

Проблема решается достаточно просто. Существует много сценарных утилит, которые позволяют проверить, открыт ли конкретный сетевой порт, прежде чем продолжать выполнение команды. Одна из таких утилит называется `wait-for-it` и написана на Python, что позволяет легко установить ее из `pip`.

Сценарий `wait-for-it` можно запускать таким образом:

```
$ wait-for-it --service <адрес_службы> -- command [...]
```

Конструкция `-- command [...]` — это стандартный шаблон для программ, которые служат оберткой для других команд. При этом `[...]` обозначает произвольный набор аргументов команды. Процесс `wait-for-it` пытается создать подключение TCP и, когда это получится, выполняет `command [...]`. Например, если нужно дождаться подключения к локальному хосту на порте 2000, прежде чем выполнять команду `python echo.py`, то для этого достаточно запустить такую команду:

```
$ wait-for-it --service localhost:2000 -- python echo.py
```

Ниже приведен пример измененного файла docker-compose.yml, который элегантно переопределяет команду `docker image` и декорирует ее вызовом утилиты `wait-for-it`. Тем самым гарантируется, что эхо-сервер будет запущен только тогда, когда он сможет подключиться к базе данных:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    depends_on:
      - database
    command:
      wait-for-it --service database:5432 --
      python echo.py

  database:
    image: postgres
    environment:
      POSTGRES_PASSWORD: password
```

По умолчанию время ожидания `wait-for-it` истекает через 15 секунд. По окончании этого времени процесс после метки `--` будет запущен независимо от того, удалось ли подключиться. Время ожидания можно отключить аргументом `--timeout 0` — тогда `wait-for-it` будет ожидать бесконечно.

Как добовить оперативную перезагрузку для абсалютно любого кода

Разрабатывая новое приложение, мы обычно работаем пад кодом в итеративном режиме: вносим изменения и смотрим па результаты. Мы либо проверяем код вручную, либо запускаем тесты. Работает цикл с постоянной обратной связью.

Чтобы код работал с Docker, его нужно заключить в образ контейнера. Но выполнять команду `docker build` или `docker-compose build` каждый раз, когда вы вносите изменения на хосте, будет крайне неэффективно.

Вместо этого лучше использовать тома Docker, чтобы передавать код контейнеру на стадии разработки. Здесь основная идея в том, чтобы связать каталог локальной файловой системы с путем внутренней файловой системы контейнера. При этом любые изменения в файловой системе хоста будут автоматически отражаться внутри контейнера. С Docker Compose это делается чрезвычайно легко, потому что программа позволяет определять тома в конфигурации служб. Ниже приведена измененная версия файла для службы `echo`, где корневой каталог проекта монтируется на пути `/app/`:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    volumes:
      - .:/app/
```

Изменения на смонтированных томах Docker активно распространяются на обеих сторонах. Многие серверы и фреймворки Python поддерживают активную горячую перезагрузку (hot reloading) каждый раз, когда обнаруживают изменения в коде. Это делает разработку гораздо более удобной, потому что изменения в работе приложения видны в процессе написания кода и не требуется ручной перезапуск.

Вероятно, не каждый фрагмент кода, который вы пишете, будет создаваться с помощью фреймворков, поддерживающих активную перезагрузку. К счастью, существует отличный пакет Python `watchdog`, который позволяет перезагружать любое приложение при внесении изменений в код. Этот пакет содержит удобную утилиту `watchmedo`, которая может инкапсулировать выполнение любого процесса по аналогии с `wait-for-it`.

Утилита `watchmedo` из пакета `watchdog` требует дополнительных зависимостей. Чтобы применить их при установке этого пакета, используйте такой синтаксис pip install:

```
pip install watchdog[watchmedo]
```

Следующая команда демонстрирует формат, с помощью которого можно перезагружать заданные процессы, когда изменяются любые файлы Python в текущем рабочем каталоге:

```
$ watchmedo auto-restart --patterns "*.py" --recursive -- command [...]
```

Параметр `--patterns "*.py"` указывает, какие файлы процесс `watchmedo` будет отслеживать на предмет изменений. Флаг `--recursive` вызывает рекурсивный обход текущего рабочего каталога, чтобы обнаруживать изменения даже на нижних уровнях дерева каталогов. Параметр `-- command [...]` используется так же, как команда `wait-for-it`, упомянутая в разделе «Как отложить запуск кода до открытия портов служб». Он просто означает, что весь текст после `--` рассматривается как одна команда (с необязательными аргументами). `watchmedo` запускает эту команду и перезапускает ее, если обнаружены изменения в отслеживаемых файлах.

Если вы установили пакет `watchdog` в своем образе Docker, его можно элегантно включить в файл docker-compose.yml, как в следующем примере:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    depends_on:
      - database
    command:
      watchmedo auto-restart --patterns "*.py" --recursive --
      python echo.py
    volumes:
      - .:/app/
```

Эта конфигурация Docker Compose перезапускает процесс внутри контейнера каждый раз, когда обнаружены изменения в коде Python. В нашем примере отслеживаются все файлы с расширением .py в пути /app/. Поскольку исходный каталог смонтирован как том Docker, утилита `watchmedo` актуализирует любые изменения, внесенные в файловой системе хоста, и выполняет перезапуск сразу же после того, как вы сохраняете изменения в редакторе.

Среды разработки с Docker и Docker Compose в высшей степени полезны и удобны, но у них есть ряд ограничений. Самое очевидное из них заключается в том, что они позволяют выполнять код только в операционной системе Linux. Хотя Docker доступен для macOS и Windows, в этих ОС он использует виртуальную машину Linux как промежуточный уровень, так что контейнеры Docker все равно будут выполняться под Linux. Если во время разработки вам нужно, чтобы приложение выполнялось в точности так же, как в некоторой системе, отличной от Linux, то понадобится совершенно иной подход к изоляции приложения. В следующем разделе рассматривается один из инструментов для этой задачи.

Виртуальные среды разработки на базе Vagrant

Хотя Docker в сочетании с Docker Compose обеспечивает хорошую основу для создания воспроизводимых и изолированных сред разработки, существуют ситуации, когда настоящая виртуальная машина попросту оказывается лучшим (или единственным) вариантом. Один из примеров — системное программирование для операционных систем, отличных от Linux.

Vagrant сейчас считается одним из самых популярных инструментов управления виртуальными машинами для целей локальной разработки. Vagrant позволяет просто и удобно описывать среды разработки со всеми системными зависимостями так, чтобы это описание было напрямую связано с исходным кодом проекта. Система доступна для Windows, macOS и нескольких популярных дистрибутивов Linux (см. https://www.vagrantup.com).

У Vagrant нет внешних зависимостей. Новые среды разработки создаются в виде виртуальных машин или контейнеров, конкретная реализация которых зависит от выбранного поставщика виртуализации. По умолчанию используется VirtualBox, включенный в установку Vagrant, но доступны и другие поставщики. Наиболее распространенные из них — VMware, Docker, Linux Containers (LXC) и Hyper-V.

Самые важные параметры конфигурации передаются Vagrant в одном файле с именем Vagrantfile. Для каждого проекта создается независимый экземпляр этого файла. Основные параметры в нем таковы:

- Поставщик виртуализации.
- Образ виртуальной машины.
- Метод оснащения виртуальной машины.
- Совместное использование дискового пространства виртуальной машиной и хостом.
- Порты для переадресации между виртуальной машиной и хостом.

Код Vagrantfile записывается на языке Ruby. Пример файла конфигурации служит хорошим шаблоном для начальной настройки проекта и отлично документирован, так что знать Ruby не обязательно. Стандартную конфигурацию можно создать всего одной командой:

```
$ vagrant init
```

Эта команда создает в текущем каталоге новый файл с именем Vagrantfile. Как правило, его лучше всего хранить в корневом каталоге исходного кода проекта. Этот файл уже содержит рабочую конфигурацию, которая создает новую вир-

туальную машину с поставщиком VirtualBox и образом на основе дистрибутива Ubuntu Linux. Стандартный файл **Vagrantfile**, созданный командой **vagrant init**, содержит множество комментариев, которые помогут вам окончательно настроить конфигурацию.

Ниже приведен пример простейшего файла **Vagrantfile** для среды разработки Python 3.9 на базе ОС Ubuntu. Файл содержит несколько стандартных настроек, включая переадресацию порта 80 на случай, если вы будете заниматься вебпрограммированием на Python:

```
Vagrant.configure("2") do |config|
  # Каждой среде разработки Vagrant необходим образ.
  # Образы можно найти по адресу https://vagrantcloud.com/search.
  # Здесь используется версия Bionic Ubuntu для архитектуры x64.
  config.vm.box = "ubuntu/bionic64"

  # Настроить переадресацию портов так,
  # чтобы конкретные порты внутри машины были доступны с порта на хосте;
  # разрешить доступ только с 127.0.0.1, чтобы блокировать внешний доступ.
  config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip:
"127.0.0.1"

  config.vm.provider "virtualbox" do |vb|
    vb.gui = false
    # Задать объем памяти для виртуальной машины:
    vb.memory = "1024"
  end

  # Разрешить оснащение сценарием командой оболочки.
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install python3.9 -y
  SHELL
end
```

В этом примере в разделе `config.vm.provision` настраивается дополнительное оснащение системными пакетами с помощью простого сценария командной оболочки. Образ виртуальной машины `ubuntu/bionic64` по умолчанию не включает версию Python 3.9, поэтому ее нужно установить с помощью менеджера пакетов `apt-get`.

Когда файл **Vagrantfile** будет готов, запустите виртуальную машину такой командой:

```
$ vagrant up
```

Первый запуск может занять несколько минут, потому что образ загружается по сети. При запуске существующей виртуальной машины могут выполняться и другие процессы инициализации, которые требуют некоторого времени в за-

висимости от поставщика виртуализации, образа и производительности вашей системы. После того как образ загружен, дальнейшие операции обычно занимают лишь несколько секунд. Когда среда Vagrant заработает, к ней можно подключиться через SSH с помощью следующей команды:

```
$ vagrant ssh
```

Эту команду можно занустить из любого места дерева исходного кода проекта ниже каталога с файлом Vagrantfile. Для удобства разработчиков Vagrant обходит все каталоги над текущим рабочим каталогом пользователя в дереве файловой системы, ищет файл конфигурации и соноставляет его с экземпляром виртуальной машипы. Затем Vagrant подключается к командной оболочке по SSH, чтобы со средой разработки можно было взаимодействовать как с обычной удалепной машиной. Единственное различие состоит в том, что все дерево исходного кода проекта (корнем которого считается местонахождение Vagrantfile) доступно в файловой системе виртуальной машины в каталоге /vagrant/. Он автоматически синхронизируется с файловой системой хоста, так что вы можете использовать свой любимый редактор кода или IDE на хосте и рассматривать сеанс SSH с виртуальной машиной Vagrant просто как обычный локальный сеанс командной оболочки.

Популярные средства повышения производительности

Почти каждый накет Python с открытым кодом, опубликованный в PyPI, так или иначе является средством новышения производительности, потому что предоставляет готовые решения для тех или иных задач. Благодаря этому нам не приходится раз за разом изобретать велосипед. Можно даже сказать, что сам язык Python — средство повышения производительности. Создается впечатление, что почти все в этом языке и в окружающем его сообществе устроепо так, чтобы сделать разработку программного обеснечения как можно более эффективной.

Так образуется цикл с положительной обратной связью. Поскольку писать код на языке Python просто и иптересно, многие нрограммисты проводят свободное время за разработкой инструментов, которые делают этот процесс еще нроще и интереснее. Здесь мы возьмем этот факт за основу для очень субъективного и ненаучного определения средства повышения производительности: это инструмент, который делает разработку проще и интереснее.

По своей природе средства повышения производительности нанравлены на те или иные элементы процесса разработки (такие, как тестирование, отладка и управление пакетами) и не являются ключевыми компопентами продуктов,

в создании которых они участвуют. Порой они даже могут вообще не упоминаться в кодовой базе проекта, несмотря на то что разработчики используют их постоянно.

Мы уже рассмотрели инструменты для управления пакетами и изоляции виртуальных сред. Несомненно, они относятся к средствам производительности, нотому что их задача — упрощать рутинные процессы настройки локальных рабочих сред. Позднее в книге вы познакомитесь с другими средствами повышения производительности для решения особых задач, таких как профилирование и тестирование. Этот раздел посвящен другим инструментам, которые заслуживают упоминания, однако в книге не нашлось более подходящих глав, чтобы нолноценно их описать.

Специализированные оболочки Python

Программисты Python проводят много времени в интерактивных сеансах интерпретаторов. Эти сеансы отлично нодходят, чтобы тестировать небольшие фрагменты кода, обращаться к документации и даже отлаживать код во время выполнения. Интерактивный сеанс Python по умолчанию очень прост и лишеп многих возможностей, таких как автозавершение или инструменты интроспективного анализа кода. К счастью, стандартную оболочку Python легко расширить и настроить.

Если вы часто работаете в интерактивной оболочке, имеет смысл настроить ее под себя. При запуске Python проверяет переменную окружения `PYTHONSTARTUP` и ищет путь к сценарию инициализации. В некоторых дистрибутивах операционных систем, где Python является стандартным системным компонентом (например, Linux или macOS), стартовый сценарий по умолчанию бывает задан изначально. Обычно оп хранится в домашнем каталоге пользователя под именем `.pythonstartup`.

Такие сценарии часто используют модуль `readline` (на основе библиотеки GNU `readline`) в сочетании с `rlcompleter`, чтобы обеспечить интерактивное автозавершение и историю команд. Оба модуля входят в стандартную библиотеку Python.

В Windows не бывает модуля readline. Вместо него пользователи Windows часто подключают пакет pyreadline, доступный в PyPI.

Если у вас нет стартового сценария Python по умолчанию, его можно легко создать самостоятельно. Простейший сценарий, который поддерживает историю команд и автозавершение, может быть таким:

```
# Стартовый файл python

import atexit
import os

try:
   import readline
except ImportError:
   print("Автозавершение недоступно: модуль readline недоступен")
else:
    import rlcompleter
    # Автозавершение
    readline.parse_and_bind('tab: complete')

    # Путь к файлу истории в домашнем каталоге пользователя.
    # Вы можете использовать другой путь для своей системы.
    history_file = os.path.expanduser('~/.python_shell_history')
    try:
        readline.read_history_file(history_file)
    except IOError:
        pass

    atexit.register(readline.write_history_file, history_file)
    del os, history_file, readline, rlcompleter
```

Создайте этот файл в домашнем каталоге и назовите его .pythonstartup. Затем добавьте в окружение переменную PYTHONSTARTUP с путем к файлу.

Если вы работаете в Linux или macOS, стартовый сценарий Python можно создать в домашнем каталоге. Затем свяжите его с переменной окружения PYTHONSTARTUP, которая настраивается в стартовом сценарии системной оболочки. Например, оболочки Bash и Korn используют файл .profile, в который можно добавить следующую строку:

```
export PYTHONSTARTUP=~/.pythonstartup
```

В системе Windows можно легко создать новую переменную окружения с нравами администратора в окне системных настроек и сохранить сценарий в удобном месте, а не в каком-либо стандартном каталоге для конкретного нользователя.

Писать сценарии PYTHONSTARTUP — нолезное упражнение, однако создать хорошую специализированную версию командной оболочки — непростое дело, для которого мало кому удается выкроить время. К счастью, существует несколько уже готовых расширенных оболочек Python, с которыми интерактивные сеансы Python становятся намного удобнее. В следующем разделе мы рассмотрим самую популярную версию — IPython.

IPython

IPython — это расширенная командная оболочка Python. Она доступна в виде пакета в PyPI, что позволяет легко установить ее с помощью `pip` или `poetry`. Среди многочисленных возможностей IPython особенно интересны следующие:

- динамический интроспективный анализ объектов;
- доступ к системной командной оболочке;
- многострочное редактирование кода;
- цветовое выделение синтаксиса;
- расширенные возможности копирования и вставки;
- поддержка прямого профилирования;
- средства отладки.

Сейчас IPython является частью более крупного проекта Jupyter — интерактивного блокнота с «живым» кодом, который можно писать на многих языках. Блокноты Jupyter популярны в сообществе анализа данных, где Python проявляет себя во всей красе. Так что стоит познакомиться с соответствующей оболочкой.

Оболочка IPython запускается командой `ipython`. После запуска вы немедленно заметите, что стандартное приглашение Python заменилось цветными цифрами, которые обозначают количество ячеек выполнения:

```
$ ipython
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]:
```

Два свойства оболочки IPython особенно полезны:

- Она позволяет легко работать с многострочным кодом, в том числе вставленным из буфера обмена.
- В ней работают сочетания клавиш и другие средства для просмотра doc-строк, документации модулей и кода импортированных модулей.

Эти особенности уже сами по себе делают IPython отличным инструментом для учебных целей. Прежде всего, если вы нашли полезные фрагменты кода (например, в этой книге), то их можно легко вставить из системного буфера обмена и редактировать так, как если бы интерпретатор Python был редактором кода. Ниже приведен снимок экрана терминала с интерактивным сеансом IPython, куда был вставлен исходный код приложения `echo` (рис. 2.1).

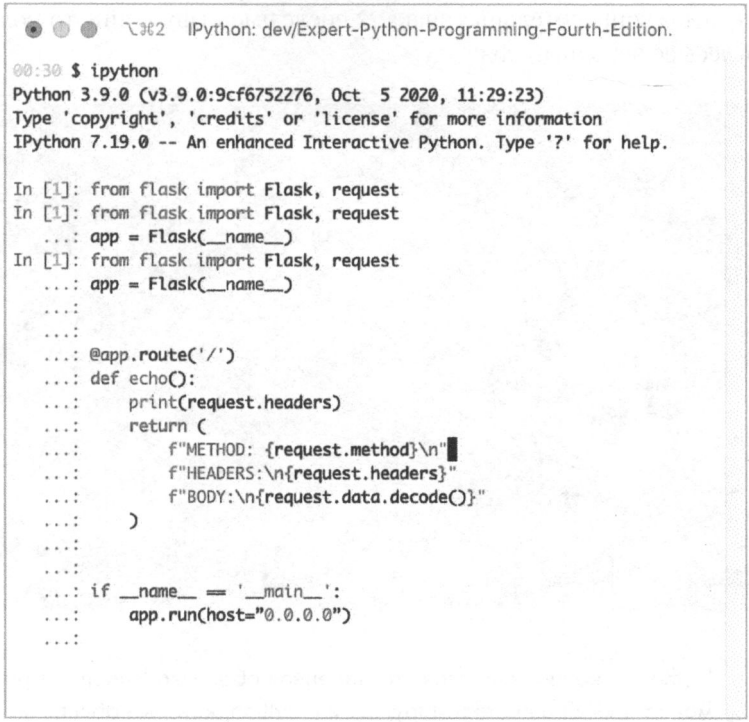

Рис. 2.1. Вставка кода в IPython

Для интроспективного анализа кода IPython предлагает быстрый способ просматривать документацию и исходный код импортированных модулей, функций и классов. Чтобы просмотреть doc-строку, просто введите имя соответствующей функции или другого участка кода и вопросительный знак `?`. В следующем сеансе терминала просматривается информация о функции `urlunparse()` из модуля `urllib.parse`:

```
In [1]: urllib.parse.urlunparse?
Signature: urllib.parse.urlunparse(components)
Docstring:
Put a parsed URL back together again.  This may result in a
slightly different, but equivalent URL, if the URL that was parsed
originally had redundant delimiters, e.g. a ? with an empty query
(the draft states that these are equivalent).
File:      /Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/urllib/parse.py
Type:      function
```

Если ввести два вопросительных знака ?? после имени функции, то можно просмотреть весь ее исходный код:

```
In [2]: urllib.parse.urlunparse??
Signature: urllib.parse.urlunparse(components)
Source:
def urlunparse(components):
    """Put a parsed URL back together again.  This may result in a
    slightly different, but equivalent URL, if the URL that was parsed
    originally had redundant delimiters, e.g. a ? with an empty query
    (the draft states that these are equivalent)."""
    scheme, netloc, url, params, query, fragment, _coerce_result = (
                                                    _coerce_
args(*components))
    if params:
        url = "%s;%s" % (url, params)
    return _coerce_result(urlunsplit((scheme, netloc, url, query,
fragment)))
File:      /Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/urllib/parse.py
Type:      function
```

IPython — не единственная расширенная оболочка Python. Обратите внимание также на проекты btpython и ptpython, которые обеспечивают аналогичные возможности, но имеют несколько иной интерфейс.

Интерактивные сеансы не только отлично подходят для экспериментов и исследования модулей, но и порой приносят пользу в реальных приложениях. В следующем разделе вы узнаете, как встроить их в свой код.

Как встроить оболочку в сценарии и программы

Иногда требуется встроить в программный продукт цикл **REPL** (Read-Eval-Print Loop, цикл «чтение — вычисление — вывод») — аналог интерактивного сеанса Python. Это дает возможность экспериментировать с кодом и анализировать его внутреннее состояние. Порой действительно проще встроить в приложение интерактивный терминал, вместо того чтобы разрабатывать собственный интерфейс командной строки, особенно если он будет использоваться относительно редко. Интерактивные интерпретаторы часто встраиваются во фреймворки веб-приложений, чтобы разработчики могли взаимодействовать с данными, хранящимися внутри приложения, через Python REPL, а не через терминалы конкретных баз данных.

Простейший модуль, который эмулирует интерактивный интерпретатор Python, уже включен в стандартную библиотеку и называется `code`.

Сценарий, который запускает интерактивные сеансы, состоит из одной команды импорта и одного вызова функции:

```
import code
code.interact()
```

Для этого интерфейса легко выполнить косметическую настройку (например, изменить приглашение, добавить заставки и сообщения при выходе), но более серьезные модификации потребуют значительных усилий. Если вам нужны расширенные возможности (такие, как цветовое выделение кода, автозавершение или прямой доступ к системной оболочке), лучше использовать уже готовые инструменты. К счастью, упомянутую в предыдущем разделе оболочку IPython так же легко встроить в программу, как и модуль `code`.

Вот как можно вызвать из вашего кода все упоминавшиеся ранее оболочки:

```
# Пример для IPython
import IPython
IPython.embed()

# Пример для bpython
import bpython
bpython.embed()

# Пример для ptpython
from ptpython.repl import embed
embed(globals(), locals())
```

Первые два аргумента функции `embed()` — это словари объектов, которые будут доступны как глобальные и локальные пространства имен в интерактивном сеансе. Их можно использовать, чтобы предварительно загрузить в сеанс модули, переменные, функции или классы, которые с большой вероятностью будут там применяться.

Интерактивные сеансы отлично подходят для того, чтобы напрямую предоставить пользователю низкоуровневый интерфейс приложения. Иногда из этих сеансов можно вручную анализировать внутреннее состояние приложения, предоставив доступ к локальным или глобальным переменным. Тем не менее если нужно отслеживать выполнение кода в интерактивном режиме, то, вероятно, стоит воспользоваться отладчиком. К счастью, в Python есть встроенный отладчик, который предоставляет нужные возможности в форме интерактивного сеанса.

Интерактивные отладчики

Отладка кода — неотъемлемая часть процесса разработки. Для многих программистов основным средством отладки остается интенсивное журналирование и функция `print()`, однако большинство профессиональных разработчиков предпочитают использовать тот или иной специализированный отладчик.

Python поставляется вместе со встроенным интерактивным отладчиком `pdb`. Его можно запустить из командной строки для существующего сценария, чтобы Python перешел в режим ретроспективной отладки, если программа завершится аварийно:

```
$ python3 -m pdb -c continue script.py
```

Того же результата можно добиться, запустив интерпретатор с флагом `-i`:

```
$ python3 -i script.py
```

Эта команда открывает интерактивный сеанс в тот момент, когда Python обычно должен завершиться. С этой точки можно начать сеанс ретроспективной отладки, для чего импортировать модуль `pdb` и использовать функцию `pdb.pm()`, как в следующем примере:

```
>>> import pdb
>>> pdb.pm()
```

Ретроспективная отладка эффективна, но она не покрывает все возможные ситуации. Она полезна, только если приложение завершается с исключением при возникновении ошибки. Между тем часто дефектный код ведет себя аномально, но не вызывает аварийного завершения программы. В таких случаях можно установить пользовательскую точку останова в конкретной строке кода функцией `breakpoint()`. В следующем примере точка останова устанавливается внутри простой функции:

```
import math

def circumference(r: float):
    breakpoint()
    return 2 * math.pi * r
```

Функция breakpoint() появилась только в Python 3.7, поэтому в старом коде иногда можно встретить следующую идиому:

```
import pdb; pdb.set_trace()
```

Эта инструкция заставляет интерпретатор Python запустить сеанс отладки в этой строке во время выполнения.

Модуль `pdb` очень полезен для отслеживания проблем, и на первый взгляд он напоминает известный отладчик GNU Debugger (GDB). Так как Python — динамический язык, сеанс `pdb` очень похож на обычный сеанс интерпретатора. Это означает, что разработчик может не только наблюдать за выполнением кода, но и вызывать любой код и даже импортировать модули.

К сожалению, из-за происхождения `pdb` (он является потомком `gdb`) первое знакомство с этим инструментом может вызвать легкий шок из-за обилия загадочных коротких команд отладки, таких как `h`, `b`, `s`, `n`, `j` и `r`. Когда у вас возникают сомнения, введите команду `help pdb`, которую можно использовать во время сеанса отладки; она выдаст подробную справку об использовании отладчика. Также можно воспользоваться сокращенной командой `h`.

Сеанс отладки в `pdb` очень прост, и в нем нет таких дополнительных возможностей, как автозавершение или цветовое выделение кода. К счастью, как и в случае с расширенными оболочками Python, в PyPI можно найти несколько улучшенных оболочек для отладки. Одна из них даже построена на базе IPython и называется `ipdb`.

Если вы хотите использовать `ipdb` вместо `pdb`, либо модифицируйте идиому отладки (`import ipdb; ipdb.set_trace()`), либо присвойте переменной окружения `PYTHONBREAKPOINT` значение `ipdb.set_trace`.

Наконец, многие IDE предлагают визуальные отладчики — очень удобные по мнению многих разработчиков. Они позволяют устанавливать точки останова в разных местах приложения, причем для этого не требуется изменять код, вручную расставляя вызовы `breakpoint()`. Визуальные отладчики также позволяют настраивать контрольные значения (variable watches): чтобы выполнение автоматически останавливалось, когда выбранная переменная принимает указанное значение.

Другие средства повышения производительности

До этого момента речь шла о средствах повышения производительности, специфических для Python. Однако, по сути, программирование на других языках не так уж сильно отличается. Какой бы язык ни использовали программисты, им часто приходится сталкиваться с одинаковыми проблемами и рутинными задачами: например, преобразовывать данные в разных форматах, загружать ресурсы по сети, выполнять поиск в файловых системах и переключаться между проектами.

Наверное, самым гибким средством повышения производительности всех времен можно считать Bash в сочетании со стандартными инструментами,

присутствующими в каждой операционной системе, которая поддерживает POSIX или входит в семейство UNIX. Вероятно, досконально изучить их все не под силу ни одному простому смертному. Однако если хорошо разбираться хотя бы в нескольких из таких систем, можно работать по-настоящему продуктивно.

Проще говоря, нет необходимости писать изощренный сценарий Python для одноразовой задачи, если вместо этого можно быстро скомбинировать несколько вызовов команд `curl`, `grep`, `sed` и `sort`. Для конкретных нетривиальных задач (например, для подсчета строк кода) часто уже существуют специализированные инструменты, разработка которых с нуля заняла бы много времени.

В следующей таблице приведен короткий список полезных утилит, которые мы считаем бесценными при работе с любым кодом. По сути, это своего рода «список классных ресурсов» для повышения производительности программирования:

Утилита	Описание
`jq` https://stedolan.github.io/jq/	Программа для обработки данных в формате JSON. В высшей степени удобна, чтобы оперировать выводом веб-API прямо из оболочки. Данные читаются из стандартного ввода, а результаты выводятся в стандартный вывод. Операции описываются на специальном языке, который очень прост в изучении
`yq` https://pypi.org/project/yq/	Родственник jq, использующий тот же синтаксис для работы с документами YAML
`curl` https://curl.se	Классический инструмент для передачи данных по URL-адресам. Чаще всего используется для взаимодействия с HTTP, но всего поддерживает более 20 протоколов
`HTTPie` https://httpie.io	Программа на Python для взаимодействия с серверами HTTP. Многие разработчики считают ее более удобной, чем curl
`autojump` https://github.com/wting/autojump	Программа командной оболочки, позволяющая быстро перейти к недавно посещенным каталогам. Незаменима для программистов, которые параллельно работают с десятками проектов. Просто введите j и несколько начальных символов имени нужного каталога, и скорее всего, вы окажетесь в нужном месте. Хорошо сочетается с рабочими процессами Poetry

Утилита	Описаиие
`cloc` https://github.com/AlDanial/cloc	Одна из лучших и самых функциональных утилит для подсчета строк кода. Также, если требуется узнать размер проекта или количество используемых в нем языков программирования или разметки, cloc быстро вернет нужную информацию
`ack-grep` https://beyondgrep.com	Продвинутая версия grep. Позволяет быстро искать конкретные строки в больших кодовых базах. Обеспечивает фильтрацию по языку программирования. Работать с `ack-grep` часто быстрее и эффективнее, чем открывать проект в IDE
`GNU parallel` https://www.gnu.org/software/parallel/	Улучшенная альтернатива xargs. Практически незаменима, если нужно выполнить несколько операций параллельно в оболочке или в сценарии Bash, особенно если важна надежность и эффективность

Итоги

Эта глава была посвящена средам разработки для программистов на Python. Мы обсудили, как важно изолировать среды для Python-проектов. Вы узнали о двух уровнях изоляции сред (уровень приложения и уровень системы), а также о различных средствах, которые позволяют изолировать среды, сохраняя целостность и обеспечивая хорошую воспроизводимость. Также были рассмотрены некоторые важные вопросы управления зависимостями Python в проектах. Глава завершилась обзором инструментов, которые улучшают возможности экспериментов с Python или отладки программ, повышая эффективность вашей работы.

Освоив все эти инструменты, вы будете хорошо подготовлены к нескольким следующим главам, где рассматриваются особенности современного синтаксиса Python. Вероятно, вам уже не терпится увидеть код на Python, поэтому мы начнем с краткого обзора нововведений, появившихся в Python в нескольких последних выпусках.

Если вы представляете себе, что сейчас происходит в Python, то следующую главу, вероятно, можно пропустить. Тем не менее хотя бы просмотрите заголовки: не исключено, что вы могли что-то упустить, ведь Python развивается очень быстро.

3

Новые возможности Python

Вероятно, одним из самых важных этапов в истории Python стал выпуск версии Python 3.0. Самые заметные изменения этой версии таковы:

- Решение различных проблем, связанных с обработкой текста, данных и Юникода.
- Отказ от классов старого формата.
- Начало реорганизации стандартной библиотеки.
- Появление аннотаций к функциям.
- Новый синтаксис обработки исключений.

Как вы знаете из главы 1, Python 3 не сохраняет обратную совместимость с Python 2. Это главная причина, по которой сообществу Python понадобилось столько лет, чтобы полноценно принять Python 3. Это стало горьким, хотя и необходимым уроком для разработчиков языка Python и профессионального сообщества.

К счастью, проблемы с принятием Python 3 не остановили эволюцию языка. С 3 декабря 2008 года (дата официального релиза Python 3.0) наблюдается стабильный поток новых крупных обновлений. Каждый новый выпуск вносит улучшения в язык, в его стандартную библиотеку и его интерпретатор. Более того, начиная с версии 3.9, Python перешел на ежегодный цикл выпуска. Это означает, что новые возможности и улучшения появляются каждый год.

Чтобы подробнее узнать о цикле выпусков Python, ознакомьтесь с документом PEP 602 («Годичный цикл релизов Python»), который доступен по адресу https://www.python.org/dev/peps/pep-0602/.

В этой главе мы поближе познакомимся с последними этапами эволюции Python. Будут рассмотрены важные новшества, которые появились в недавних выпусках. А еще мы попытаемся заглянуть в будущее и представить некоторые возможности, которые были приняты в процессе PEP и официально войдут в язык Python в ближайшее время. Вот о чем пойдет речь:

- Недавние добавления в Python.
- Не самые свежие, но важные новшества.
- Чего стоит ожидать в будущем?

Но прежде чем переходить к этим темам, начнем с технических требований.

Технические требования

В этой главе упомипаются пакеты Python, которые можно загрузить из PyPI:

- `mypy`;
- `pyright`.

О том, как устанавливать пакеты, рассказано в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%203.

Недавние добавления в Python

Каждый выпуск Python сопровождается значительпыми изменениями разного рода. Почти в каждом выпуске появляются новые элементы синтаксиса. Большинство изменепий относится к стапдартной библиотеке Python, интерпретатору CPython, Python API и C API интерпретатора CPython. Рассмотреть все эти новшества в рамках одной книги — невыполнимая задача, так что мы ограничимся новыми возможностями синтаксиса и свежими дополнениями к стандартной библиотеке.

В двух последних версиях Python можно выделить четыре основных обповления синтаксиса:

- Операторы слияния и обновлепия для словарей (появились в Python 3.9).
- Выражения присваивания (появились в Python 3.8).
- Аннотации обобщенных типов (появились в Python 3.9).
- Чисто позиционные аргументы (появились в Python 3.8).

Эти четыре возможности можно охарактеризовать как «повышение качества жизни». Они не вводят новых парадигм программирования и не требуют писать код принципиально по-иному. Они всего лишь обеспечивают более эффективные приемы написания кода и добавляют строгость в определения API.

В последние годы разработчики ядра Python больше удаляли отмершие или ненужные модули из стандартной библиотеки, чем добавляли что-то новое. Тем не менее время от времени в стандартной библиотеке появляются дополнения. В двух последних выпусках разработчики получили в свое распоряжение два совершенно новых модуля:

- Модуль `zoneinfo` для поддержки базы данных часовых поясов IANA (Internet Assigned Numbers Authority) (появился в Python 3.9).
- Модуль `graphlib` для работы с графоподобными структурами (появился в Python 3.8).

Оба модуля относительно невелики, если рассматривать размер их API. Позже мы рассмотрим примеры областей, в которых их можно применять. Но сначала познакомимся с изменениями синтаксиса, внедренными в Python 3.8 и Python 3.9.

Операторы слияния и обновления для словарей

Некоторые арифметические операторы Python работают со встроенными типами контейнеров, включая списки, кортежи, множества и словари.

В случае списков и кортежей оператор `+` (сложение) можно использовать для конкатенации двух переменных, если они относятся к одному типу. Аналогичный оператор с присвоением `+=` изменяет существующие переменные на месте. Ниже приведены примеры конкатенации списков и кортежей в интерактивном сеансе:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> value = [1, 2, 3]
>>> value += [4, 5, 6]
>>> value
[1, 2, 3, 4, 5, 6]
>>> value = (1, 2, 3)
>>> value += (4, 5, 6)
>>> value
(1, 2, 3, 4, 5, 6)
```

Что касается множеств, то существует ровно четыре бинарных (то есть имеющих два операнда) оператора, которые возвращают новое множество:

- **Пересеченне**: онератор & (поразрядная операция И). Создает множество с элементами, общими для обоих множеств.
- **Объединенне**: оператор | (поразрядная операция ИЛИ). Создает множество с элементами, входящими хотя бы в одно множество.
- **Разность множеств**: онератор - (вычитание). Создает множество с элементами левого множества, не входящими в правое множество.
- **Снмметрнческая разность**: онератор ^ (поразрядная онерация ИЛИ-НЕ). Создает множество с элементами, входящими в одно или другое множество, но не в оба одновременно.

Вот примеры онераций пересечения и объединения множеств в интерактивном сеансе:

```
>>> {1, 2, 3} & {1, 4}
{1}
>>> {1, 2, 3} | {1, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} - {1, 4}
{2, 3}
>>> {1, 2, 3} ^ {1, 4}
{2, 3, 4}
```

В Python очень долго не было специального бинарного онератора, который создавал бы новый словарь из двух существующих словарей. Начиная с Python 3.9, можно иснользовать онераторы | (поразрядная операция ИЛИ) и |= (поразрядная операция ИЛИ на месте) для слияния и обновления словарей. Этот способ объединения двух словарей следует считать идиоматическим. Причины добавления новых операторов описаны в PEP 584 («Добавление онераторов объединения для словарей»).

Идиома программирования — распространенный и наиболее предпочтительный способ выполнения конкретных операций в языке программирования. Идиоматичность кода — важная часть культуры Python. В «Дзене Python» сказано: «Должен существовать один — и желательно только один — очевидный способ сделать это». Другие идиомы рассматриваются в главе 4 «Python в сравнении с другими языками».

Чтобы вынолнить слияние двух словарей в новый словарь, используйте такое выражение:

```
словарь_1 | словарь_2
```

Итоговый словарь будет совершенно новым объектом, содержащим все ключи обоих исходных словарей. Если в обоих словарях встречаются одинаковые ключи, то новый объект получает значения из правого словаря.

В следующем примере этот синтаксис используется с двумя литералами словарей. Левый словарь обновляется значениями из правого словаря:

```
>>> {'a': 1} | {'a': 3, 'b': 2}
{'a': 3, 'b': 2}
```

Если нужно обновить переменную словаря ключами, взятыми из другого словаря, можно использовать оператор |= для обновления на месте:

```
Существующий_словарь |= другой_словарь
```

Вот пример использования этого оператора с конкретной переменной:

```
>>> mydict = {'a': 1}
>>> mydict |= {'a': 3, 'b': 2}
>>> mydict
{'a': 3, 'b': 2}
```

В старых версиях Python простейшим способом обновить существующий словарь содержимым другого словаря был метод `update()`, как в следующем примере:

```
Существующий_словарь |= другой_словарь
```

Этот метод изменяет существующий словарь на месте и не возвращает значения[1]. А следовательно, он не дает простого способа получить объединенный словарь как выражение и всегда используется в формате инструкции.

Разница между выражениями и инструкциями объясняется в разделе «Выражения присваивания».

Альтернативный способ: распаковка словаря

Мало кто знает, что в Python до версии 3.9 уже поддерживался довольно компактный способ слияния двух словарей, основанный на так называемой распаковке словарей. Распаковка словарей в литералах `dict` появилась в Python 3.5 и описана в документе PEP 448 «Дополнительные обобщения распаковки». Синтаксис распаковки двух (и более) словарей в новый объект выглядит так:

```
Существующий_словарь |= другой_словарь
```

[1] Технически он возвращает None. — *Примеч. ред.*

Пример с литералами словарей:

```
>>> a = {'a': 1}; b = {'a':3, 'b': 2}
>>> {**a, **b}
{'a': 3, 'b': 2}
```

Эта возможность в сочетании с распаковкой списков (синтаксис: `*значение`) может быть знакома читателям, у которых есть опыт написания функций, способных принимать неопределенно большой набор позиционных и именованных аргументов (такие функции называются *вариативными*). Распаковка особенно полезна при написании декораторов.

Вариативные функции и декораторы подробно рассматриваются в главе 4 «Python в сравнении с другими языками».

Хотя распаковка словарей повсеместно встречается в определениях функций, ее крайне редко применяют для слияния словарей, и она может смутить менее опытных программистов, читающих ваш код. Поэтому в коде на Python 3.9 и более новых версий стоит отдавать предпочтение новому оператору слияния. В более старых версиях Python иногда лучше использовать временный словарь и простой метод `update()`.

Альтернативный способ: ChainMap из модуля collections

Еще один способ создания объектов, которые с функциональной точки зрения являются результатом слияния двух словарей, основан на классе `ChainMap` из модуля `collections`. Этот класс-обертка принимает несколько объектов отображений (в данном случае — словарей) и ведет себя как единый объект отображения.

Синтаксис слияния двух словарей с помощью `ChainMap`:

```
new_map = ChainMap(словарь_2, словарь_1)
```

Обратите внимание, что словари идут в обратном порядке по сравнению с оператором `|`. Это означает, что при обращении к конкретному ключу объекта `new_map` поиск по внутренним объектам будет выполняться слева направо. Следующий пример демонстрирует операции с использованием класса `ChainMap`:

```
>>> from collections import ChainMap
>>> user_account = {"iban": "GB71BARC20031885581746", "type":
"account"}
>>> user_profile = {"display_name": "John Doe", "type": "profile"}
>>> user = ChainMap(user_account, user_profile)
>>> user["iban"]
```

```
'GB71BARC20031885581746'
>>> user["display_name"]
'John Doe'
>>> user["type"]
'account'
```

Этот пример ясно показывает, что итоговый пользовательский объект ChainMap содержит ключи как из словаря user_account, так и из user_pro Если какие-то ключи перекрываются, то экземпляр ChainMap возвращает з ние, соответствующее указанному ключу, из самого левого отображения. Е его поведение полностью противоположно оператору слияния для слов

ChainMap — это объект-обертка. Это означает, что он не копирует содерж исходных отображений, а сохраняет ссылки на них. Если используемые об изменятся, ChainMap сможет вернуть обновленные данные. Рассмотрим продолжение предыдущего интерактивного сеанса:

```
'GB71BARC20031885581746'
>>> user["display_name"]
'John Doe'
>>> user["type"]
'account'
```

Кроме того, объект ChainMap доступен для записи. Правда, следует помнит операции записи, обновления и удаления распространяются только на левое отображение. Если использовать их без должной осторожности, это м привести к неожиданностям, как в этом продолжении предыдущего сеан

```
>>> user["display_name"] = "John Doe"
>>> user["age"] = 33
>>> user["type"] = "extension"
>>> user_profile
{'display_name': 'Abraham Lincoln', 'type': 'profile'}
>>> user_account
{'iban': 'GB71BARC20031885581746', 'type': 'extension', 'display_name':
'John Doe', 'age': 33}
```

В этом примере ключ 'display_name' был применен к словарю user_ac тогда как изначально этот ключ был в user_profile. Такое обратное распр ление значений, присущее ChainMap, во многих ситуациях нежелательно. П му стандартная идиома использования ChainMap для слияния двух сло состоит в том, чтобы явно преобразовывать результат слияния в новый сл В следующем примере используются ранее определенные входные сло

```
>>> dict(ChainMap(user_account, user_profile))
{'display_name': 'John Doe', 'type': 'account', 'iban':
'GB71BARC20031885581746'}
```

Если требуется просто объединить два словаря, лучше использовать новый оператор слияния, а не `ChainMap`. Впрочем, это не означает, что класс `ChainMap` полностью бесполезен. Когда нужно именно прямое и обратное распространение изменений, используйте `ChainMap`. Кроме того, `ChainMap` также работает с отображениями любого типа. Следовательно, если вы хотите обеспечить унифицированный доступ к разным объектам, которые ведут себя подобно словарям, то `ChainMap` подойдет в качестве единого инструмента слияния.

Если у вас есть собственный класс, подобный словарю, его всегда можно расширить специальным методом `__or__()`, чтобы обеспечить совместимость с оператором `|` и не задействовать `ChainMap`. Переопределение специальных методов рассматривается в главе 4 «Python в сравнении с другими языками». Однако использовать `ChainMap` обычно проще, чем писать собственный метод `__or__()`. Кроме того, `ChainMap` позволяет работать с существующими экземплярами классов, которые нельзя изменить.

Чаще всего `ChainMap` вместо распаковки словарей или оператора объединения используют ради обратной совместимости. В версиях Python, предшествующих 3.9, не работает новый синтаксис оператора слияния для словарей. Таким образом, если вы пишете код для старых версий Python, используйте `ChainMap`. А если нет, то лучше выбрать оператор слияния.

Другое новшество синтаксиса, значительно влияющее на обратную совместимость, — выражения присваивания.

Выражения присваивания

Выражения присваивания — примечательная возможность, потому что их появление затронуло фундаментальную часть синтаксиса Python: различия между выражениями (expressions) и инструкциями (statements). Выражения и инструкции — ключевые структурные элементы почти каждого языка программирования. Различия между ними очень просты: у выражений есть значение, а у инструкций нет.

Инструкции можно рассматривать как последовательные действия или операции, которые выполняет программа. Таким образом, присваивания значений, условные конструкции `if`, циклы `for` и `while` — все это примеры инструкций. Определения функций и классов — тоже инструкции.

Выражения — это, по сути, все, что можно включить в условие `if`. Типичные примеры выражений — литералы, значения, возвращаемые операторами (не считая операторов, работающих на месте), и включения (comprehensions) сни-

сков, словарей и множеств[1]. Вызовы функций и методов тоже относятся к выражениям.

Во многих языках программирования некоторые концепции выражаются только с помощью инструкций. К ним обычно относятся:

- Определения функций и классов.
- Циклы.
- Условные конструкции `if...else`.
- Присваивание значений переменным.

Python преодолел эту безальтернативность: в нем есть синтаксические средства, которые позволяют записать аналоги ряда инструкций в форме выражений, а именно:

- Лямбда-выражения для анонимных функций как аналог определений функций:

```
lambda x: x**2
```

- Создание экземпляров объектов-типов как аналог определений классов:

```
type("MyClass", (), {})
```

- Различные включения как аналог циклов:

```
squares_of_2 = [x**2 for x in range(10)]
```

- Условные выражения как аналог инструкций `if... else`:

```
"odd" if number % 2 else "even"
```

Однако долгие годы в Python не было синтаксиса, который передавал бы семантику присваивания значения переменной в форме выражения. Это было сознательным решением создателей Python. В таких языках, как C, где присваивание бывает и в форме выражения, и в форме инструкции, оператор присваивания часто нутают с проверкой равенства. Каждый, кто нрограммировал на C, может подтвердить, что эта путаница вызывает много раздражающих ошибок. Рассмотрим пример кода на C:

```
int err = 0;
if (err = 1) {
    printf("Error occured");
}
```

[1] List (dict/set) comprehensions обычно переводят как «списковые включения», хотя термин немного сбивает с толку. В реальной жизни большинство питонистов говорят «генератор списков» (словарей/множеств). Но в переводе этот термин конфликтовал бы с generator. — *Примеч. ред.*

Сравните его со следующим фрагментом:

```
int err = 0;
if (err == 1) {
    printf("Error occured");
}
```

В языке C оба примера правильны с точки зрения синтаксиса, потому что `err = 1` — это выражение языка C, при вычислении которого получится 1. Сравните с языком Python, где следующий код приведет к синтаксической ошибке:

```
err = 0
if err = 1:
     print("Error occured")
```

Однако в редких случаях была бы удобна операция присваивания, которая возвращает значение. К счастью, в Python 3.8 появился специальный оператор `:=`, который присваивает значение переменной, но ведет себя как выражение, а не как инструкция. Из-за внешнего вида его быстро окрестили «моржовым оператором» (walrus operator).

Откровенно говоря, сценарии практического применения этого оператора встречаются нечасто. Однако там, где оператор `:=` уместен, он помогает сделать код компактнее, и такой код часто бывает проще для понимания, потому что он улучшает соотношение «сигнал/шум». Чаще всего «моржовый оператор» применяется, когда нужно вычислить сложное значение, а затем сразу же использовать его в следующих инструкциях.

Типичные примеры такого рода попадаются при работе с регулярными выражениями. Представим себе простое приложение, которое читает исходный код, написанный на Python, и с помощью регулярных выражений ищет в коде импортированные модули.

Без выражений присваивания код приложения может выглядеть примерно так:

```
import os
import re
import sys

import_re = re.compile(
    r"^\s*import\s+\.{0,2}((\w+\.)*(\w+))\s*$"
)
import_from_re = re.compile(
    r"^\s*from\s+\.{0,2}((\w+\.)*(\w+))\s+import\s+(\w+|\*)+\s*$"
)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"usage: {os.path.basename(__file__)} file-name")
```

```
        sys.exit(1)

    with open(sys.argv[1]) as file:
        for line in file:
            match = import_re.search(line)
            if match:
                print(match.groups()[0])

            match = import_from_re.search(line)
            if match:
                print(match.groups()[0])
```

Можно заметить, что нам пришлось дважды повторить конструкцию, которая ищет совпадения для сложного выражения и затем выводит подгруппы каждого совпадения. Этот блок кода можно переписать с выражениями присваивания:

```
if match := import_re.match(line):
    print(match.groups()[0])

if match := import_from_re.match(line):
    print(match.groups()[0])
```

Как видите, нам удалось немного улучшить удобочитаемость. Такие правки особенно полезны, когда в коде приходится многократно повторять однотипные конструкции. Если промежуточные результаты раз за разом присваиваются одной и той же переменной, код выглядит громоздким.

Выражения присваивания также полезны в ситуациях, когда одни и те же данные используются в нескольких местах крупных выражений. Рассмотрим пример литерала словаря, который содержит заранее определенные сведения о вымышленном пользователе:

```
first_name = "John"
last_name = "Doe"
height = 168
weight = 70

user = {
    "first_name": first_name,
    "last_name": last_name,
    "display_name": f"{first_name} {last_name}",
    "height":  height,
    "weight": weight,
    "bmi": weight / (height / 100) ** 2,
}
```

Допустим, в нашем случае важно сохранить целостность данных. В частности, отображаемое имя (`display_name`) всегда должно состоять из имени (`first_name`) и фамилии (`last_name`), а индекс массы тела (`bmi`) должен вычисляться на ос-

нове веса (`weight`) и роста (`height`). Чтобы предотвратить возможные ошибки при редактировании компонентов данных, их приходится определять как отдельные переменные. После того как словарь создан, эти переменные больше не нужны. Выражения присваивания позволяют записать тот же словарь компактнее:

```
user = {
    "first_name": (first_name := "John"),
    "last_name": (last_name := "Doe"),
    "display_name": f"{first_name} {last_name}",
    "height": (height := 168),
    "weight": (weight := 70),
    "bmi": weight / (height / 100) ** 2,
}
```

Здесь выражения присваивания пришлось заключить в круглые скобки, потому что без них синтаксис оператора `:=` конфликтует с символом двоеточия `:` в литералах словарей.

Выражения присваивания улучшают внешний вид кода, и ничего более. Всегда следите за тем, чтобы с ними программа лучше читалась, а не становилась более запутанной.

Аннотации обобщенных типов

Хотя без аннотаций типов в Python вполне можно обойтись, они становятся все более популярными. Они позволяют помечать переменные, аргументы и возвращаемые значения функций определениями типов. Аннотации типов полезны для документирования, а также помогают проверять правильность кода сторонними инструментами. Многие интегрированные среды распознают аннотации типов и визуально выделяют потенциальные проблемы с типами. Также существуют статические средства проверки типов (такие, как `mypy` или `pyright`), которые могут анализировать целую кодовую базу и сообщать обо всех ошибках типизации в коде, использующем аннотации.

История проекта mypy очень интересна. Он начался как исследование для PhD диссертации Юкки Лехтосало, однако развился в полной мере, когда Юкка начал работать вместе с Гвидо ван Россумом (создателем Python) в компании Dropbox. Об этой истории можно подробнее узнать из прощального письма, адресованного Гвидо, в техническом блоге Dropbox по адресу https://blog.dropbox.com/topics/company/thank-you--guido.

В простейшей форме аннотации типов можно использовать со встроенными или пользовательскими типами, чтобы указывать желательные типы аргументов

функций и возвращаемых значений, а также локальных переменных. Рассмотрим следующую функцию, которая ищет ключи (без учета регистра) в словаре со строковыми ключами:

```
from typing import Any

def get_ci(d: dict, key: str) -> Any:
    for k, v in d.items():
        if key.lower() == k.lower():
            return v
```

Конечно, это весьма наивная реализация поиска без учета регистра. Чтобы поиск был эффективнее, вероятно, стоит воспользоваться специальным классом. Мы вернемся к этой теме позднее.

Первая инструкция импортирует из модуля `typing` тип `Any`, который определяет, что переменная или аргумент могут быть произвольного типа. Сигнатура функции указывает, что первый аргумент `d` должен быть словарем, а второй аргумент `key` — строкой. Сигнатура завершается спецификацией возвращаемого значения, которое может иметь любой тип.

Если вы используете средства проверки тинов, то предыдущих аннотаций будет достаточно, чтобы выявлять многие ошибки. Например, если вызывающая сторона изменит порядок позиционных аргументов, эта ошибка быстро обнаружится, потому что аргументы `key` и `d` аннотированы разными тинами. Однако средства проверки не поднимут тревогу, если пользователь передаст словарь с разными типами ключей. Именно по этой причине обобщенные типы (`tuple`, `list`, `dict`, `set`, `frozenset` и многие другие) можно дополнительно снабжать аннотациями типов их содержимого. Для словаря аннотация записывается в следующей форме:

```
dict[тип_ключа, тип_значения]
```

Вот как будет выглядеть сигнатура функции `get_ci()` с аннотациями типов, устанавливающими более строгие ограничения:

```
def get_ci(d: dict[str, Any], key: str) -> Any: ...
```

В старых версиях Python встроенные типы коллекций нельзя было так легко пометить тином их содержимого. Модуль `typing` предоставлял специальные тины, которые можно было использовать для этой цели. К ним относятся:

- `typing.Dict` для словарей;
- `typing.List` для списков;
- `typing.Tuple` для кортежей;

- `typing.Set` для множеств;
- `typing.FrozenSet` для неизменяемых множеств.

Эти типы все еще могут пригодиться, если нужно обеспечить функциональность для широкого спектра версий Python. Но если вы пишете код только для Python 3.9 и более новых версий, используйте встроенные обобщенные типы. Импортирование этих типов из модуля `typing` считается устаревшим, и впоследствии они будут исключены из Python.

Аннотации типов подробнее рассматриваются в главе 4 «Python в сравнении с другими языками».

Чисто позиционные параметры

Python предлагает довольно гибкие механизмы передачи аргументов функциям. Аргументы можно передавать двумя способами:

- как **позиционные** аргументы;
- как **именованные** аргументы.

Для многих функций вызывающая сторона сама выбирает, как передавать аргументы. И это хорошо, потому что программист, использующий функцию, может счесть, что конкретный формат записи более удобен или лучше читается в той или иной ситуации. Для примера возьмем функцию, которая объединяет строки с разделителем:

```
def concatenate(first: str, second: str, delim: str):
    return delim.join([first, second])
```

Эту функцию можно вызвать разными способами:

- с позиционными аргументами: `concatenate("John", "Doe", " ")`;
- с именованными аргументами: `concatenate(first="John", second="Doe", delim=" ")`;
- с комбинацией позиционных и именованных аргументов: `concatenate("John", "Doe", delim=" ")`.

Если вы пишете библиотеку, рассчитанную на повторное использование, то, возможно, уже представляете себе, как она будет применяться. Иногда вы по опыту знаете, что с одними паттернами использования код будет читаться лучше, а с другими — хуже. А возможно, вы еще не уверены в том, какой вариант эффективнее, и хотите сделать так, чтобы API библиотеки можно было изменить за разумное время без вреда для пользователей. В любом случае рекомендуется

определять сигнатуры функций таким образом, чтобы они поддерживали предполагаемый сценарий использования, но при этом допускали будущие расширения.

После того как библиотека онубликована, сигнатура функции определяет контракт использования этой библиотеки. Любые изменения в именах и порядке аргументов могут нарушить работоспособность приложений, в которых применяется эта библиотека.

Если в какой-то момент времени вы сочтете, что имена аргументов `first` и `second` недостаточно хорошо отражают их назначение, их не удастся изменить, не нарушив обратную совместимость. Ведь какой-нибудь программист может вызывать функцию так:

```
concatenate(first="John", second="Doe", delim=" ")
```

Если вы захотите преобразовать функцию так, чтобы она могла принимать произвольное количество строк, это тоже не удастся сделать, не нарушив обратную совместимость, потому что может найтись программист, который использует такой вызов:

```
concatenate("John", "Doe", " ")
```

К счастью, в Python 3.8 появилась возможность определять конкретные аргументы как чисто нозиционные. При этом можно указать, какие аргументы не могут передаваться как именованные, чтобы избежать проблем с обратной совместимостью в будущем. Также можно обозначить некоторые аргументы как чисто именованные. Если тщательно продумывать, какие аргументы должны передаваться как чисто позиционные, а какие — как чисто именованные, это поможет сделать определения функций более устойчивыми перед будущими изменениями. Наша функция `concatenate()`, определенная с чисто позиционными и чисто именованными аргументами, могла бы выглядеть так:

```
def concatenate(first: str, second: str, /, *, delim: str):
    return delim.join([first, second])
```

Такое определение читается следующим образом:

- Все аргументы, предшествующие символу `/`, являются чисто позиционными.
- Все аргументы, следующие за символом `*`, являются чисто именованными.

Это определение гарантирует, что функцию `concatenate()` можно будет вызвать только в такой форме:

```
concatenate("John", "Doe", delim=" ")
```

При попытке вызвать фупкцию по-другому вы получите ошибку `TypeError`, как в следующем примере:

```
>>> concatenate("John", "Doe", " ")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: concatenate() takes 2 positional arguments but 3 were given
```

Допустим, мы опубликовали функцию в библиотеке в последнем формате, а теперь хотим, чтобы она принимала неограничепное количество позиционных аргумептов. Поскольку функцию можно вызвать только одним способом, мы можем воспользоваться распаковкой аргумептов и изменить код так:

```
def concatenate(*items, delim: str):
    return delim.join(items)
```

Аргумент `*items` сохрапяет все позиционпые аргументы в кортеже `items`. Благодаря этим изменениям пользователи смогут вызывать фупкцию с переменпым количеством позиционных аргумептов, как в следующем примере:

```
>>> concatenate("John", "Doe", delim=" ")
'John Doe'
>>> concatenate("Ronald", "Reuel", "Tolkien", delim=" ")
'Ronald Reuel Tolkien'
>>> concatenate("Jay", delim=" ")
'Jay'
>>> concatenate(delim=" ")
''
```

Чисто позициопные и чисто имеповаппые аргументы — отличное подспорье для создателей библиотек: они оставляют простор для будущих измепепий, которые не затронут существующих пользователей. Но эти два вида аргументов также полезны при паписапии приложений, особенно если вы работаете с другими программистами. С помощью чисто позиционных и чисто именованных аргументов можпо гарантировать, что фупкции будут вызываться именпо так, как задумано. Это поможет упростить будущий рефакторипг.

Модуль zoneinfo

Работа со временем и часовыми поясами — один из самых сложных аспектов программирования. В первую очередь это объясняется мпогочисленными неверными представлениями программистов о времени и часовых поясах. Еще одна нричина — непрекращающийся поток коррективов, которые вносятся в структуру часовых поясов. Такие изменения происходят каждый год, часто по политическим мотивам.

Начиная с версии Python 3.9, работать с информацией о текущих и исторических часовых поясах стало проще, чем когда-либо. В стандартной библиотеке Python есть модуль `zoneinfo`, который обеспечивает интерфейс доступа к базе данных часовых поясов. Это может быть либо база, предоставленная вашей операционной системой, либо оригинальный пакет `tzdata` из PyPI.

Пакеты из PyPI считаются сторонними (third-party), а модули стандартной библиотеки — оригинальными (first-party). Пакет `tzdata` уникален тем, что его сопровождают разработчики ядра CPython. Содержимое базы данных IANA извлекается в отдельные пакеты PyPI для того, чтобы обеспечить регулярные обновления, не зависящие от графика выпусков CPython.

Чтобы использовать информацию часовых поясов, создайте объект `ZoneInfo` следующим вызовом конструктора:

```
ZoneInfo(timezone_key)
```

Здесь `timezone_key` — имя файла из базы данных часовых поясов IANA. Имена файлов подобны строковому представлению часовых поясов в различных приложениях, например:

- `Europe/Warsaw`
- `Asia/Tel_Aviv`
- `America/Fort_Nelson`
- `GMT-0`

Экземпляры класса `ZoneInfo` можно нередавать в параметре `tzinfo` конструктора объекта `datetime`, как в следующем нримере:

```
from datetime import datetime
from zoneinfo import ZoneInfo

dt = datetime(2020, 11, 28, tzinfo=ZoneInfo("Europe/Warsaw"))
```

Это позволяет создавать объекты даты/времени с ноддержкой часовых поясов. Такие объекты необходимы, чтобы правильно вычислять разницу во времени в разных часовых поясах, принимая во внимание переход на летнее и зимнее время, а также исторические изменения в базе данных IANA.

Полный список всех часовых поясов, доступных в вашей системе, можно получить с помощью функции `zoneinfo.available_timezones()`.

Модуль graphlib

Еще одно интересное нововведение в стандартной библиотеке Python — модуль `graphlib`, появившийся в Python 3.9. Он предоставляет средства для работы с графоподобными структурами.

Графом называется структура данных, которая состоит из вершин, соединенных ребрами. Изучением графов занимается специальная область математики — теория графов. В зависимости от типа ребер можно выделить две основные разновидности графов:

- В **неорнентированном графе** у ребер нет направления. Если бы граф описывал систему городов, соединенных дорогами, то ребра неориентированного графа соответствовали бы дорогам с двустороппим движением, по которым можно перемещаться в любом направлении. Таким образом, если в неориентированпом графе две вершины A и B соединены ребром E, то по этому ребру можно переместиться как из A в B, так и из B в A.
- В ориентированном графе с каждым ребром связано онределенпое направление. Возвращаясь к предыдущему примеру с городами и дорогами, ребра в ориентированном графе соответствовали бы дорогам с односторонним движением, но которым можно перемещаться только от начальной точки. Если две вершины A и B соединены ребром E, которое начинается от вершины A, то но этому ребру можно переместиться из A в B, но не из B в A.

Кроме того, графы могут быть циклическими или ациклическими. **Цнклнческий граф** содержит как минимум один цикл — замкнутый путь, который пачинается и завершается в одной и той же вершине. В **ациклическом графе** нет ни одного цикла. На рис. 3.1 изображепы примеры ориентированных и неориентированных графов.

В теории графов рассматривается мпого математических проблем, для моделирования которых используются графовые структуры. В программировании графы помогают решать многие алгоритмические задачи. В computer science с помощью графов можно представлять нотоки данных или отношения между объектами. Из этого вытекает множество практических применений, в том числе:

- моделирование деревьев зависимостей;
- представление знаний в машиночитаемом формате;
- визуализация информации;
- моделирование транспортных систем.

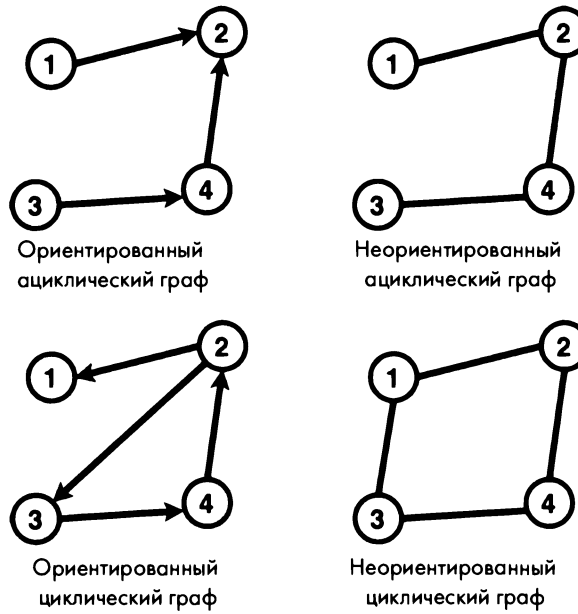

Рис. 3.1. Графические представления различных видов графов

Модуль `graphlib` предназначен для того, чтобы с его помощью программисты на Python работали с графами. Это новый модуль, и сейчас он включает только один вспомогательный класс `TopologicalSorter`. Как можно догадаться по имени, этот класс выполняет топологическую сортировку ориентированных ациклических графов.

Топологическая сортировка упорядочивает вершины ориентированных ациклических графов (directed acyclic graph, **DAG**) особым образом. В результате этой сортировки получается список всех вершин, где каждая вершина идет перед теми вершинами, к которым от нее можно перейти. Иными словами:

- Первой в списке будет вершина, к которой невозможно перейти ни от какой другой вершины.
- Каждой следующей в списке будет вершина, от которой невозможно перейти ни к одной предыдущей вершине.
- Последней в списке будет вершина, от которой невозможно перейти ни к какой другой вершине.

Бывают графы, которые можно упорядочить несколькими способами, удовлетворяющими требованиям топологической сортировки. На рис. 3.2 представлен пример ориентированного ациклического графа с тремя вариантами топологической сортировки:

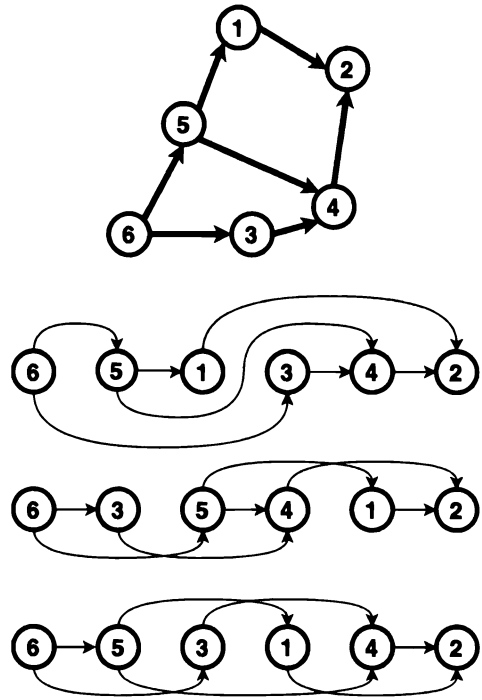

Рис. 3.2. Различные варианты топологической сортировки одного графа

Следующая задача поможет лучше понять, зачем нужна топологическая сортировка. Допустим, есть сложная операция, состоящая из множества зависимых задач. Например, это может быть миграция нескольких таблиц из одной системы баз данных в другую. Это хорошо известная задача, и существует немало готовых инструментов для миграции данных между разными системами управления базами данных. Но для этого примера предположим, будто таких систем нет и решение придется строить с нуля.

В реляционных системах управления базами данных между строками таблиц часто существуют перекрестные ссылки, целостность которых защищена ограничениями внешнего ключа. Если важно в любой момент времени сохранять ссылочную целостность базы данных, то придется перемещать все таблицы в определенном порядке. Предположим, что база состоит из следующих таблиц:

- Таблица `customers` содержит личную информацию о клиентах.
- Таблица `accounts` содержит информацию о счетах клиентов, включая баланс. У одного пользователя может быть несколько счетов (например, персональный и счет предприятия), но один счет не может принадлежать нескольким пользователям.

- Таблица `products` содержит информацию о товарах, которые продаются в системе.
- Таблица `orders` содержит отдельные заказы, оформленные одним пользователем с одного счета.
- Таблица `order_products` содержит информацию о количестве отдельных товаров в каждом заказе.

В Python нет специального типа данных для представления графов. Впрочем, обычный словарь хорошо подходит, чтобы описывать отношения между ключами и значениями. Определим отношения между таблицами из нашего примера:

```
table_references = {
    "customers": set(),
    "accounts": {"customers"},
    "products": set(),
    "orders": {"accounts", "customers"},
    "order_products": {"orders", "products"},
}
```

Если граф не содержит циклов, к нему можно применить топологическую сортировку. Результатом сортировки будет возможный порядок миграции таблиц. Конструктор класса `graphlib.TopologicalSorter` получает на входе один словарь, ключи которого представляют начальные вершины, а значения — множества конечных вершин. Это означает, что переменную `table_references` можно напрямую передать конструктору `TopologicalSorter()`.

Чтобы выполнить топологическую сортировку, можно вызвать метод `static_order()`, как в следующем фрагменте интерактивного сеанса:

```
>>> from graphlib import TopologicalSorter
>>> table_references = {
...     "customers": set(),
...     "accounts": {"customers"},
...     "products": set(),
...     "orders": {"accounts", "customers"},
...     "order_products": {"orders", "products"},
... }
>>> sorter = TopologicalSorter(table_references)
>>> list(sorter.static_order())
['customers', 'products', 'accounts', 'orders', 'order_products']
```

Топологическую сортировку можно выполнять только над ориентированными ациклическими графами. `TopologicalSorter` не проверяет наличие циклов при

инициализации, но обнаруживает их в процессе сортировки. Обнаружив цикл, метод `static_order()` выдает исключение `graphlib.CycleError`.

Конечно, задача из нашего примера тривиальна и легко решается вручную. Однако реальные базы данных часто состоят из десятков или даже сотен таблиц. Вручную составлять план миграции для баз данных такого размера — очень трудоемкое занятие, в котором легко ошибиться.

Возможности, которые мы рассмотрели, появились относительно недавно, так что пройдет некоторое время, прежде чем они внедрятся в массовую практику программирования на Python. Дело в том, что они не обладают обратной совместимостью, а старые версии Python все еще поддерживаются многими разработчиками библиотек.

В следующем разделе мы рассмотрим некоторые важные элементы, появившиеся в Python 3.6 и Python 3.7, так что наш обзор охватит более широкий диапазон версий Python. Хотя не все эти элементы одинаково популярны, мы все же надеемся, что вы узнаете что-то интересное для себя.

Не самые свежие, но важные новшества

В каждом выпуске Python появляется что-то новое. Некоторые новшества стали настоящими откровепиями: они существенно расширили возможности программирования и были почти мгновенно приняты сообществом. Преимущества других изменений поначалу не столь очевидны, и потребуется время, чтобы программисты оценили их по достоипству.

Мы были свидетелями того, как это происходило с аннотациями функций. Они поддерживались в Python с самого первого выпуска 3.0, однако ушли годы на формирование экосистемы инструментов, в которых они бы эффективно использовались. Тенерь аннотации стали практически стандартным компонентом современных приложений на Python.

Разработчики ядра Python предельно консервативно относятся к добавлению новых модулей в стандартную библиотеку, и свежие дополнения появляются редко. Не исключено, что вы быстро забудете о модулях `graphlib` или `zoneinfo`, если вам не приходится часто работать с задачами, которые требуют манипуляций с графоподобными структурами или скрупулезной обработки часовых поясов. Возможно, вы уже забыли о других примечательных дополнениях в Python, которые появились за последние несколько лет. Поэтому ниже приводится краткий обзор важных изменений в версиях до Python 3.7. Это либо

мелкие, но интересные новшества, которые легко упустить из виду, либо то, к чему привыкаешь не сразу.

Функция breakpoint()

Тема отладчиков рассматривалась в главе 2 «Современные среды разработки для Python». Упоминавшаяся там функция `breakpoint()` считается идиоматическим способом вызова отладчика Python.

Эта функция появилась в Python 3.7, так что она доступна уже довольно давно. Тем не менее это одно из тех изменений, которые требуют определенных усилий, чтобы войти в привычку. Нас долгие годы учили, что простейший способ вызвать отладчик из кода Python — это такая конструкция:

```
import pdb; pdb.set_trace()
```

Она выглядит некрасиво и не особо наглядно, но если вы, как многие программисты, используете этот прием ежедневно на протяжении долгих лет, то он давно закрепился у вас в подкорке. Возникла проблема? Переходите в код, нажимаете несколько клавиш для запуска `pdb`, а затем перезапускаете программу. Теперь вы оказываетесь в оболочке интерпретатора в той самой точке, где возникла ошибка. Разобрались? Возвращаетесь к коду, удаляете `import pdb; pdb.set_trace()` и начинаете работать над исправлением.

Тогда о чем беспокоиться? Разве это не вопрос личных предпочтений? Ведь точки останова даже не попадают в окончательную версию кода.

Действительно, отладка часто оказывается индивидуальной и глубоко личной задачей. Мы часто проводим долгие часы в поиске ошибок, диагностике и чтении кода снова и снова, отчаянно пытаясь найти тот мелкий дефект, из-за которого приложение отказывается работать. Когда вы глубоко сосредоточены на поиске причины проблемы, определенно стоит использовать инструменты, которые кажутся вам наиболее удобными. Некоторым программистам удобнее отладчики, интегрированные в IDE. Другие вообще не используют отладчики, предпочитая изощренные вызовы `print()`, разбросанные по всему коду. Всегда выбирайте то, с чем вам комфортнее работать.

Но если вы привыкли к традиционному отладчику на основе командной оболочки, то функция `breakpoint()` упростит вам работу. Ее главное преимущество в том, что она не привязана к конкретному отладчику. По умолчанию она запускает сеанс `pdb`, но эту настройку можно изменить с помощью переменной окружения `PYTHONBREAKPOINT`. Если вам привычнее альтернативный отладчик (например, `ipdb`, уномипавшийся в главе 2 «Современные среды разработки для Python»), присвойте этой переменной окружения соответствующее значение.

На практике предпочтительный отладчик обычно назначается в профильном сценарии оболочки, чтобы вам не приходилось настраивать PYTHONBREAKPOINT в каждом сеансе оболочки. Например, если вы пользуетесь Bash и хотите всегда запускать ipdb вместо pdb, включите следующую инструкцию в файл .bash_profile:

```
PYTHONBREAKPOINT=ipdb.set_trace()
```

Этот прием также хорошо подходит для совместной работы. Например, если коллеги обращаются к вам за помощью с отладкой, вы можете предложить им вставить вызовы breakpoint в подозрительных местах. Когда после этого вы запустите код на своем компьютере, у вас откроется тот отладчик, который вам больше нравится.

Если вы не знаете, где разместить точку останова, а приложение завершается с необработанным исключением, используйте ретроспективный режим анализа pdb. Следующая команда позволяет запустить сценарий Python в отладочном сеансе, который будет приостановлен в тот момент, когда возникло исключение:

```
python3 -m pdb -c continue script.py
```

Режим разработки

Начиная с версии 3.7, интерпретатор Python можно вызывать в специальном режиме разработки, где действуют дополнительные проверки времени выполнения. Они помогают диагностировать потенциальные проблемы, которые могут возникать при выполнении кода. В правильно работающем коде такие проверки приводят к излишним затратам ресурсов, поэтому по умолчанию они отключены.

Режим разработки можно включить двумя способами:

- Передать при запуске интерпретатора Python параметр командной строки -X dev, например:

```
python -X dev my_application.py
```

- Использовать переменную окружения PYTHONDEVMODE, например:

```
python -X dev my_application.py
```

Вот главные эффекты, действующие в этом режиме:

- Перехватчики выделения памяти: переполнение или опустошение буфера, нарушения API подсистемы выделения памяти, небезопасное использование глобальной блокировки интерпретатора (global interpreter lock, GIL).

- Предупреждения о возможных ошибках при импортировании модулей.
- Предупреждения при некорректпом обращении с ресурсами, папример, если программа не закрывает открытые файлы.
- Предунреждения об элементах стандартной библиотеки, которые считаются устаревшими и будут удалены в будущих версиях.
- Включение обработчика отказов, который выводит трассировку стека приложения, если оно получает системные сигналы `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` или `SIGILL`.

Предупреждения в режиме разработки означают, что что-то ведет себя не так, как должно. Они помогают найти проблемы, которые не обязательно нроявляются как ошибки в ходе нормальной работы кода, но могут привести к ощутимым дефектам в долгосрочной перспективе.

Из-за некорректного освобождения открытых файлов рапо или поздно могут исчерпаться ресурсы среды, в которой выполпяется приложение. Дескрипторы файлов — это такие же ресурсы, как оперативпая память или дисковое пространство. В каждой операционной системе ограничено количество файлов, которые можпо открыть одновременно. Если ваше приложение открывает новые файлы, но не закрывает их, то в какой-то момепт опо пе сможет открыть очередной файл.

Режим разработки позволяет выявлять такие проблемы заранее, поэтому его стоит использовать на этапе тестирования приложения. Однако из-за того что проверки в режиме разработки требуют дополнительных ресурсов, использовать его в средах эксплуатации не рекомендуется.

Иногда в режиме разработки удается диагностировать существующие проблемы. Напримеp, ошибка сегментации — одна из весьма проблемных ситуаций, с которыми может столкнуться ваше приложение. Когда она происходит в Python, вы обычно не нолучаете подробного описания ошибки, а видите лишь краткое сообщение в стандартном выводе командной оболочки:

```
Segmentation fault: 11
```

При ошибке сегментации процесс Python получает системпый сигпал `SIGSEGV` и немедленно завершается. В некоторых операциоппых системах доступен дамп ядра — мгновенный снимок состояния памяти процесса на момент сбоя. Эту информацию можно использовать для отладки приложения. К сожалению, в случае CPython вы получаете снимок памяти процесса интерпретатора, так что отладку придется выполнять на уровне кода C.

В режиме разработки функционирует дополнительный обработчик сбоев, который выводит трассировку стека Python при получении сигнала об ошибке сегмептации. Благодаря этому вы получите немного больше информации о том,

какая часть кода могла привести к проблеме. Вот пример кода, который вызывает ошибку сегментации в Python 3.9:

```
import sys

sys.setrecursionlimit(1 << 30)

def crasher():
    return crasher()

crasher()
```

Если запустить этот код в интерпретаторе Python с флагом `-X dev`, вы получите вывод наподобие такого:

```
Fatal Python error: Segmentation fault

Current thread 0x000000010b04edc0 (most recent call first):
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  ...
```

Обработчик сбоев также можно включить вне режима разработки. Для этого укажите параметр командной строки `-X faulthandler` или присвойте переменной окружения `PYTHONFAULTHANDLER` значение `1`.

Вызвать ошибку сегментации в Python не так просто. Часто такие сбои происходят из-за расширений Python, написанных на C или C++, или из-за функций, вызываемых из совместно используемых библиотек (DLL, `.dylib` или объектов `.so`). Тем не менее существует перечень известных и хорошо документированных условий, при которых эта проблема может возникнуть в коде на чистом Python. Коллекция соответствующих фрагментов кода доступна в репозитории интерпретатора CPython по адресу https://github.com/python/cpython/tree/master/Lib/test/crashers.

Функции `__getattr__()` и `__dir__()` на уровне модуля

Каждый класс Python может определить собственные методы `__getattr__()` и `__dir__()`, чтобы обращаться к динамическим атрибутам объектов. Фупкция `__getattr__()` вызывается, если запрошенпое имя атрибута не найдено; она перехватывает обращение к песуществующему атрибуту и потенциально может

сгенерировать значение «на ходу». Метод `__dir__()` вызывается, когда объект передается функции `dir()`; он должен вернуть список имен атрибутов объекта.

Начиная с Python 3.7, функции `__getattr__()` и `__dir__()` можно определять на уровне модулей. По своей семантике они похожи на методы объектов. Если функция `__getattr__()` определена на уровне модуля, она будет вызываться при неудачном поиске компонента модуля. Функция `__dir__()` будет вызываться, когда объект модуля передается функции `dir()`.

Эта возможность может пригодиться специалистам по сопровождению библиотек, когда классы или функции модуля нужно вывести из употребления. Допустим, вы разместили собственную функцию `get_ci()` (см. раздел «Аннотации обобщенных типов») в библиотеке с открытым кодом под названием dict_helpers.py. Если вы решили переименовать фупкцию в `lookup_ci()`, но при этом хотите сохранить возможность ее импортирования под старым именем, можно воспользоваться следующим паттерном:

```
from typing import Any
from warnings import warn

def ci_lookup(d: dict[str, Any], key: str) -> Any:
    ...

def __getattr__(name: str):
    if name == "get_ci":
        warn(f"{name} is deprecated", DeprecationWarning)
        return ci_lookup

    raise AttributeError(f"module {__name__} has no attribute {name}")
```

Этот код будет выдавать предупреждение `DeprecationWarning` независимо от того, была ли функция `get_ci()` импортирована непосредственно из модуля (например, инструкцией `from dict_helpers import get_ci`) или использовалось прямое обращение к атрибуту `dict_helpers.get_ci`.

Предупреждения о выводе из употребления (depreciation warnings) не отображаются по умолчанию. Их можно включить в режиме разработки.

Форматирование строк с помощью f-строк

F-строки, также называемые строковыми литералами с форматированием, — одна из самых популярных возможностей, появившихся в Python 3.6. Это новый способ форматирования строк, представленный в PEP 498. До Python 3.6 уже существовали два разных метода форматирования строк, так что теперь их стало три:

- Оператор интерполяции `%`: самый старый способ, использующий подстановку в стиле функции `printf()` из стандартной библиотеки C:

```
>>> import math
>>> "approximate value of π: %f" % math.pi
'approximate value of π: 3.141593'
```

- Метод `str.format()`: более удобный и надежный по сравнению с оператором `%`, хотя и не такой компактный. Поддерживает именованные подстановки, а также позволяет многократно использовать одно значение:

```
>>> import math
>>> "approximate value of π: %f" % math.pi
'approximate value of π: 3.141593'
```

- Строковые литералы с форматированием (так называемые f-строки): самый компактный, гибкий и удобный способ форматирования строк. Автоматически заменяет значения в литералах, используя переменные и выражения из локальных пространств имен:

```
>>> import math
>>> f"approximate value of π: {math.pi:f}"
'approximate value of π: 3.141593'
```

Строковые литералы с форматированием обозначаются префиксом `f`, а по синтаксису близки к методу `str.format()`, поскольку используют похожую разметку для полей замены в форматируемом тексте. В методе `str.format()` для подстановки используются позиционные и именованные аргументы. F-строки отличаются тем, что полями замены могут быть произвольные выражения Python, вычисляемые во время выполнения. Внутри строк можно обращаться к любым переменным, доступным в одном пространстве имен с литералом.

При использовании выражений в качестве полей замены код форматирования становится проще и короче. Кроме того, для полей можно применять те же спецификаторы форматирования (для выравнивания, отбивки, вывода знака и т. д.), что и для метода `str.format()`, а синтаксис выглядит так:

```
f"{выражение_поля_замены:спецификатор_формата}"
```

Ниже приведен простой пример кода, который выполняется в интерактивном сеансе и выводит первые десять степеней числа 10 с помощью f-строк, выравнивая результаты отбивкой:

```
>>> for x in range(10):
...     print(f"10^{x} == {10**x:10d}")
...
10^0 ==          1
```

```
10^1 ==          10
10^2 ==         100
10^3 ==        1000
10^4 ==       10000
10^5 ==      100000
10^6 ==     1000000
10^7 ==    10000000
10^8 ==   100000000
10^9 == 1000000000
```

Полная спецификация форматирования строк в Python образует отдельный мини-язык внутри Python. Лучший источник информации об этом языке — официальная документация по адресу https://docs.python.org/3/library/string.html.

Другой полезный интернет-ресурс по этой теме — https://pyformat.info/ — демонстрирует важнейшие элементы этой спецификации на практических примерах.

Символы подчеркивания в числовых литералах

Вероятно, символы подчеркивания в числовых литералах — одна из самых простых в освоении возможностей, однако они еще не завоевали той популярности, которую заслуживают. Начиная с Python 3.6, с помощью символов подчеркивания _ можно разделять разряды в числовых литералах, чтобы большие числа лучше читались. Рассмотрим следующую инструкцию присваивания:

```
account_balance = 100000000
```

С таким количеством нулей трудно с ходу сказать, имеем мы дело с миллионами или миллиардами. С помощью символов подчеркивания можно отделить тысячи, миллионы, миллиарды и т. д.:

```
account_balance = 100_000_000
```

Теперь не придется тщательно подсчитывать нули, чтобы убедиться, что значение `account_balance` равно ста миллионам.

Модуль secrets

Многие программисты разделяют распространенное заблуждение из области безопасности, считая, будто модуль `random` действительно генерирует случайные числа. Это не так, хотя псевдослучайность модуля `random` достаточна для статистических целей. В модуле используется генератор псевдослучайных чисел «вихрь Мерсенна». Он обладает равномерным распределением и достаточно

большой периодичностью, что позволяет использовать его для симуляций, моделирования или численного интегрирования.

Тем не менее «вихрь Мерсенна» — полностью детерминированный алгоритм, как и его реализация в модуле `random`. Это означает, что, зная его начальное условие, то есть значение инициализации, или затравку (seed), можно сгенерировать те же псевдослучайные числа. Более того, если известно достаточное количество последовательных результатов псевдослучайного генератора (в том числе «вихря Мерсенна»), то обычно можно вычислить значение затравки и спрогнозировать следующие результаты.

Если вам интересно, как прогнозировать числа, генерируемые «вихрем Мерсенна», ознакомьтесь со следующим проектом на GitHub: https://github.com/kmyk/mersenne-twister-predictor.

Эта особенность генераторов псевдослучайных чисел означает, что они не годятся для выработки случайных значений там, где важна безопасность. Например, если вам нужно получить случайное значение, которое будет использоваться в качестве пароля или маркера доступа, следует выбрать другой источник случайности.

Модуль `secrets` служит именно для этой цели. Он опирается на лучший источник случайности, доступный в той или иной операционной системе. То есть в системах семейства Unix это будет устройство `/dev/urandom`, а в Windows — генератор `CryptGenRandom`.

Три важнейшие функции модуля `secrets` таковы:

- `secrets.token_bytes(nbytes=None)`: возвращает `nbytes` случайных байтов. Эта функция используется во внутренней реализации `secrets.token_hex()` и `secrets.token_urlsafe()`. Если значение `nbytes` не задано, возвращается количество байтов по умолчанию, которое в документации охарактеризовано как «разумное».
- `secrets.token_hex(nbytes=None)`: возвращает `nbytes` случайных байтов в форме строки в шестнадцатеричном формате, а не объекта `bytes()`. Так как для кодирования одного байта требуются две шестнадцатеричные цифры, итоговая строка будет состоять из `nbytes` × 2 символа. Если значение `nbytes` не указано, функция возвращает то же количество байтов по умолчанию, что и `secrets.token_bytes()`.
- `secrets.token_urlsafe(nbytes=None)`: возвращает `nbytes` случайных байтов в форме строки в кодировке base64, совместимой с форматом URL. Так как один байт занимает в кодировке base64 приблизительно 1,3 символа, полученная строка будет состоять из `nbytes` × 1,3 символа. Если значение `nbytes`

не указано, функция возвращает то же количество байтов по умолчанию, что и `secrets.token_bytes()`.

Еще одна важная функция, о которой часто забывают, — `secrets.compare_digest(a, b)`. Функция сравнивает две строки или два байтоподобных объекта таким образом, чтобы, проанализировав время сравнения, атакующий не мог догадаться, совпадают ли они хотя бы частично. Обычное сравнение строк (с помощью оператора `==`) подвержено так называемой атаке по времени, когда атакующий пытается многократно верифицировать конфиденциальные данные и за счет статистического анализа постепенно подобрать последовательные символы исходного значения.

Чего стоит ожидать в будущем?

На момент написания книги версия Python 3.9 существовала всего несколько месяцев, но скорее всего, когда вы будете читать книгу, уже выйдет Python 3.10 или более новые версии.

Открытые и прозрачные процессы разработки языка Python позволяют постоянно следить за новшествами, которые утверждены в документах PEP и уже реализованы в альфа- и бета-выпусках. Благодаря этому можно представить некоторые возможности, которые появятся в Python 3.10. Ниже приведена краткая сводка важнейших изменений, ожидающихся в ближайшем будущем.

Оператор | для объединения типов

В Python 3.10 появится еще одно упрощение синтаксиса для аннотаций типов. С новым синтаксисом станет легче составлять аннотации для объединений типов.

В Python используется динамическая типизация и нет полиморфизма. В результате функция вполне может в качестве одного и того же аргумента принимать данные разных типов в зависимости от вызова и правильно обрабатывать их, если эти типы имеют одинаковый интерфейс. Чтобы лучше понять это, вспомним сигнатуру функции, которая перебирает значения из словаря со строковыми ключами без учета регистра символов:

```
def get_ci(d: dict[str, Any], key: str) -> Any: ...
```

В реализации функции мы использовали метод `upper()` для ключей, полученных из словаря. И это главная причина, по которой для аргумента `d` мы указали тип `dict[str, Any]`, а для аргумента `key` — тип `str`.

Однако `str` — не единственный встроенный тип, у которого есть метод `upper()`. Тип `bytes` тоже поддерживает этот метод. Если вы хотите, чтобы функция `get_ci()` могла получать словари как со строковыми, так и с байтовыми ключами, нужно указать объединение возможных типов.

В настоящее время объединение типов можно задавать только аннотацией `typing.Union`. Она позволяет объединить типы `bytes` и `str` в виде `typing.Union [str, bytes]`. При этом полная сигнатура функции `get_ci()` выглядит так:

```
def get_ci(
    d: dict[Union[str, bytes], Any],
    key: Union[str, bytes]
) -> Any:
    ...
```

Запись и так довольно громоздкая, а для более сложных функций она получится еще тяжеловеснее. Однако начиная с Python 3.10, для объединения тинов можно будет использовать оператор |. В будущем код можно будет писать так:

```
def get_ci(d: dict[str | bytes, Any], key: str | bytes) -> Any: ...
```

В отличие от аннотаций обобщенных типов, аннотации `typing.Union` не будут объявлены устаревшими с появлением оператора объединения типов. Это значит, что можно будет взаимозаменяемо использовать оба механизма.

Структурное сопоставление с шаблоном

Структурное сопоставление с шаблоном (structural pattern matching) определенно стало не только самым неоднозначным нововведением Python за последнее десятилетие, но и самым сложным.

Утверждению этой функциональности предшествовали жаркие обсуждения и бесчисленные предварительные проекты. Сложность темы наглядно проявляется, если взглянуть на документы PEP, которые пытались решить проблему. В этой таблице перечислены все PEP, относящиеся к структурному сопоставлению (статусы приводятся по состоянию на март 2021 года):

Дата	PEP	Название	Тип	Статус
23 июня 2020-го	622	Структурное сопоставление с шаблоном	Стандарт	Заменен PEP 634
12 сентября 2020-го	634	Структурное сопоставление с шаблоном: спецификация	Стандарт	Принят

(*окончание*)

Дата	PEP	Названиие	Тип	Статус
12 сентября 2020-го	635	Структурное сопоставление с шаблоном: мотивация и обоснования	Информационный	Окончательный
12 сентября 2020-го	636	Структурное сопоставление с шаблоном: учебное руководство	Ипформационный	Окончательный
26 сентября 2020-го	642	Явный синтаксис шаблонов для структурного сопоставлепия	Стандарт	Черповик
21 февраля 2021-го	653	Точная семантика для сопоставления с шаблоном	Стандарт	Черновик

Документов много, и все они весьма объемные. Что же такое структурное сопоставлепие с шаблоном и какая от него польза?

Структурное сопоставление вводит **ииструкцию match** и два новых «мягких» ключевых слова: `match` и `case`. Инструкция сопоставляет заданное значение со списком вариантов и выполняет те или иные действия в зависимости от обнаруженных совпадений.

«Мягким» (soft) ключевым словом называется ключевое слово, которое зарезервировано не во всех контекстах. Вне контекста инструкции match идентификаторы match и case можно использовать как имена обычных переменных или функций.

Некоторым программистам инструкция `match` по синтаксису напоминает `switch` из таких языков, как C, C++, Pascal, Java и Go. С ее помощью действительно можно реализовать тот же паттерн программирования, но она гораздо мощнее.

В общем виде (упрощенно) синтаксис инструкции `match` выглядит так:

```
match выражение:
    case шаблон:
        ...
```

Выражением может быть любое действительное выражение Python, а *шаблон*, который применяется для сопоставления, — это новая концепция в Python.

Блок `case` может содержать несколько инструкций. Сложность инструкции `match` обусловлена в основном тем, что в ней используются шаблоны сопоставления, которые поначалу трудно понять. Шаблоны также легко спутать с выражениями, но в отличие от обычных выражений, у них нет вычисляемых значений.

Прежде чем углубляться в подробности сопоставления с шаблоном, рассмотрим простой пример инструкции `match`, который воспроизводит функциональность команд `switch` из других языков программирования:

```
import sys

match sys.platform:
    case "windows":
        print("Running on Windows")
    case "darwin":
        print("Running on macOS")
    case "linux":
        print("Running on Linux")
    case _:
        raise NotImplementedError(
            f"{sys.platform} not supported!"
        )
```

Хотя это очень примитивный пример, он демонстрирует ряд важных элементов. Во-первых, литералы можно использовать в качестве шаблонов. Во-вторых, существует специальный универсальный шаблон _ (подчеркивание). Универсальные шаблоны и другие шаблоны, которые по своему синтаксису совпадают всегда, создают неопровержимый блок `case`. Такой блок может быть только последним блоком инструкции `match`.

Конечно, этот пример можно реализовать простой цепочкой инструкций `if`, `elif` и `else`. На собеседовании при приеме на работу соискателям часто предлагают написать программу FizzBuzz, которая перебирает числовой диапазон от 0 до произвольного числа и в зависимости от значения выполняет следующие действия:

- Если значение делится на 3, выводится строка `Fizz`.
- Если значение делится на 5, выводится строка `Buzz`.
- Если значение делится на 3 и 5, выводится строка `FizzBuzz`.
- Во всех остальных случаях выводится значение.

По сути, это простая задача, но вы не поверите, сколько людей может споткнуться даже на простейших задачах из-за волнения на собеседовании. Конечно, можно добиться результата несколькими командами `if`, но использование `match` придает решению некоторую естественную элегантность:

```
for i in range(100):
    match (i % 3, i % 5):
        case (0, 0): print("FizzBuzz")
        case (0, _): print("Fizz")
        case (_, 0): print("Buzz")
        case _: print(i)
```

В этом примере на каждой итерации цикла происходит сопоставление с шаблоном `(i % 3, i % 5)`. Приходится выполнять два целочисленных деления, потому что результат каждой итерации зависит от обоих результатов деления. Инструкция `match` сопоставляет шаблоны только до тех пор, пока не обнаружит первый совпадающий блок, и только этот блок будет выполнен.

Обратите внимание на принципиальное отличие от предыдущего примера: вместо шаблонов литералов здесь используются шаблоны последовательностей:

- Шаблон `(0, 0)`: совпадает с последовательностью из двух элементов, если оба элемента равны 0.
- Шаблон `(0, _)`: совпадает с последовательностью из двух элементов, если первый элемент равен 0. Второй элемент может иметь любое значение и тип.
- Шаблон `(_, 0)`: совпадает с последовательностью из двух элементов, если второй элемент равен 0. Первый элемент может иметь любое значение и тип.
- Шаблон `_`: универсальный шаблон, совпадающий с любым значением.

В выражении после `match` можно использовать не только простые литералы и последовательности литералов. Можно сопоставлять шаблоны с конкретными классами и даже с шаблонами классов — и здесь начинается настоящее волшебство. Несомненно, это самая сложная часть новой функциональности.

Python 3.10 еще не вышел на момент написания книги, так что трудно продемонстрировать характерный и практичный сценарий применения шаблонов сопоставления с классами. Поэтому рассмотрим пример из официального учебника. Ниже приведен измененный код из документа PEP 636 с простой функцией `where_is()`, которая выполняет сопоставление со структурой экземпляра класса `Point`:

```
class Point:
    x: int
    y: int

    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
```

```
            print("Начало координат")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Где-то еще")
        case _:
            print("Это не точка")
```

В этом примере происходит много интересного, поэтому разберем все шаблоны, которые в нем встречаются:

- `Point(x=0, y=0)`: совпадает, если `point` является экземпляром класса `Point`, а оба его атрибута, `x` и `y`, равны 0.
- `Point(x=0, y=y)`: совпадает, если `point` является экземпляром класса `Point`, а его атрибут `x` равен 0. Атрибут `y` сохраняется в переменной `y`, которую можно использовать в блоке `case`.
- `Point(x=x, y=0)`: совнадает, если `point` является экземпляром класса `Point`, а его атрибут `y` равен 0. Атрибут `x` сохраняется в переменной `x`, которую можно использовать в блоке `case`.
- `Point()`: совпадает, если `point` является экземпляром класса `Point`.
- `_`: совпадает всегда.

Как видите, сопоставление с шаблоном умеет проверять атрибуты объектов. Хотя шаблон `Point(x=0, y=0)` выглядит как вызов конструктора, Python не вызывает конструктор объекта при обработке шаблонов. Python также не проверяет аргументы и именованные аргументы методов `__init__()`, так что в шаблоне сопоставления можно обратиться к значению любого атрибута.

В шаблонах также можно использовать синтаксис «позиционных атрибутов», но этот способ чуть более трудоемкий. Вам придется объявить дополнительный атрибут класса `__match_args__`, который задает естественный порядок атрибутов экземпляра класса, как в следующем примере:

```
class Point:
    x: int
    y: int

    __match_args__ = ["x", "y"]

    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
```

```
case Point(0, 0):
    print("Начало координат")
case Point(0, y):
    print(f"Y={y}")
case Point(x, 0):
    print(f"X={x}")
case Point():
    print("Где-то еще")
case _:
    print("Это не точка")
```

И это лишь верхушка айсберга. Инструкции сопоставления с шаблоном в действительности гораздо сложнее, чем показано в этом коротком разделе. Если бы мы рассматривали все потенциальные сценарии использования, варианты синтаксиса и граничные случаи, для этого потребовалась бы целая глава. Чтобы подробнее погрузиться в эту тему, вам определенно стоит прочитать три «канонических» PEP: 634, 635 и 636.

Итоги

В этой главе рассматриваются важнейшие изменения в синтаксисе языка и стандартной библиотеке, произошедшие за последние четыре версии Python. Если вы не следите за примечаниями к выпускам Python или еще не перешли на Python 3.9, эта глава содержит достаточно информации для того, чтобы вы оставались в курсе дела.

В этой главе вы также познакомились с концепцией идиом программирования, с которой еще неоднократно встретитесь в этой книге. В следующей главе мы более внимательно рассмотрим многие идиомы Python, сравнивая отдельные возможности языка с другими языками программирования. Если вы — опытный программист, недавно перешедший на Python, это поможет вам разобраться, как добиваться тех или иных результатов «по-питонически». Также вы увидите, в чем заключаются самые сильные стороны Python, а в чем он еще отстает от конкурентов.

4

Python в сравнении с другими языками

Многие программисты приходят в Python с опытом работы на других языках. Нередко они уже знакомы с идиомами этих языков и пытаются воспроизводить их на Python. Поскольку каждый язык программирования уникален, чрезмерное увлечение такими чужеродными идиомами нередко ведет к чересчур пространному или неоптимальному коду.

Классический пример чужеродной идиомы, которую часто применяют неопытные программисты, — обход списков. Программист, знакомый с массивами по языку C, мог бы написать примерно такой код Python:

```
for index in range(len(some_list)):
    print(some_list[index])
```

Между тем опытный программист на Python скорее напишет:

```
for item in some_list:
    print(item)
```

Языки программирования часто классифицируются по парадигмам, которые можно рассматривать как целостный набор средств, поддерживающих определенные стили программирования. Python — мультипарадигменный язык, поэтому он в чем-то похож на многие другие языки программирования. Как следствие, код на Python можно писать и структурировать почти так же, как это делается в Java, C++ или любом другом популярном языке.

Однако такой подход часто уступает по эффективности общепринятым паттернам Python. Знание «родных» идиом языка поможет вам писать более понятный и эффективный код.

Эта глава предназначена для программистов, имеющих опыт работы на других языках. Здесь дается обзор некоторых важных возможностей Python и рассматриваются идиоматические способы решения типичных задач. Мы сравним характерные особенности Python и других языков, а также перечислим стандартные ловушки, которые поджидают опытных программистов, только начинающих свое путешествие в мир Python. По ходу главы рассмотрим следующие темы:

- Модель классов и объектно-ориентированное программирование.
- Динамический полиморфизм.
- Классы данных.
- Функциональное программирование.
- Перечисления.

Начнем с технических требований.

Технические требования

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%204.

Модель классов и объектно-ориентированное программирование

На практике в языке Python преобладает парадигма **объектно-ориентированного программирования** (ООП). Центральное место в ней занимают объекты, которые инкапсулируют данные (в форме атрибутов объектов) и поведение (в форме методов). Пожалуй, ООП можно назвать одной из самых разносторонних парадигм. За многолетнюю историю программирования в ООП сформировался широкий спектр всевозможных стилей, направлений и реализаций. Создатели Python вдохновлялись опытом многих других языков, поэтому в этом разделе реализация ООП в Python рассматривается сквозь призму разных языков программирования.

Чтобы упростить повторное использование кода, улучшить его расширяемость и модульность, объектно-ориентированные языки обычно предоставляют сред-

ства композиции или наследования классов. В этом отношении Python не отличается от других языков и тоже поддерживает подклассы тинов.

Возможно, в Python не такой богатый ассортимент средств ООП, как в других объектно-ориентированных языках, но в нем используется довольно гибкая модель данных и классов, которая позволяет весьма элегантно реализовать большинство паттернов ООП. Кроме того, все сущности в Python являются объектами, включая функции, определения классов и такие базовые значения, как числа (целые и с плавающей точкой), логические значения и строки.

Если говорить о том, какие еще популярные языки программирования похожи на Python синтаксическими средствами ООП и моделью данных, то одним из ближайших аналогов, вероятно, будет Kotlin — язык, выполняемый в основном на виртуальной машине Java (JVM). Ниже перечислены некоторые общие черты Kotlin и Python:

- Удобный механизм вызова методов надклассов: для явного обращения к методам или атрибутам надклассов в Kotlin есть ключевое слово `super`, а в Python — функция `super()`.
- Самореферентность, то есть возможность объектов ссылаться на самих себя: в Kotlin выражение `this` всегда указывает на текущий объект класса. В Python первый аргумент метода всегда является ссылкой на экземпляр. По общепринятому соглашению ему присваивается имя `self`.
- Возможность создавать классы данных: подобно Python, Kotlin поддерживает классы данных как «синтаксический сахар» поверх традиционных определений классов. С этим механизмом легче создавать структуры данных, которые основаны на классах, но не предполагают сложного поведения.
- Концепция свойств: Kotlin позволяет определять методы чтения и записи для свойств класса в виде функций. Python предоставляет декоратор `property()` с аналогичным назначением, а также механизм дескрипторов, позволяющих полностью настроить доступ к атрибутам объектов.

На фоне других реализаций ООП в языках программирования Python выделяется своим подходом к наследованию. В отличие от Kotlin и многих других языков, Python поддерживает множественное наследование (хотя злоупотреблять им не рекомендуется). В прочих языках оно нередко отсутствует или поддерживается с ограничениями. Еще одно важное отличие Python от других языков — отсутствие ключевых слов наподобие `private`/`public`, которые управляют доступом к внутренним атрибутам объекта за пределами определения класса.

Рассмотрим более подробно возможность, которая роднит Python с Kotlin и рядом других языков на базе JVM, — доступ к надклассам с помощью вызова `super()`.

Доступ к надклассам

В объектно-ориентированных языках есть много механизмов инкапсуляции поведения объектов, но один из самых распространенных способов — использование классов. Реализация ООП в Python основана именно на концепции классов и подклассов.

Наследование путем создания подклассов — удобный механизм повторного использования классов, позволяющий расширить или специализировать их поведение. Подклассы часто опираются на поведение своих базовых классов, но наращивают их дополнительными методами или предоставляют совершенно новые реализации существующих методов, заменяя их определения.

Однако переопределение методов никак не облегчит повторное использование кода, если их исходные реализации из надкласса недоступны. Поэтому в Python есть функция `super()`, которая возвращает объект-посредник для доступа к реализациям методов всех базовых классов. Чтобы лучше понять возможности функции `super()`, представьте, что вы хотите наследовать тип словаря Python так, чтобы доступ к ключам выполнялся без учета регистра. Это может пригодиться, например, чтобы хранить заголовки протокола HTTP, так как, согласно спецификации HTTP, в именах заголовков регистр символов не учитывается.

Вот простой пример реализации такой структуры на Python с помощью подклассов:

```
from collections import UserDict
from typing import Any

class CaseInsensitiveDict(UserDict):
    def __setitem__(self, key: str, value: Any):
        return super().__setitem__(key.lower(), value)

    def __getitem__(self, key: str) -> Any:
        return super().__getitem__(key.lower())

    def __delitem__(self, key: str) -> None:
        return super().__delitem__(key.lower())
```

Наша реализация `CaseInsensitiveDict` основана на `collections.UserDict` вместо встроенного типа `dict`. Теоретически наследование от типа `dict` возможно, но оно чревато нестыковками, потому что встроенный тип `dict` не всегда вызывает метод `__setitem__()`, чтобы обновлять свое состояние. Что еще важнее, этот метод не будет использоваться при инициализации объектов и вызовах методов `update()`. Аналогичные проблемы могут возникнуть и при наследовании типа `list`. Поэтому подклассы `dict` настоятельно рекомендуется создавать на

основе класса `collections.UserDict`, а подклассы `list` — на основе `collections.UserList`.

Основное поведение модифицированного словаря сосредоточено в `__getitem__(self, item: str)` и `__setitem__(self, key: str, value: Any)`. Эти методы отвечают соответственно за доступ к элементам словарей в синтаксисе `словарь[ключ]` и за присвоение значений элементам в синтаксисе `словарь[ключ] = значение`. Аннотации типов позволяют указать, что ключи должны быть строками, а значения могут иметь произвольный тип Python.

Метод `__setitem__()` позволяет сохранять и изменять значения в словарях. Было бы нелогично наследовать базовый тип словаря, но не пользоваться его внутренним механизмом хранения пар «ключ — значение». Именно поэтому мы применяем `super().__setitem__()`, чтобы вызвать исходную реализацию соответствующего метода. Но прежде чем сохранить значение, мы преобразуем ключ к нижнему регистру методом `str.lower()`. Это гарантирует, что все ключи в словаре всегда будут храниться в нижнем регистре.

Реализация `__getitem__()` устроена аналогично `__setitem__()`. Мы знаем, что перед сохранением в словаре каждый ключ преобразуется к нижнему регистру. Значит, при поиске ключ тоже можно преобразовать к пижнему регистру. Если реализация метода `__getitem__()` из надкласса не возвращает результат, это говорит о том, что в словаре нет элемента с заданным ключом без учета регистра.

Наконец, метод `__delitem__()` удаляет существующие ключи словаря. В нем используется тот же прием: ключ нреобразуется к нижнему регистру, затем вызывается реализация надкласса. Это позволяет удалять ключи из словаря инструкцией `del dictionary[key]`.

Следующий фрагмент демонстрирует поиск по ключу без учета регистра:

```
>>> headers = CaseInsensitiveDict({
...     "Content-Length": 30,
...     "Content-Type": "application/json",
... })
>>> headers["CONTENT-LENGTH"]
30
>>> headers["content-type"]
'application/json'
```

Этот пример использования `super()` достаточно прост и понятен, по при множественном наследовании ситуация заметпо усложняется. Python поддерживает множественное наследование с помощью такой концепции, как порядок разрешения методов, или **MRO** (Method Resolution Order). Этот механизм более подробно рассматривается в следующем разделе.

Множественное наследование и порядок разрешения методов

Порядок разрешения методов (MRO) в Python основан на **C3-линиеаризации** — детерминированном алгоритме, изначально созданном для языка программирования Dylan. Алгоритм C3 строит линейное представление класса — упорядоченный список его предков. Этот список используется для поиска атрибутов по дереву наследования класса.

Дополнительную информацию о языке Dylan можно найти по адресу http://opendylan.org. В Википедии есть превосходная статья о C3-линеаризации, доступная по адресу https://en.wikipedia.org/wiki/C3_linearization.

C3-линеаризация используется в качестве MRO языка Python не с самого начала. Она появилась в Python 2.3 вместе с общим базовым типом для всех объектов (то есть типом `object`). До перехода на C3-линеаризацию дела обстояли иначе: если у класса были два предка (рис. 4.1), то порядок разрешения методов легко вычислялся только для простых ситуаций без каскадного множественного наследования.

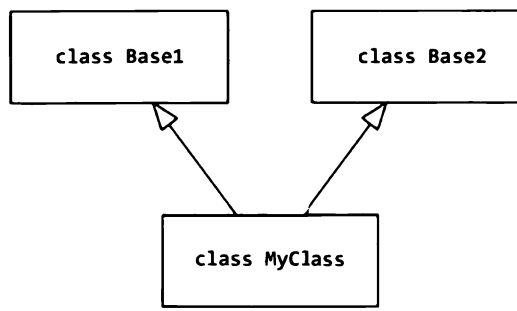

Рис. 4.1. Иерархия классов

Вот пример простого множественного наследования, которое не требует специального алгоритма MRO:

```
class Base1:
    pass


class Base2:
    def method(self):
        print("Base2.method() called")

class MyClass(Base1, Base2):
    pass
```

До выхода Python 2.3 здесь применялся бы простой поиск в глубину по дереву иерархии классов. Другими словами, при вызове `MyClass().method()` интерпретатор ищет метод сначала в `MyClass`, затем в `Base1` и, наконец, паходит его в `Base2`.

Когда па вершине иерархии классов появляется класс `CommonBase` (рис. 4.2), ситуация усложняется:

```
class CommonBase:
    pass

class Base1(CommonBase):
    pass

class Base2(CommonBase):
    def method(self):
        print("Base2.method() called")

class MyClass(Base1, Base2):
    pass
```

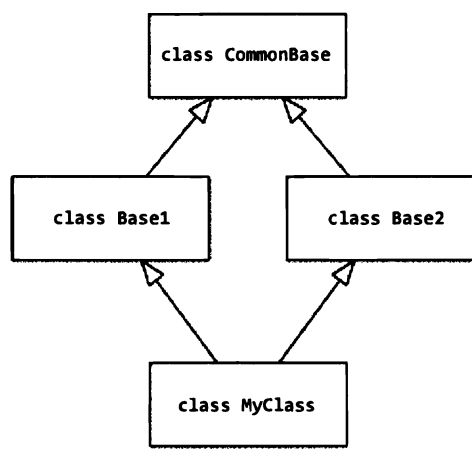

Рис. 4.2. Ромбовидная иерархия классов

В такой ситуации простой порядок разрешепия, который обходит дерево по принципу «в глубину слева направо», достигнет вершины иерархии через базовый класс `Base1`, прежде чем рассматривать класс `Base2`. Алгоритм вернет результат, который противоречит нашей интуиции. Без C3-линеаризации будет выполпяться не тот метод, который находится ближе всего в дереве наследования.

Такой сценарий наследования, называемый ромбовидной иерархией классов, нетипичен в массовой практике разработки. В стандартной библиотеке подобное

наследование обычно не применяется, и многие программисты считают его нежелательным. Тем не менее в Python такая возможность существует, а значит, для нее нужна четко определенная и понятная стратегия обработки.

Кроме того, начиная с Python 2.3, на вершине иерархии типов классов находится класс `object`. По сути, каждый класс стал частью большой ромбовидной иерархии классов, для которой потребовался механизм разрешения методов также и на стороне реализации Python в языке C. Поэтому в качестве алгоритма MRO в Python теперь используется C3-линеаризация.

В Python 2 классы, наследующие типу `object`, назывались классами нового стиля. Неявное наследование от `object` не использовалось. В Python 3 все классы относятся к новому стилю, а классы старого стиля недоступны.

В исходном справочном документе Python MRO, написанном Мишелем Симионато (Michele Simionato), линеаризация описывается так:

> *Линеаризация C представляет собой сумму C и результата слияния списка линеаризаций родителей со списком родителей.*

Справочный документ Мишеля Симионато с подробным описанием Python MRO доступен по адресу http://www.python.org/download/releases/2.3/mro.

Это попросту означает, что алгоритм C3 — рекурсивный. В символической записи применение C3 в предыдущем примере с наследованием выглядит так:

```
L[MyClass(Base1, Base2)] =
        [MyClass] + merge(L[Base1], L[Base2], [Base1, Base2])
```

Здесь `L[MyClass]` — линеаризация `MyClass`, а `merge` — конкретный алгоритм, выполняющий слияние нескольких результатов линеаризации.

Алгоритм слияния отвечает за то, чтобы в списке не было дубликатов и сохранялся правильный порядок. При этом используется концепция **головы** (начала) и **хвоста** (конца) списка. Головой считается первый элемент списка, а хвостом — его остальная часть, следующая за головой. Симионато описывает алгоритм слияния следующим образом (в адаптации для нашего примера):

> *Возьмем голову первого списка, то есть L[Base1][0]; если эта голова не входит в хвост любого другого списка, добавляем ее в линеаризацию MyClass и удаляем из списков в слиянии. В противном случае переходим к голове следующего списка и берем ее, если это подходящая голова.*

Затем операция повторяется до тех пор, пока либо все классы не будут удалены, либо не удастся найти следующую подходящую голову. Во втором случае построить слияние невозможно; Python 2.3 не сможет создать класс MyClass и выдаст исключение.

Иначе говоря, C3 выполняет рекурсивный ноиск в глубину для каждого родителя, чтобы получить последовательность списков. Затем производится слияние всех снисков слева направо с устранением неоднозначностей в иерархии, если класс оказался в нескольких снисках.

Если бы нам пришлось вручную вычислять MRO для `MyClass` в символическом виде, то на первом этапе пришлось бы разложить все линеаризации `L[class]`:

```
L[MyClass]
  = [MyClass] + merge(L[Base1], L[Base2], [Base1, Base2])]
  = [MyClass] + merge(
      [Base1 + merge(L[CommonBase], [CommonBase])],
      [Base2 + merge(L[CommonBase], [CommonBase])],
      [Base1, Base2]
    )
  = [MyClass] + merge(
      [Base1] + merge(L[CommonBase], [CommonBase]),
      [Base2] + merge(L[CommonBase], [CommonBase]),
      [Base1, Base2]
    )
  = [MyClass] + merge(
      [Base1] + merge([CommonBase] + merge(L[object]), [CommonBase]),
      [Base2] + merge([CommonBase] + merge(L[object]), [CommonBase]),
      [Base1, Base2]
    )
```

У класса `object` нет других предков, поэтому его C3-линеаризация представляет собой список из одного элемента `[object]`. Значит, на следующем этапе развернем `merge([object])` в `[object]`:

```
  = [MyClass] + merge(
    [Base1] + merge([CommonBase] + merge([object]), [CommonBase]),
    [Base2] + merge([CommonBase] + merge([object]), [CommonBase]),
    [Base1, Base2]
  )
```

`merge([object])` — это список из одного элемента, поэтому он немедленно разворачивается в `[object]`:

```
  = [MyClass] + merge(
    [Base1] + merge([CommonBase, object], [CommonBase]),
    [Base2] + merge([CommonBase, object], [CommonBase]),
    [Base1, Base2]
  )
```

Теперь нужно разложить `merge([CommonBase, object], [CommonBase])`. Головой первого списка является класс `CommonBase`. Он не входит в хвост какого-либо из остальных списков, поэтому его можно немедленно вынести из слияния во внешний результат линеаризации:

```
= [MyClass] + merge(
  [Base1, CommonBase] + merge([object]),
  [Base2, CommonBase] + merge([object]),
  [Base1, Base2]
)
```

Во внутренних слияниях снова остается только `merge([object])`, который по-прежнему раскладывается:

```
= [MyClass] + merge(
  [Base1, CommonBase, object],
  [Base2, CommonBase, object],
  [Base1, Base2]
)
```

Остается последнее слияние, которое оказывается нетривиальным. Первая голова `Base1` не входит в хвосты других списков. Ее можно вынести из слияния:

```
= [MyClass, Base1] + merge(
  [CommonBase, object],
  [Base2, CommonBase, object],
  [Base2]
)
```

Теперь первой головой оказывается класс `CommonBase`, который входит в хвост второго списка `[Base2, CommonBase, object]`. Это означает, что прямо сейчас его пельзя обработать и нужно перейти к следующей голове, то есть `Base2`. Она не входит в хвосты других списков. `Base2` можно выпести из слияния во внешпий результат линеаризации:

```
= [MyClass, Base1, Base2] + merge(
  [CommonBase, object],
  [CommonBase, object],
  []
)
```

`CommonBase` снова остается первой головой, но на этот раз не входит в хвосты других списков. Теперь этот класс можно тоже вынести из слияния:

```
= [MyClass, Base1, Base2, CommonBase] + merge(
  [object],
  [object],
  []
)
```

Последний шаг `merge([object], [object], [])` тривиален. Конечный результат линеаризации выглядит так:

```
[MyClass, Base1, Base2, CommonBase, object]
```

Результаты линеаризации C3 можно легко просмотреть в атрибуте `__mro__` любого класса. В следующем фрагменте показан вычисленный результат MRO для класса `MyClass`:

```
>>> MyClass.__mro__
(<class '__main__.MyClass'>, <class '__main__.Base1'>, <class '__
main__.Base2'>, <class '__main__.CommonBase'>, <class 'object'>)
```

Атрибут `__mro__` класса (доступный только для чтения) содержит результат C3-линеаризации и вычисляется при загрузке определения класса. Чтобы получить это же значение, можно вызвать метод `MyClass.mro()`.

Инициализация экземпляров класса

Объектом в ООП называется сущность, которая инкапсулирует данные вместе с поведением. В Python данные хранятся в атрибутах объектов: по сути, это просто переменные объектов. Поведение представляется методами. Это типично почти для всех объектно-ориентированных языков, хотя конкретная терминология может различаться. Например, в C++ и Java данные объектов хранятся в полях, а в Kotlin — с помощью свойств (хотя они устроены сложнее, чем обычные переменные объектов).

От объектно-ориентированных языков со статической типизацией Python отличается подходом к объявлению и инициализации атрибутов объектов. В двух словах, классы Python не требуют определять атрибуты в теле класса. Переменные начинают свое существование в момент их инициализации. Именно по этой причине канонический способ объявления атрибутов объектов состоит в том, чтобы присваивать им значения во время инициализации объекта в методе `__init__()`:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Такая схема может сбивать с толку разработчиков, приходящих в Python с опытом программирования на языках со статической типизацией. В таких языках объявления полей объекта обычно статичны и находятся вне функции инициализации объекта. Вот почему программисты, привыкшие к C++ или Java, часто

склонны воспроизводить этот паттерн, присваивая значения по умолчанию как атрибуты класса в теле класса:

```
class Point:
    x = 0
    y = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Этот код — классический образец того, как идиома другого языка воспроизводится на Python. Прежде всего она избыточна: значения атрибутов класса всегда перекрываются атрибутами объектов при инициализации. Но эта идиома также является опасным примером «кода с душком»: она может привести к коварным ошибкам, если кто-то захочет назначить атрибутом класса изменяемый тип, такой как `list` или `dict`.

«Код с душком» (code smell) — симптом, который может указывать на более глубокую проблему. Бывает, что тот или иной фрагмент кода функционально правилен и не содержит ошибок, но при этом оказывается причиной будущих проблем. «Код с душком» — это обычно небольшие дефекты архитектуры или небезопасные конструкции, которые провоцируют ошибки.

Проблема возникает оттого, что атрибуты класса (атрибуты, назначенные вне тела метода) связываются с объектами тина, а не с экземплярами типа. При обращении к атрибуту в форме `self.attribute` Python сначала ищет значение, связанное с имепем `attribute`, в пространстве имен экземпляра класса. Если найти значение не удалось, поиск продолжается в пространстве имен типа класса. Однако когда атрибуту `self.attribute` присваивается значение в методе класса, Python обрабатывает это совершенно иначе: новые значения всегда присваиваются в пространстве имен экземпляра класса. Это создает особенно много проблем с изменяемыми типами, потому что может вызвать случайную утечку состояния объекта между экземплярами класса.

Поскольку использовать изменяемые типы как атрибуты класса, а не атрибуты экземпляра считается дурным тоном, трудно привести реалистичные примеры кода. Впрочем, это не помешает продемонстрировать проблему в общих чертах. Рассмотрим следующий класс, который накапливает значения в списке и отслеживает последнее значение:

```
class Aggregator:
    all_aggregated = []
    last_aggregated = None
```

```
    def aggregate(self, value):
        self.last_aggregated = value
        self.all_aggregated.append(value)
```

Чтобы понять, где скрывается проблема, запустим интерактивный сеанс, создадим два разных агрегатора и начнем наканливать элементы:

```
>>> a1 = Aggregator()
>>> a2 = Aggregator()
>>> a1.aggregate("a1-1")
>>> a1.aggregate("a1-2")
>>> a2.aggregate("a2-1")
```

Если тенерь взглянуть на сниски обоих экземпляров, мы увидим довольно странный вывод:

```
>> a1.all_aggregated
['a1-1', 'a1-2', 'a2-1']
>>> a2.all_aggregated
['a1-1', 'a1-2', 'a2-1']
```

Читая код, можно подумать, что каждый экземпляр **Aggregator** должен отслеживать историю своих собственных накоплений. Но вместо этого мы видим, что все экземпляры **Aggregator** совместно используют одно и то же состояние атрибута **all_aggregated**. С другой стороны, если запросить последние накопленные значения, мы увидим нравильные результаты для обоих агрегаторов:

```
>> a1.all_aggregated
['a1-1', 'a1-2', 'a2-1']
>>> a2.all_aggregated
['a1-1', 'a1-2', 'a2-1']
```

В нодобных ситуациях загадка легко разгадывается, если просмотреть значения несвязанных атрибутов класса:

```
>> a1.all_aggregated
['a1-1', 'a1-2', 'a2-1']
>>> a2.all_aggregated
['a1-1', 'a1-2', 'a2-1']
```

Как видно из этого примера, все экземпляры **Aggregator** совместно используют одно и то же состояние благодаря изменяемому атрибуту **Aggregator.all_aggregated**. Иногда именно это и требуется, но гораздо чаще это тиничная ошибка, которую не всегда легко обпаружить. Из-за этого все значения атрибутов, которые должны быть уникальными для каждого экземпляра класса, нужно ипициализировать только в методе **__init__()**.

Исправленная версия класса `Aggregator` выглядит так:

```
class Aggregator:
    def __init__(self):
        self.all_aggregated = []
        self.last_aggregated = None

    def aggregate(self, value):
        self.last_aggregated = value
        self.all_aggregated.append(value)
```

Мы просто перенесли инициализацию атрибутов `all_aggregated` и `last_aggregated` в метод `__init__()`. Теперь повторим вызовы инициализации и накопления данных из предыдущего примера:

```
>>> a1 = Aggregator()
>>> a2 = Aggregator()
>>> a1.aggregate("a1-1")
>>> a1.aggregate("a1-2")
>>> a2.aggregate("a2-1")
```

Если после этого просмотреть состояние экземпляров `Aggregator`, мы увидим, что накопления в них отслеживаются независимо друг от друга:

```
>>> a1.all_aggregated
['a1-1', 'a1-2']
>>> a2.all_aggregated
['a2-1']
```

А если вам так уж необходимо объявить все атрибуты в начале определения класса, используйте аннотации типов, как в следующем примере:

```
from typing import Any, List

class Aggregator:
    all_aggregated: List[Any]
    last_aggregated: Any

    def __init__(self):
        self.all_aggregated = []
        self.last_aggregated = None

    def aggregate(self, value: Any):
        self.last_aggregated = value
        self.all_aggregated.append(value)
```

Аннотировать атрибуты классов — на самом деле не дурной тон. Это может пригодиться средствам статической проверки типов или IDE, чтобы помочь повысить качество кода, а также эффективнее выразить предполагаемое ис-

пользование класса и возможные ограничения типов. Аннотации атрибутов класса также упрощают инициализацию классов данных, о которых мы поговорим в разделе «Классы данных».

Паттерны обращения к атрибутам

Еще одно отличие Python от объектно-ориептированных языков со статической типизацией — отсутствие концепции открытых, частных и защищенных члепов класса. В других языках эти категории часто используются, чтобы ограничить или открыть доступ к атрибутам объектов из кода за пределами класса. Ближайший аналог этой концепции в Python — декорирование имен (name mangling). Каждый раз, когда интерпретатор встречает атрибут с префиксом __ (двойное подчеркивание) в теле класса, он переименовывает этот атрибут «на ходу»:

```
class MyClass:
    def __init__(self):
        self.__secret_value = 1
```

Паттерн двойного подчеркивания называется «dunder» (double underscores). Подробнее см. в разделе «Двойные подчеркивания (языковые протоколы)».

Если обратиться к атрибуту `__secret_value` по его исходному имени за пределами класса, возникает исключение `AttributeError`:

```
>>> instance_of = MyClass()
>>> instance_of.__secret_value
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__secret_value'
>>> instance_of._MyClass__secret_value
1
```

Можно подумать, будто этот механизм аналогичен частным/защищенным полям и методам, которые обычно встречаются в других объектно-ориентированных языках. Действительно, он усложняет доступ к атрибутам извне класса, но не запрещает его полностью. Частные и защищенные поля и методы во многих других языках обеспечивают инкапсуляцию классов. Они ограничивают доступ к конкретным членам класса извне этого класса (частные) или извне дерева наследования (защищенные). В Python декорирование имен не ограничивает доступ к атрибутам, а только делает его менее удобным.

На самом деле декорирование имен — это неявный способ предотвращать конфликты имен. Например, может оказаться, что тот или иной идентификатор

идеально подходит для нового внутреннего атрибута некоторого подкласса. Если такое имя уже используется где-то в дереве наследования, конфликт имен может привести к неожиданным эффектам.

В таких ситуациях программист может выбрать другое имя или декорировать имена, чтобы избежать коллизий. Декорирование имен также помогает избегать конфликтов имен в подклассах. Однако его не рекомендуется использовать в базовых классах по умолчанию просто ради того, чтобы предотвратить конфликты заранее.

В конечном итоге все сводится к тому, как писать код «по-питонически». В языках со статической типизацией ключевые слова `private` и `protected` ограничивают доступ к атрибутам так, что обычно обратиться к частным/защищенным атрибутам за пределами класса невозможно. В Python применяется другой подход: четко обозначать предполагаемое использование каждого атрибута, вместо того чтобы запрещать пользователям делать то, что они хотят. С декорированием имен или без него, программисты все равно найдут возможность обратиться к атрибуту. Так стоит ли создавать лишние препятствия?

Если атрибут не является открытым (public), по действующим соглашениям он снабжается префиксом _. Префикс не запускает никакого алгоритма декорирования имен, а только указывает, что атрибут является внутренним членом класса и не предназначен для использования вне контекста класса. Многие IDE и системы проверки стиля поддерживают это соглашение и могут автоматически отмечать обращения к таким внутренним элементам за пределами их классов.

В Python есть и другие способы отделять открытую часть класса от внутреннего кода: дескрипторы и свойства.

Дескрипторы

Дескрипторы позволяют задать особые действия, которые должны выполняться при обращении к атрибутам объекта. На дескрипторах основан доступ к сложным атрибутам Python. Они используются во внутренней реализации свойств, методов, методов классов, статических методов и класса `super`. Дескрипторы — это объекты, которые определяют, как должен быть устроен доступ к атрибутам другого класса. Иначе говоря, один класс может делегировать другому управление атрибутами.

В основе классов дескрипторов лежат три специальных метода, образующих протокол дескриптора:

- `__set__(self, obj, value)`: вызывается, когда задается значение атрибута. В дальнейших примерах этот метод будет называться методом записи, или сеттером (setter).

- `__get__(self, obj, owner=None)`: вызывается при чтении атрибута; далее будет называться методом чтения, или геттером (getter).
- `__delete__(self, obj)`: вызывается, когда над атрибутом выполняется `del`.

Дескриптор, реализующий `__get__()` и `__set__()`, называется **дескриптором данных**. Если дескриптор реализует только `__get__()`, то он называется **дескриптором без данных**.

Методы из протокола дескриптора вызываются специальным методом `__getattribute__()` объекта при каждом обращении к атрибуту (не путайте с методом `__getattr__()`, предназначенным для другой цели). Каждый раз, когда в программе происходит такое обращение — либо с помощью точечной записи в форме `'экземпляр.атрибут`, либо при вызове функции `getattr(экземпляр, 'атрибут')`, — неявно вызывается метод `__getattribute__()`, который ищет атрибут по такой схеме:

1. Проверить, является ли атрибут дескриптором данных в объекте класса экземпляра.
2. Если нет, искать атрибут в атрибуте `__dict__` экземпляра.
3. Наконец, проверить, является ли атрибут дескриптором без данных в объекте класса экземпляра.

Другими словами, у дескрипторов данных приоритет выше, чем у поиска в `__dict__`, а у поиска в `__dict__` — выше, чем у дескрипторов без данных.

Для лучшего понимания приведем измененный пример из официальной документации Python, который демонстрирует, как дескрипторы работают в реальном коде:

```
class RevealAccess(object):
    """Дескриптор данных, который записывает и читает значения,
       выводя сообщение о доступе к ним.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Получаем', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Обновляем', self.name)
        self.val = val

    def __delete__(self, obj):
```

```
        print('Удаляем', self.name)

class MyClass(object):
    x = RevealAccess(10, 'var "x"')
    y = 5
```

Официальное руководство по использованию дескрипторов, содержащее многочисленные примеры, доступно по адресу https://docs.python.org/3.9/howto/descriptor.html.

Обратите внимание: `x = RevealAccess()` определяется как атрибут класса, а не присваивается в методе `__init__()`. Чтобы дескрипторы работали, их нужно определять как атрибуты класса. Кроме того, они ведут себя скорее как методы, чем как обычные атрибуты-переменные. Вот пример использования дескриптора `RevealAccess` в интерактивном сеансе:

```
>>> m = MyClass()
>>> m.x
Получаем var "x"
10
>>> m.x = 20
Обновляем var "x"
>>> m.x
Получаем var "x"
20
>>> m.y
5
>>> del m.x
Удаляем var "x"
```

Этот пример показывает, что если у класса есть дескриптор данных для заданного атрибута, то при каждом чтении атрибута экземпляра вызывается метод `__get__()` дескриптора, а при каждом присваивании такому атрибуту вызывается `__set__()`. Метод `__del__` дескриптора вызывается, когда атрибут экземпляра удаляется инструкцией `del экземпляр.атрибут` или вызовом `delattr(экземпляр, 'атрибут')`.

Различия между дескрипторами данных и дескрипторами без данных важны по причинам, описанным в начале раздела. Python уже использует протокол дескрипторов, чтобы связывать функции класса с экземплярами как методы.

На дескрипторах также основан механизм работы декораторов `classmethod` и `staticmethod`. Это объясняется тем, что на самом деле объекты функций — это тоже дескрипторы без данных:

```
>>> def function(): pass
>>> hasattr(function, '__get__')
True
>>> hasattr(function, '__set__')
False
```

Это относится и к функциям, созданным с помощью лямбда-выражений:

```
>>> hasattr(lambda: None, '__get__')
True
>>> hasattr(lambda: None, '__set__')
False
```

Таким образом, если бы `__dict__` не имел более высокий приоритет, чем дескринторы без данных, нельзя было бы динамически переопределять конкретные методы уже сконструированных экземпляров во время выполнения. К счастью, дескрипторы в Python обеспечивают такую возможность, поэтому разработчики могут применять популярный прием **горячей подмены**, чтобы править внутреннее устройство экземпляров «на ходу» без наследования.

Горячая подмена, или манкипатчинг (monkey patching), — динамическая модификация экземпляра класса во время выполнения, в процессе которой добавляются, изменяются или удаляются атрибуты, но не затрагивается определение класса или исходный код.

Пример из практики: отложенное вычисление атрибутов

Рассмотрим пример практического использования дескрипторов, когда инициализация атрибута класса откладывается до момента обращения к нему из экземпляра. Это может быть полезно, если инициализация таких атрибутов зависит от некоторого контекста, который еще недоступен на момент импортирования класса. Также этот подход помогает сэкономить ресурсы, если инициализация очень ресурсоемка, но в момент импортирования класса еще неизвестно, будет ли атрибут вообще использоваться. Такой дескриптор можно реализовать так:

```
class InitOnAccess:
    def __init__(self, init_func, *args, **kwargs):
        self.klass = init_func
        self.args = args
        self.kwargs = kwargs
        self._initialized = None
    def __get__(self, instance, owner):
        if self._initialized is None:
```

```
            print('инициализировано!')
            self._initialized = self.klass(*self.args,
             **self.kwargs)
        else:
            print('кэшировано!')
        return self._initialized
```

Класс дескриптора `InitOnAccess` включает вызовы `print()`, которые позволяют увидеть, инициализируются ли значения в момент обращения или берутся из кэша.

Представьте, что у вас есть класс, всем экземплярам которого доступен общий список отсортированных случайных значений. Список может иметь произвольную длину, поэтому целесообразно повторно использовать его для всех экземпляров. С другой стороны, сортировка очень длинного входного набора может занимать много времени. Поэтому класс `InitOnAccess` обеспечивает инициализацию списка только при первом обращении. Определение класса может выглядеть так:

```
import random

class WithSortedRandoms:
    lazily_initialized = InitOnAccess(
        sorted,
        [random.random() for _ in range(5)]
    )
```

Обратите внимание, что здесь мы передаем функции `range()` относительно небольшой входной набор, чтобы вывод легче читался. Вот пример использования класса `WithSortedRandoms` в интерактивном сеансе:

```
>>> m = WithSortedRandoms()
>>> m.lazily_initialized
инициализировано!
[0.2592159616928279, 0.32590583255950756, 0.4015520901807743,
0.4148447834912816, 0.4187058605495758, 0.4534290894962043,
0.4796775578337028, 0.6963642650184283, 0.8449725511007807,
0.8808174325885045]
>>> m.lazily_initialized
кэшировано!
[0.2592159616928279, 0.32590583255950756, 0.4015520901807743,
0.4148447834912816, 0.4187058605495758, 0.4534290894962043,
0.4796775578337028, 0.6963642650184283, 0.8449725511007807,
0.8808174325885045]
```

В официальной библиотеке OpenGL для Python, которая доступна в PyPI под именем PyOpenGL, с номощью похожего приема реализуется объект `lazy_property`, который является одновремепно декоратором и дескриптором данных:

```
class lazy_property(object):
    def __init__(self, function):
        self.fget = function

    def __get__(self, obj, cls):
        value = self.fget(obj)
        setattr(obj, self.fget.__name__, value)
        return value
```

Функция `setattr()` задает атрибут экземпляра объекта в соответствии с атрибутом из предоставленного позиционного аргумента, в данном случае `self.fget.__name__`. Он устроен таким образом, потому что дескриптор `lazy_property` должен использоваться как декоратор метода, который предоставляет инициализированное значение, как в следующем примере:

```
class lazy_property(object):
    def __init__(self, function):
        self.fget = function

    def __get__(self, obj, cls):
        value = self.fget(obj)
        setattr(obj, self.fget.__name__, value)
        return value

class WithSortedRandoms:
    @lazy_property
    def lazily_initialized(self):
        return sorted([[random.random() for _ in range(5)]])
```

Такая реализация сходна с использованием декоратора `property`, речь о котором пойдет в следующем разделе. Унаковапная в ней функция выполняется только один раз, после чего атрибут экземпляра заменяется значением, которое возвращает это свойство-функция. Приоритет атрибута экземпляра выше, чем дескриптора (атрибута класса), ноэтому в каждом конкретном экземпляре класса не будут выполняться повторные инициализации. Этот прием часто оказывается полезпым, если нужно выполнить два требования одновременно:

- Экземпляр объекта должеп храниться как атрибут класса, общий для всех его экземпляров (для экономии ресурсов).
- Объект нельзя инициализировать в момент импортировапия, потому что процесс его создания зависит от некоторого глобального состояния/контекста приложения.

Такие ситуации часто встречаются в приложениях, написанных с использованием OpenGL. Например, создание шейдеров в OpenGL — затратная операция, потому что она требует компиляции кода, написанного на языке GLSL (OpenGL Shading Language). Имеет смысл создать шейдеры один раз и в то же время раз-

местить их определение вблизи от классов, которым они требуются. С другой стороны, невозможно компилировать шейдеры, не инициализировав контекст OpenGL, поэтому их трудно определить и надежно скомпилировать в глобальном пространстве имен модуля в момент импортирования.

В следующем примере показано, как можно использовать измененную версию декоратора `lazy_property` для PyOpenGL (здесь декоратор называется `lazy_class_attribute`) в воображаемом приложении на базе OpenGL. В исходный декоратор `lazy_property` понадобилось внести выделенное изменение, чтобы разные экземпляры класса могли совместно использовать атрибут:

```
import OpenGL.GL as gl
from OpenGL.GL import shaders


class lazy_class_attribute(object):
    def __init__(self, function):
        self.fget = function

    def __get__(self, obj, cls):
        value = self.fget(cls)
        # внимание: хранится в объекте класса, а не в экземпляре,
        #           независимо от того, осуществляется доступ
        #           на уровне класса или экземпляра
        setattr(cls, self.fget.__name__, value)
        return value

class ObjectUsingShaderProgram(object):
    # Тривиальная сквозная реализация вершинных шейдеров
    VERTEX_CODE = """
        # Ядро версии 330
        layout(location = 0) in vec4 vertexPosition;
        void main(){
            gl_Position =  vertexPosition;
        }
    """
    # Тривиальный фрагментный шейдер, который закрашивает
    # все белым цветом
    FRAGMENT_CODE = """
        #version 330 core
        out lowp vec4 out_color;
        void main(){
            out_color = vec4(1, 1, 1, 1);
        }
    """

    @lazy_class_attribute
    def shader_program(self):
        print("компилируем!")
        return shaders.compileProgram(
```

```
            shaders.compileShader(
                self.VERTEX_CODE, gl.GL_VERTEX_SHADER
            ),
            shaders.compileShader(
                self.FRAGMENT_CODE, gl.GL_FRAGMENT_SHADER
            )
        )
```

Как и все продвинутые средства синтаксиса Python, дескрипторы стоит использовать с осторожностью и хорошо документировать в коде. Дескрипторы влияют на фундаментальные аспекты поведения класса. Для неопытных разработчиков изменепное поведение класса может оказаться неожиданным и непонятным. Поэтому если дескрипторы играют важную роль в кодовой базе вашего проекта, стоит позаботиться о том, чтобы все участники вашей команды хорошо понимали, как устроены дескрипторы и как они работают.

Свойства

Вероятно, термин «инкапсуляция» знаком каждому, кто программировал на C++ или Java. Этот механизм предотвращает прямые обращения к полям класса; он основан на предположении, что все внутренние данные, хранящиеся в классе, должны считаться частными. В нолностью инкапсулированном классе должно быть как можно меньше методов, открытых для внешнего доступа. Обращаться к состоянию объекта нужно через сеттеры и геттеры, которые обеспечивают корректный доступ. Например, в Java этот паттерн может выглядеть так:

```
public class UserAccount {
  private String username;

  public String getUsername() {
    return username;
  }

  public void setUsername(String newUsername) {
    this.username = newUsername;
  }
}
```

Метод `getUsername()` является методом чтения (геттером) для `username`, а `setUsername()` — методом записи (сеттером) для `username`. Этот механизм опирается на вполне здравую идею: если скрыть доступ к элементам класса за геттерами и сеттерами, можно обеспечить корректное обращение к внутренним значениям класса (допустим, проверяя данные перед записью). При этом в открытом API класса создается точка расширения на тот случай, если понадобится добавить новое поведение, не нарушив обратную совместимость API класса.

Рассмотрим класс, который представляет учетную запись пользователя и среди прочего хранит пароль. Если вы хотите, чтобы в журнале сохранялась запись аудита при каждом обращении к паролю, можно либо помещать вызовы аудита в каждую точку кода, где происходят эти обращения, либо обращаться к паролю через методы чтения и записи, в которых журналирование выполняется по умолчанию.

Проблема в том, что нельзя точно предсказать, какие участки программы потребуют расширения в будущем. Из-за этого простого факта часто происходит чрезмерная инкапсуляция и громоздятся бесконечные перечни геттеров и сеттеров для всех возможных полей, которые могли бы быть открытыми. Писать эти методы утомительно, притом часто они не дают никаких преимуществ, а только захламляют код.

К счастью, в Python принят совершенно иной подход к геттерам и сеттерам, основанный на механизме свойств. Они обеспечивают свободный доступ к открытым элементам классов, которые просто преобразовываются в методы чтения и записи там, где возникает такая необходимость. При этом обратная совместимость API класса не нарушается.

Рассмотрим пример инкапсулированного класса `UserAccount`, в котором не используются свойства:

```
class UserAccount:
    def __init__(self, username, password):
        self._username = username
        self._password = password

    def get_username(self):
        return self._username

    def set_username(self, username):
        self._username = username

    def get_password(self):
        return self._password

    def set_username(self, password):
        self._password = password
```

Увидев такой код с многочисленными методами `get_` и `set_`, можно почти со стопроцентной уверенностью сказать, что это идиома стороннего языка. Скорее всего, это написал программист на C++ или Java. Опытный разработчик на Python скорее напишет нечто такое:

```
class UserAccount:
    def __init__(self, username, password):
        self.username = username
        self.password = password
```

И только когда возникнет необходимость скрыть конкретное поле за свойством, и не ранее, опытный программист внесет следующее изменение:

```
class UserAccount:
    def __init__(self, username, password):
        self.username = username
        self._password = password


    @property
    def password(self):
        return self._password

    @password.setter
    def password(self, value):
        self._password = value
```

Свойства предоставляют встроенный тип дескриптора, который умеет связывать атрибут с набором методов. Функция `property()` получает четыре необязательных аргумента: `fget`, `fset`, `fdel` и `doc`. Последний аргумент можно передавать, чтобы определить функцию `docstring`, которая связывается с атрибутом так, как если бы он был методом. Вот пример класса `Rectangle`, которым можно манипулировать как с помощью прямого доступа к атрибутам, представляющим две угловые точки, так и через свойства `width` и `height`:

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2

    def _width_get(self):
        return self.x2 - self.x1

    def _width_set(self, value):
        self.x2 = self.x1 + value

    def _height_get(self):
        return self.y2 - self.y1

    def _height_set(self, value):
        self.y2 = self.y1 + value

    width = property(
        _width_get, _width_set,
        doc="ширина прямоугольника, измеряемая слева"
    )
    height = property(
        _height_get, _height_set,
```

```
        doc="высота прямоугольника, измеряемая сверху"
    )

    def __repr__(self):
        return "{}({}, {}, {}, {})".format(
            self.__class__.__name__,
            self.x1, self.y1, self.x2, self.y2
        )
```

Свойства, определенные таким образом, можно продемонстрировать в интерактивном сеансе:

```
>>> rectangle = Rectangle(10, 10, 25, 34)
>>> rectangle.width, rectangle.height
(15, 24)
>>> rectangle.width = 100
>>> rectangle
Rectangle(10, 10, 110, 34)
>>> rectangle.height = 100
>>> rectangle
Rectangle(10, 10, 110, 110)
>>> help(Rectangle)
Help on class Rectangle

class Rectangle(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, x1, y1, x2, y2)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------
 |  Data descriptors defined here:
 |  (...)
 |
 |  height
 |      высота прямоугольника, измеряемая сверху
 |
 |  width
 |      ширина прямоугольника, измеряемая слева
```

Со свойствами легче писать дескрипторы, но если вы используете наследование классов, нужно действовать осторожно. Атрибут создается «на ходу» с помощью методов текущего класса и не использует методы, переопределенные в производных классах.

Так, в следующем нримере не удастся переопределить реализацию метода `fget` для свойства `width` родительского класса:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} метров".format(self.x2 - self.x1)
...
>>> Rectangle(0, 0, 100, 100).width
100
```

Чтобы справиться с этим затруднением, достаточно переписать все свойство в производном классе:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} метров".format(self.x2 - self.x1)
...     width = property(_width_get, Rectangle.width.fset)
...
>>> MetricRectangle(0, 0, 100, 100).width
'100 метров'
```

К сожалению, этот код грозит проблемами с сопровождением. Он может вызвать путаницу, если разработчик решит изменить родительский класс, но забудет обновить вызов `property`. Поэтому не рекомендуется обновлять только отдельные части поведения свойства. Если вам нужно, чтобы методы свойств в производном классе работали не так, как в родительском, лучше переписать все методы свойств в производном классе. В большинстве случаев по-другому просто не получится, потому что обычно когда у свойства меняется что-то в атрибуте `setter`, это сопровождается изменениями и в `getter`.

Из-за этого наилучший синтаксис создания свойств — такой, где `property` используется как декоратор. При этом сокращается количество сигнатур методов внутри класса, а код лучше читается и создает меньше проблем с сопровождением:

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2

    @property
    def width(self):
        """ширина прямоугольника, измеряемая слева"""
        return self.x2 - self.x1

    @width.setter
    def width(self, value):
        self.x2 = self.x1 + value

    @property
```

```
    def height(self):
        """высота прямоугольника, измеряемая сверху"""
        return self.y2 - self.y1

    @height.setter
    def height(self, value):
        self.y2 = self.y1 + value
```

Самое замечательное в механизме свойств Python — то, что их можно постепенно вводить в классы. Можно начать с того, чтобы не прятать открытые атрибуты экземпляра класса и преобразовывать их в свойства только тогда, когда возникнет такая необходимость. Другие части вашего кода не заметят никаких изменений в API класса, потому что доступ к свойствам ничем не будет отличаться от доступа к обычным атрибутам экземпляров.

К этому моменту мы рассмотрели объектно-ориентированную модель данных Python в сравнении с другими языками программирования. Однако модель данных — всего лишь часть общей картины ООП. Другим важным аспектом каждого объектно-ориентированного языка является его подход к полиморфизму. Python предлагает несколько реализаций полиморфизма, о которых пойдет речь в следующем разделе.

Динамический полиморфизм

Полиморфизм присутствует во многих объектно-ориентированных языках. Это механизм, который абстрагирует интерфейс объекта от его типа. В разных языках программирования полиморфизм устроен по-разному. В языках со статической типизацией он обычно реализуется следующими средствами:

- **Подтипы**: подтипы типа `A` можно использовать в любом интерфейсе, поддерживающем тип `A`. Интерфейсы определяются явно, а подтипы/подклассы наследуют интерфейсы своих родителей. Эта реализация полиморфизма используется в C++.
- **Неявные интерфейсы**: в интерфейсе, поддерживающем тип `A`, можно использовать любой тип, если он реализует те же методы (то есть имеет такой же интерфейс), что и тип `A`. Интерфейсы все еще объявляются явно, но подклассы/подтипы не обязаны явно наследовать базовым классам/типам, которые определяют интерфейс. Эта реализация полиморфизма используется в Go.

Python — язык с динамической типизацией, поэтому в нем применяется менее строгий механизм полиморфизма, который часто называют **«утиной типизацией»** (duck typing). Ее принцип формулируется так:

> *Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка.*

В применении к Python этот принцип означает, что в заданном контексте можно использовать любой объект, который ведет себя так, как ожидает этот контекст. Такое понимание типизации очень близко к неявным интерфейсам языка Go, хотя при этом для аргументов функций не приходится объявлять ожидаемые интерфейсы. Поскольку Python не требует строгого соответствия типов или интерфейсов аргументов функций, неважно, какие типы объектов передаются функции. Важно лишь то, какие методы этих объектов фактически используются в теле функции.

Чтобы лучше понять концепцию, рассмотрим пример функции, которая читает файл, выводит его содержимое и закрывает файл:

```
def printfile(file):
    try:
        contents = file.read()
        print(file)
    finally:
        file.close()
```

По сигнатуре функции `printfile()` уже можно догадаться, что она ожидает получить файл или объект, подобный файлу (скажем, `StringIO` из модуля `io`). Но в действительности эта функция примет любой объект и не выдаст исключения, если предоставить ей аргумент, который удовлетворяет следующим условиям:

- У аргумента `file` есть метод `read()`.
- Результат `file.read()` является донустимым аргументом для функции `print()`.
- У аргумента `file` есть метод `close()`.

Эти три условия соответствуют трем участкам кода, на которых в этом примере проявляется нолиморфизм. В зависимости от реального тина аргумента `file` функция `printfile()` будет использовать разные реализации методов `read()` и `close()`. Тип переменной `contents` тоже может различаться в зависимости от реализации `file.read()`, и функция `print()` будет использовать соответствующую реализацию строкового представления объекта.

Это чрезвычайно мощный и гибкий подход к полиморфизму и типизации, хотя у него есть свои недостатки. Поскольку нет контроля тинов и интерфейсов, становится сложнее проверить нравильность кода перед выполнением. Поэтому качественным приложениям приходится полагаться на то, что код будет основательно протестирован с тщательным покрытием всех нутей, по которым может пойти его выполнение. Python позволяет частичю преодолеть эту про-

блему за счет аннотаций типов, которые могут проверяться сторонними средствами перед выполнением.

Динамическая система типов Python в сочетании с утиной типизацией создает неявную разновидность динамического полиморфизма, которая пронизывает все аспекты языка. В этом отношении у Python много общего с JavaScript, где тоже нет статического контроля типов. Однако разработчикам Python доступны и другие формы полиморфизма, более «классические» и явно выраженные по своей природе. Одна из таких форм — перегрузка операторов.

Перегрузка операторов

Перегрузка операторов — особая разновидность полиморфизма, которая позволяет языку использовать разные реализации одного и того же оператора в зависимости от типов онерандов.

Операторы многих языков программирования уже полиморфны. Следующие выражения — действительные конструкции в Python:

```
7 * 6
3.14 * 2
["a", "b"] * 3
"abba" * 2
```

В Python эти выражения будут использовать четыре разные реализации:

- `7 * 6` — целочисленное умножение, результат — целое число `42`.
- `3.14 * 2` — вещественное умножепие, результат — число с плавающей точкой `6.28`.
- `["a", "b"] * 3` — умножение списка, результат — список `['a', 'b', 'a', 'b', 'a', 'b']`.
- `"abba" * 2` — умножение строки, результат — строка `'abbaabba'`.

Все операторы Python используют разную семантику и реализацию в зависимости от типов операндов. В Python есть несколько встроенных типов, которым соответствуют разные реализации онераторов, но это пе означает, что любой оператор можно использовать с любым типом.

Например, оператор `+` применяется для суммирования или конкатенации операндов. Эта операция имеет смысл как для числовых типов (целых и с плавающей точкой), так и для строк и списков. Однако этот оператор нельзя использовать со множествами или словарями, потому что такие операции не имеют математического смысла (множества могут пересекаться или объединяться), и результат получится неоднозпачным (какие значения из двух словарей использовать в случае конфликта ключей?).

Перегрузка операторов — это всего лишь расширение встроенного полиморфизма операторов, который уже включен в язык программирования. Многие языки, включая Python, позволяют определить новую реализацию для типов операндов, у которых нет действительной реализации операторов, или скрыть существующую реализацию с помощью подклассов.

Двойные подчеркивания (языковые протоколы)

В модели данных Python есть множество методов со специальными именами, которые можно переопределять в ваших классах, чтобы наделять их дополнительными синтаксическими возможностями. Эти методы можно узнать по особым именам, которые начинаются и заканчиваются двойным подчеркиванием. Их называют также **dunder-методами** (dunder — сокращение от double underscores, то есть двойное подчеркивание).

Самый очевидный и распространенный пример такого dunder-метода — `__init__()`, который используется для инициализации экземпляров классов:

```
class CustomUserClass:
    def __init__(self, initiatization_argument):
        ...
```

Эти методы — сами по себе или определенные в особых комбинациях — составляют так называемые языковые протоколы. Если мы говорим, что объект реализует тот или иной языковой протокол, это означает, что он совместим с соответствующей частью синтаксиса Python. Ниже приведен список самых распространенных протоколов языка Python.

Имя нротокола	Методы	Описание
Протокол вызываемого объекта	`__call__()`	Позволяет вызывать объекты с круглыми скобками: `instance()`
Протоколы дескрипторов	`__set__()`, `__get__()` и `__del__()`	Позволяют управлять доступом к атрибутам классов (см. раздел «Дескрипторы»)
Протокол контейнера	`__contains__()`	Позволяет проверить, содержит ли объект некоторое значение, с помощью ключевого слова `in`: `value in instance`

Протокол итерируемого объекта	`__iter__()`	Позволяет перебирать объекты с помощью ключевого слова `for`:
		`for value in instance:`
		`...`

(*окончание*)

Имя протокола	Методы	Описание

Протокол последовательности	`__getitem__()`, `__len__()`	Позволяет индексировать объекты с помощью квадратных скобок и запрашивать длину объектов встроенной функцией:
		`item = instance[index]`
		`length = len(instance)`

У каждого оператора Python есть собственный протокол, и для перегрузки операторов требуется реализовывать dunder-методы этого протокола. Python предоставляет более 50 перегружаемых операторов, которые можно разделить на пять основных категорий:

- Арифметические операторы.
- Операторы присваивания на месте.
- Операторы сравнения.
- Операторы проверки идентичности.
- Поразрядные операторы.

Протоколов достаточно много, и мы не будем рассматривать их все. Вместо этого разберем практический пример, который поможет лучше попять, как самостоятельно реализовать перегрузку оператора.

Полный список доступных dunder-методов можно найти в разделе «Data model» официальной документации Python по адресу https://docs.python.org/3/reference/datamodel.html.

Все операторы также представлены в виде обычных функций в модуле operators. В его документации есть хороший обзор операторов Python. Ее можно найти по адресу https://docs.python.org/3.9/library/operator.html.

Допустим, мы имеем дело с математической задачей, которую можно решить посредством матричных уравнений. Матрица — это математический объект линейной алгебры с четко онределенными операциями. В простейшей форме она выглядит как двумерный массив чисел. В Python нет встроенной ноддержки многомерных массивов, если не считать списков, вложенных в списки. Поэтому было бы неплохо иметь специальный класс, который инкапсулирует матрицы и операции над ними. Начнем с инициализации этого класса:

```
class Matrix:
    def __init__(self, rows):
        if len(set(len(row) for row in rows)) > 1:
            raise ValueError("Все строки матрицы должны быть одинаковой длины")

        self.rows = rows
```

Первый dunder-метод класса `Matrix` — метод `__init__()` — позволяет безопасно инициализировать матрицу. На входе он получает переменную со списком строк матрицы. Так как в каждой строке должно быть одно и то же количество элементов, мы перебираем строки и убеждаемся, что их длина одинакова.

Теперь добавим перегрузку первого оператора:

```
def __add__(self, other):
    if (
        len(self.rows) != len(other.rows) or
        len(self.rows[0]) != len(other.rows[0])
    ):
        raise ValueError("Размеры матриц не совпадают")

    return Matrix([
        [a + b for a, b in zip(a_row, b_row)]
        for a_row, b_row in zip(self.rows, other.rows)
    ])
```

Метод `__add__()` отвечает за перегрузку оператора `+` (плюс) и здесь используется для сложения двух матриц. Складывать можно только две матрицы с одинаковыми размерами. Это довольно простая операция: каждый элемент новой матрицы вычисляется сложением соответствующих элементов матриц-операндов.

Метод `__sub__()` отвечает за перегрузку оператора `-` (минус), который будет использоваться для вычитания матриц. Результат вычисляется подобно оператору `+`:

```
def __sub__(self, other):
    if (
        len(self.rows) != len(other.rows) or
        len(self.rows[0]) != len(other.rows[0])
    ):
        raise ValueError("Размеры матриц не совпадают")

    return Matrix([
        [a - b for a, b in zip(a_row, b_row)]
        for a_row, b_row in zip(self.rows, other.rows)
    ])
```

Остается добавить в класс последний метод:

```
def __mul__(self, other):
    if not isinstance(other, Matrix):
        raise TypeError(
            f"Не определено умножение типов {type(other)} и Matrix"
        )

    if len(self.rows[0]) != len(other.rows):
        raise ValueError(
            "Размеры матриц не совпадают"
        )

    rows = [[0 for _ in other.rows[0]] for _ in self.rows]

    for i in range(len(self.rows)):
        for j in range(len(other.rows[0])):
            for k in range(len(other.rows)):
                rows[i][j] += self.rows[i][k] * other.rows[k][j]

    return Matrix(rows)
```

Последний перегруженный оператор — самый сложный. Это оператор *, который реализуется методом `__mul__()`. В линейной алгебре матрицы перемножаются не так, как вещественные числа. Умножить одну матрицу на другую можно только в том случае, если количество столбцов первой матрицы равно количеству строк второй матрицы. Результатом операции является новая матрица, каждый элемент которой равен скалярному произведению соответствующей строки первой матрицы и соответствующего столбца второй матрицы.

Здесь мы построили собственную реализацию класса матриц, чтобы продемонстрировать перегрузку операторов. На самом деле хотя в Python нет встроенного тина для матриц, конструировать его с нуля необязательно. Матричная алгебра (вместе с другими возможностями) поддерживается в NumPy — одном из лучших математических пакетов Python. NumPy можно легко загрузить из PyPI.

Сравнение с C++

Среди языков программирования, для которых особенно характерна перегрузка операторов, выделяется C++. Это объектно-ориентированный язык со статической типизацией, совсем непохожий на Python. В Python поддерживаются элементы ООП и некоторые механизмы, которые по сути аналогичны мехапизмам C++, — прежде всего это концепция классов и их наследование, а также перегрузка операторов. Тем не менее эти механизмы реализованы в языке совершенно иначе. Именно поэтому так интересно сравнивать эти два языка.

В C++, в отличие от Python, параллельно сосуществуют несколько механизмов полиморфизма. Основной механизм, основанный на подтипах, также доступен в Python. Вторая главная разновидность полиморфизма в C++ — **ситуативный (ad hoc) полиморфизм**, основанный на **перегрузке функций**. В Python нет его прямого аналога.

Перегрузка функций в C++ позволяет создать несколько реализаций одной и той же функции в зависимости от входных аргументов. Это означает, что в программе могут быть две функции или метода с одинаковыми именами, но разным количеством и/или разными типами аргументов. Так как C++ — язык со статической типизацией, тины аргументов всегда известны заранее, а подходящая реализация выбирается во время компиляции.

Чтобы достичь еще большей гибкости, перегрузку функций можно использовать в сочетании с перегрузкой операторов. Чтобы лучше понять, в каких ситуациях может пригодиться такая комбинированная перегрузка, вернемся к сценарию с умножением матриц. Мы знаем, что две матрицы можно перемножить, и в предыдущем разделе было показано, как это делается. Однако в линейной алгебре матрицу также можно умножить на скалярный тип, например, вещественное число. Результатом операции будет новая матрица, в которой каждый элемент исходной матрицы умножен на скалярное значение. В коде этому будет соответствовать другая реализация оператора умножения.

В C++ можно просто создать несколько сосуществующих функций, перегружающих оператор *. Ниже приведен пример сигнатур функций C++ для перегруженных операторов, предоставляющих разные реализации матричного и скалярного умножения:

```
Matrix operator*(const Matrix& lhs, const Matrix& rhs)
Matrix operator*(const Matrix& lhs, const int& rhs)
Matrix operator*(const Matrix& lhs, const float& rhs)
Matrix operator*(const int& lhs, const Matrix& rhs)
Matrix operator*(const float& lhs, const Matrix& rhs)
```

Python — язык с динамической типизацией, и это главная причина, по которой в нем не существует перегрузки функций, как в C++. Чтобы реализовать для класса `Matrix` перегрузку оператора * с поддержкой как матричного, так и скалярного умножения, необходимо проверять тип входных данных оператора во время выполнения. Это можно сделать с помощью встроенной функции `isinstance()`, как в следующем примере:

```
def __mul__(self, other):
    if isinstance(other, Matrix):
        ...

    elif isinstance(other, Number):
        return Matrix([
```

```
            [item * other for item in row]
            for row in self.rows
        ])
    else:
        raise TypeError(f"Нельзя умножить {type(other)} на Matrix")
```

Другое принципиальное отличие заключается в том, что перегрузка операторов C++ осуществляется свободными функциями, а не методами класса, тогда как в Python оператор всегда разрешается соответствующим специальным методом операнда. Это отличие можно снова продемонстрировать на примере скалярной реализации. Код из предыдущего примера позволяет умножать матрицу на целое число в следующей форме:

```
Matrix([[1, 1], [2, 2]]) * 3
```

Такой способ работает, потому что реализация перегруженного оператора будет обрабатываться, начиная с левого операнда. Однако следующее выражение вызовет ошибку `TypeError`:

```
3 * Matrix([1, 1], [2, 2]])
```

В C++ можно обеспечить несколько версий перегрузки оператора, покрывающих все комбинации типов операндов для оператора `*`. В Python аналогичного результата можно добиться с помощью метода `__rmul__()`. Этот метод обрабатывает операнды справа налево, если левосторонний оператор `__mul__()` выдает ошибку `TypeError`. У большинства инфиксных операторов есть альтернативные правосторонние реализации. Ниже приведен пример метода `__rmul__()` для класса `Matrix`, который позволяет выполнять скалярное умножение с числовым аргументом в правой части:

```
def __rmul__(self, other):
    if isinstance(other, Number):
        return self * other
```

Как видите, в этой реализации все еще требуется проверять тип функцией `isinstance()`, так что перегрузку операторов следует применять очень осторожно, особенно если перегруженные операторы получают совершенно новый смысл, не соответствующий их исходному назначению.

Необходимость предоставлять альтернативные перегруженные реализации оператора в зависимости от типа одного операнда обычно говорит о том, что оператор утратил свой четкий смысл. Например, умножение матриц и скалярное умножение с математической точки зрения являются двумя разными операциями и обладают разными свойствами. Например, скалярное умножение кумулятивно, а умножение матриц — нет. Перегрузка оператора для нестандартного класса, имеющего несколько внутренних реализаций, быстро приводит к путанице, особенно в коде, относящемся к математическим задачам.

Мы намеренно умалчиваем о том, что в Python на самом деле есть специальный оператор умножения матриц, хотя и нет встроенного матричного типа. Это еще более наглядно демонстрирует опасности и сложности, которыми чревато злоупотребление перегрузкой операторов. Оператор умножения матриц обозначается `@` и был введен во многом именно из-за потенциальной путаницы между скалярным и матричным умножением.

Во многих языках программирования перегрузку операторов можно рассматривать как частный случай перегрузки функций и методов. Как ни странно, в Python поддерживается перегрузка операторов, но нет настоящей перегрузки функций и методов. Чтобы восполнить этот пробел, используются другие паттерны, о которых мы поговорим в следующем разделе.

Перегрузка функций и методов

Перегрузка функций и методов — стандартная возможность многих языков программирования. Это еще одно проявление механизма полиморфизма. Перегрузка позволяет иметь несколько реализаций одной и той же функции с разными сигнатурами вызова. Компилятор или интерпретатор может выбрать подходящую реализацию в зависимости от набора аргументов, переданных при вызове. Этот выбор обычно зависит от следующих характеристик:

- **Арность функции** (количество параметров). Два определения функций могут использовать одно и то же имя, если в их сигнатурах указано разное количество параметров.
- **Типы параметров.** Два определения функций могут использовать одно и то же имя, если в их сигнатурах указаны разные типы параметров.

Как уже упоминалось в разделе «Перегрузка операторов», в Python нет механизма перегрузки функций и методов, помимо перегрузки операторов. Если в одном модуле определить несколько функций с одинаковыми именами, то последнее определение перекроет все предшествующие.

Если нужно определить несколько реализаций функции, которые ведут себя по-разному в зависимости от типа или количества аргументов, Python предоставляет такие альтернативы:

- Методы и/или подклассы: не добиваться, чтобы функция различала типы параметров, а связать функцию с конкретным типом, определив ее как метод этого типа.
- Распаковка именованных и позиционных аргументов: Python допускает определенную гибкость, позволяя сигнатурам функций поддерживать переменное количество аргументов с помощью `*args` и `**kwargs` (такие функции также называются вариативными).

- Проверка типов: функция `isinstance()` позволяет проверить входные аргументы на соответствие конкретным типам и базовым классам, чтобы решить, как их обрабатывать.

Конечно, у каждого из этих вариантов есть свои ограничения. Переносить реализацию функции прямо в определение класса в виде метода не имеет смысла, если этот метод не обеспечивает специфического поведения объекта. Распаковка аргументов может усложнить сигнатуру функции и затруднить ее сопровождение.

Часто самой надежной и удобочитаемой альтернативой перегрузки функции в Python становится простая проверка типов. Мы уже видели этот прием в действии, когда рассматривали перегрузку операторов. Вспомните метод `__mul__()`, который умел различать матричное и скалярное умножение:

```
def __mul__(self, other):
    if isinstance(other, Matrix):
        ...

    elif isinstance(other, Number):
        ...

    else:
        raise TypeError(f"Can't subtract {type(other)} from Matrix")
```

Таким образом, то, что в языках со статической типизацией делается перегрузкой функций, в Python можно реализовать простым вызовом `isinstance()`. Это можно считать как сильной, так и слабой стороной Python. Впрочем, этот способ удобен только при небольшом количестве сигнатур вызова. Если поддерживаемых типов много, часто имеет смысл использовать более модульные паттерны. Обычно они опираются на **функции с единичной диспетчеризацией** (single-dispatch functions).

Функции с единичной диспетчеризацией

В ситуациях, когда требуется альтернатива перегрузке функций, а количество альтернативных реализаций функции велико, множественные проверки `if isinstance(...)` быстро выходят из-под контроля. Хороший тон программирования велит писать небольшие функции с единственным назначением. Большая функция, которая разветвляется по нескольким типам, чтобы обрабатывать входные аргументы, редко является признаком качественного проектирования.

В стандартной библиотеке Python есть удобная альтернатива. Декоратор `functools.singledispatch()` позволяет зарегистрировать несколько реализаций функции. Эти реализации могут принимать произвольное количество аргумен-

тов, но нужная реализация будет выбираться в зависимости от тина первого из них. Единичная диспетчеризация начинается с онределения функции, которая будет использоваться по умолчанию для всех незарегистрированных типов. Допустим, мы пишем функцию, которая может представлять разные переменные в человекочитаемом формате для большого отчета, выводящегося в консоль. По умолчанию можно использовать f-строку, чтобы выводить необработанные данные в строковом формате:

```
from functools import singledispatch

@singledispatch
def report(value):
    return f"raw: {value}"
```

После этого можпо регистрировать разные реализации для разных типов с помощью декоратора `report.register()`. Он способен читать аннотации тинов аргументов функций, чтобы регистрировать обработчики для конкретных тинов. Допустим, объекты `datetime` должны выводиться в формате ISO:

```
from datetime import datetime

@report.register
def _(value: datetime):
    return f"dt: {value.isoformat()}"
```

Обратите внимание, что для фактического имени функции мы использовали маркер _. Он служит двум целям. Во-первых, это общенринятое обозначение объектов, которые не должны явно использоваться в программе. Во-вторых, если бы вместо него использовалось имя `report`, это привело бы к замещению исходной функции и мы потеряли бы возможность обращаться к ней и регистрировать новые типы.

Определим еще пару обработчиков для других типов:

```
from numbers import Real

@report.register
def _(value: complex):
    return f"complex: {value.real}{value.imag:+}j"

@report.register
def _(value: Real):
    return f"real: {value:f}"
```

Аннотации тинов необязательны, но мы использовали их как элементы хорошего тона. Если вы пе хотите использовать аннотации тинов, укажите регистрируемый тип в качестве аргумепта метода `register()`, как в этом примере:

```
@report.register(complex)
def _(value):
    return f"complex: {value.real}{value.imag:+}j"

@report.register(real)
def _(value):
    return f"real: {value:f}"
```

Если попытаться проверить, как ведет себя наша коллекция реализаций с единичной диспетчеризацией в интерактивном сеансе, результат будет выглядеть примерно так:

```
>>> report(datetime.now())
'dt: 2020-12-12T00:22:31.690377'
>>> report(100-30j)
'complex: 100.0-30.0j'
>>> report(9001)
'real: 9001.000000'
>>> report("January")
'raw: January'
>>> for key, value in report.registry.items():
...     print(f"{key} -> {value}")
...
<class 'object'> -> <function report at 0x7fdfd6929a60>
<class 'datetime.datetime'> -> <function _ at 0x7fdfd69a5af0>
<class 'complex'> -> <function _ at 0x7fdfd6993d30>
<class 'float'> -> <function _ at 0x7fdfd6d7ab80>
<class 'int'> -> <function _ at 0x7fdfd6d7ab80>
```

Как видите, функция `report()` стала точкой входа в набор зарегистрированных функций. Каждый раз, когда функция вызывается с аргументом, она ищет тип этого аргумента в отображении, которое хранится в `report.registry`. Там всегда есть как минимум один ключ, который связывает тип `object` с реализацией функции по умолчанию.

Также существует разновидность механизма с единичной диспетчеризацией, предназначенная для методов классов. Методы всегда принимают текущий экземпляр объекта в первом аргументе. Это означает, что декоратор `functools.singledispatch()` не будет работать, потому что первый аргумент методов всегда будет одного и того же типа. Декоратор `functools.singledispatchmethod()` учитывает эту особенность и позволяет регистрировать множественные реализации методов, зависящие от типов аргументов. Он проверяет первый аргумент, отличный от `self` и `cls`:

```
from functools import singledispatchmethod

class Example:
    @singledispatchmethod
```

```
    def method(self, argument):
        pass

    @method.register
    def _(self, argument: float):
        pass
```

Помните, что хотя единичная диспетчеризация является разновидностью полиморфизма и напоминает перегрузку функций, это не совсем одно и то же. Этот механизм не позволяет выбирать ту или иную реализацию функции в зависимости от типов нескольких аргументов, и в стандартной библиотеке Python в настоящее время нет средств множественной диспетчеризации.

Классы данных

Как вы узнали в разделе «Инициализация экземпляров классов», канонический способ объявлять атрибуты экземпляров классов состоит в том, чтобы присваивать им значения в методе `__init__()`:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Допустим, мы разрабатываем программу для геометрических вычислений, а класс `Vector` должен хранить информацию о двумерных векторах. Программа будет выводить сведения о векторе на экран и выполнять стандартные математические операции: сложение, вычитание и проверку равенства. Вы уже знаете, что эту задачу можно решить с помощью специальных методов и перегрузки операторов. Реализация класса `Vector` может выглядеть так:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Суммирует векторы оператором +"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )

    def __sub__(self, other):
        """Вычитает векторы оператором -"""
        return Vector(
            self.x - other.x,
```

```
            self.y - other.y,
        )

    def __repr__(self):
        """Возвращает текстовое представление вектора"""
        return f"<Vector: x={self.x}, y={self.y}>"

    def __eq__(self, other):
        """Проверяет два вектора на равенство"""
        return self.x == other.x and self.y == other.y
```

Вот как элементы класса `Vector` ведут себя в интерактивном сеансе со стандартными операторами:

```
>>> Vector(2, 3)
<Vector: x=2, y=3>
>>> Vector(5, 3) + Vector(1, 2)
<Vector: x=6, y=5>
>>> Vector(5, 3) - Vector(1, 2)
<Vector: x=4, y=1>
>>> Vector(1, 1) == Vector(2, 2)
False
>>> Vector(2, 2) == Vector(2, 2)
True
```

Эта реализация векторов довольно проста, однако в ней много кода, которого следовало бы избегать. Класс `Vector` ориентирован на работу с данными. Бóльшая часть его поведения — создание новых экземпляров `Vector` в результате математических операций. В нем нет ни сложной инициализации, ни нестандартных паттернов доступа к атрибутам. Во многих классах, ориентированных на работу с данными, операции проверки равенства, строкового представления и инициализации атрибутов выглядят почти одинаково.

Если в вашей программе много похожих простых классов, которые ориентированы на работу с данными и не требуют сложной инициализации, вы рискуете увязнуть в кучах шаблонного кода для методов `__init__()`, `__repr__()` и `__eq__()`.

С модулем `dataclasses` класс `Vector` становится намного короче:

```
from dataclasses import dataclass

@dataclass
class Vector:
    x: int
    y: int

    def __add__(self, other):
        """Суммирует векторы оператором + """
```

```
            return Vector(
                self.x + other.x,
                self.y + other.y,
            )

        def __sub__(self, other):
            """Вычитает векторы оператором - """
            return Vector(
                self.x - other.x,
                self.y - other.y,
            )
```

Декоратор класса `dataclass` читает аннотации атрибутов класса `Vector` и автоматически создает методы `__init__()`, `__repr__()` и `__eq__()`. Проверка равенства по умолчанию предполагает, что два экземпляра равны, если их соответствующие атрибуты равны друг другу.

Но у классов данных есть и много других полезных возможностей. В частности, можно легко обеспечить их совместимость с другими протоколами Python. Допустим, вы хотите, чтобы экземнляры класса `Vector` были неизменяемыми: тогда их можно будет использовать как ключи словаря и элементы множеств. Для этого достаточно добавить в декоратор `dataclass` аргумент `frozen=True`:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class FrozenVector:
    x: int
    y: int
```

Такой класс данных `Vector` становится нолностью неизменяемым, и его атрибуты нельзя модифицировать. При этом два экземпляра `Vector` все равно можно суммировать и вычитать, как в предыдущем примере; эти операции просто создают новые объекты `Vector`.

Вы уже знаете, чем опасно нрисвоение значения по умолчанию атрибутам класса в основном теле класса, а не в функции `__init__()`. Модуль `dataclass` предоставляет альтернативный механизм — конструктор `field()`. Он позволяет присваивать атрибутам класса данных как изменяемые, так и неизменяемые значения но умолчанию, причем делать это безопасно, не создавая риска утечки состояния между экземплярами класса. Статические и неизменяемые значения по умолчанию задаются с помощью вызова `field(default=значение)`. Изменяемые значения всегда должны передаваться с указанием конструктора типа в вызове `field(default_factory=конструктор)`. Ниже приведен пример класса данных с двумя атрибутами, которым присваиваются значения по умолчанию через конструктор `field()`:

```
from dataclasses import dataclass, field

@dataclass
class DataClassWithDefaults:
    immutable: str = field(default="это статическое значение по умолчанию")
    mutable: list = field(default_factory=list)
```

После того как атрибуту класса данных присвоено значение по умолчанию, соответствующий аргумент инициализации этого поля становится необязательным. В следующем фрагменте показаны разные способы инициализации экземпляров класса `DataClassWithDefaults`:

```
>>> DataClassWithDefaults()
DataClassWithDefaults(immutable='это статическое значение по умолчанию',
mutable=[])
>>> DataClassWithDefaults("Это неизменяемое значение")
DataClassWithDefaults(immutable='Это неизменяемое значение', mutable=[])
>>> DataClassWithDefaults(None, ["это", "просто", "список"])
DataClassWithDefaults(immutable=None, mutable=['это', 'просто', 'список'])
```

Классы данных по своей природе близки к структурам языков C или Go. Их основная задача — хранить данные и облегчать рутинную инициализацию атрибутов экземпляров. Но их не стоит рассматривать как основу всех возможных нестандартных классов. Если ваш класс не предпазначен для представления данных и/или требует нестандартной либо сложной инициализации состояния, то лучше использовать классический способ ипициализации методом `__init__()`.

Python пе ограничивается ООП, и в нем также поддерживаются другие парадигмы программирования. Одна из таких парадигм — функциональное программирование, где центральное место занимает вычисление функций. Языки чисто функционального программирования обычно кардипально отличаются от типичных объектно-ориентированных языков. Однако мультипарадигменные языки стремятся взять все лучшее из разных стилей программирования. Это можно сказать и о Python. В следующем разделе рассматриваются некоторые элементы Python, обеспечивающие поддержку функционального программирования. Вскоре вы заметите, что эта парадигма в Python па самом деле строится на фундаменте ООП.

Функциональное программирование

Одна из самых замечательных особенностей программирования на Python заключается в том, что он не обязывает писать программы в каком-то единственно верном стиле. Одну и ту же задачу всегда можно решить разными способами,

и иногда лучший способ — не тот, который кажется наиболее очевидным. Бывают задачи, для которых эффективнее всего оказывается декларативное программирование. К счастью, Python со своим богатым синтаксисом и обширной стандартной библиотекой предлагает средства функционального программирования, а функциональное программирование — это одна из основных парадигм декларативного программирования.

В функциональном программировании ноток выполнения реализуется в основном путем вычисления математических функций, а не последовательности шагов, которые изменяют состояние программы. Чисто функциональные программы стремятся к тому, чтобы не изменять состояние (то есть не вызывать побочных эффектов) и не использовать изменяемые структуры данных.

Чтобы нонять общую концепцию функционального программирования, стоит ознакомиться с основными термипами этой области:

- **Побочные эффекты.** Говорят, что функция имеет побочный эффект, если она изменяет состояние за пределами своего локального окружения. Иначе говоря, если в результате вызова функции произошло какое-либо наблюдаемое изменение за пределами области видимости этой функции — это и есть побочный эффект. Примером может быть модификация глобальной переменной, изменение атрибута объекта, доступного за пределами области видимости функции, или сохранение данных в некоторой впешней службе. Побочные эффекты — это сердцевипа концепции ООП, где экземпляры классов представляют собой объекты, инкапсулирующие состояние приложения, а методы — это функции, которые связаны с этими объектами и управляют их состоянием. Процедурное программирование тоже во многом онирается на нобочные эффекты.
- **Ссылочная прозрачность.** Если функция или выражение обладают ссылочной прозрачностью, то эту сущность можно заменить результирующим значением так, что поведение нрограммы не изменится. Таким образом, отсутствие нобочных эффектов необходимо для ссылочной нрозрачности, однако не каждая функция без побочных эффектов ею обладает. Например, встроенная функция Python `pow(x, y)` обладает ссылочной прозрачностью, потому что у нее нет побочных эффектов и для каждой конкретной пары аргументов `x` и `y` ее вызов можно заменить значением x^y. С другой стороны, у конструктора `datetime.now()` типа `datetime` вроде бы нет наблюдаемых побочных эффектов, по он возвращает разные значения нри каждом вызове. Таким образом, он не обладает ссылочной прозрачностью.
- **Чнстые функцнн.** Функция называется чистой, если она не имеет побочных эффектов и всегда возвращает одно и то же значение для одного набора входных аргументов. Другими словами, это функция со ссылочной прозрач-

ностью. Всякая математическая функция по определению является чистой. Аналогичным образом, если функция оставляет след своего выполнения во внешнем мире (например, изменяет полученные объекты), она не является чистой.

- **Фуикции первого класса.** Язык содержит функции первого класса, если с ними можно обращаться так же, как с другими значениями или сущностями. Функции первого класса можно передавать в аргументах другим функциям, возвращать из функций в виде зпачений и присваивать переменным. Иначе говоря, язык с функциями первого класса — это язык, в котором функции стаповятся полноправными сущностями. Функции в Python относятся к этой категории.

Вооружившись этими концепциями, можно описать чисто функциональный язык как язык, который:

- содержит функции первого класса;
- работает только с чистыми функциями;
- избегает любых изменений состояния и побочных эффектов.

Конечно, Python не является чисто функциональным языком программирования. Трудно представить себе полезную программу па Python, которая использует только чистые функции без каких-либо побочных эффектов. С другой стороны, Python предоставляет разнообразные возможности, которые долгие годы были доступны только в чисто функциональных языках:

- лямбда-фупкции и функции первого класса;
- функции `map()`, `filter()` и `reduce()`;
- частичные объекты и функции;
- генераторы и генераторные выражения.

Эти возможпости нозволяют писать значительные объемы кода на Python в функциональном стиле, несмотря на то что язык не является чисто функциональным.

Лямбда-функции

Лямбда-функции — чрезвычайно популярная концепция программирования, которая играет особенно важпую роль в функциональном программировании. В других языках лямбда-функции ипогда называются анонимными функциями, лямбда-выражениями или функциональпыми литералами. Лямбда-функции — это безымянные функции, которые не требуется связывать с каким-либо идентификатором (переменпой).

На определенном этапе разработки Python 3 шла жаркая дискуссия о том, чтобы исключить из языка лямбда-функции вместе с функциями `map()`, `filter()` и `reduce()`. Дополнительную информацию можно найти в статье Гвидо ван Россума о том, почему эти возможности заслуживают удаления: https://www.artima.com/weblogs/viewpost.jsp?thread=98196.

Лямбда-функции в Python можно определять только с помощью выражений. Синтаксис лямбда-функций выглядит так:

```
lambda <аргументы>: <выражение>
```

Этот синтаксис проще всего продемонстрировать, сравнив определение «обычной» функции с ее анонимным аналогом. Ниже приведена простая функция, которая возвращает площадь круга с заданным радиусом:

```
import math

def circle_area(radius):
    return math.pi * radius ** 2
```

Та же фупкция, выраженная в виде лямбда-функции, будет выглядеть так:

```
lambda radius: math.pi * radius ** 2
```

Лямбда-функции анонимны, но это не значит, что к ним нельзя обратиться по идентификатору. Функции в Python — объекты первого класса, поэтому когда вы обращаетесь к функции по имени, в действительности используется переменная, содержащая ссылку на объект функции. Лямбда-функции, как и любые другие функции, являются сущностями первого класса, поэтому их можно присваивать новой переменной. После того как лямбда-функция присвоепа переменной, опа практически не отличается от других фупкций, не считая некоторых атрибутов метаданпых. Это показапо в следующих фрагментах из сеапсов интерактивного интерпретатора:

```
>>> import math
>>> def circle_area(radius):
...     return math.pi * radius ** 2
...
>>> circle_area(42)
5541.769440932395
>>> circle_area
<function circle_area at 0x10ea39048>
>>> circle_area.__class__
<class 'function'>
>>> circle_area.__name__
'circle_area'

>>> circle_area = lambda radius: math.pi * radius ** 2
```

```
>>> circle_area(42)
5541.769440932395
>>> circle_area
<function <lambda> at 0x10ea39488>
>>> circle_area.__class__
<class 'function'>
>>> circle_area.__name__
'<lambda>'
```

Лямбда-выражения полезны прежде всего для того, чтобы определять контекстные одноразовые функции, которые не будут использоваться где-то еще. Чтобы лучше понять их потенциал, представьте, что ваше приложение хранит информацию о людях. Данные, относящиеся к каждому человеку, можно представить в форме записи на основе такого класса данных:

```
from dataclasses import dataclass

@dataclass
class Person:
    age: int
    weight: int
    name: str
```

Теперь представьте, что у вас есть набор таких записей и их нужно отсортировать по разным полям. Python поддерживает функцию `sorted()`, которая сортирует произвольный список при условии, что его элементы можно сравнивать хотя бы операцией «меньше» (оператор `<`). Для класса `Person` можно определить специальный перегруженный оператор, но для этого надо знать заранее, по каким полям будут сортироваться записи.

К счастью, функция `sorted()` принимает именованный аргумент `key` (ключ). В нем можно указать функцию, преобразующую каждый элемент входных данных в значение, которое можно естественным образом отсортировать с помощью `sorted()`. Лямбда-выражения позволяют определять такие ключи там, где они нужны. Например, сортировку по возрасту можно выполнить таким вызовом:

```
sorted(people, key=lambda person: person.age)
```

Такое поведение функции `sorted()` демонстрирует распространенный паттерн, в котором код принимает исполняемый аргумент, обеспечивающий то или иное внедренное поведение. Лямбда-выражения часто подходят, чтобы определять такое поведение.

Функции map(), filter() и reduce()

Встроенные функции `map()`, `filter()` и `reduce()` часто используются в сочетании с лямбда-функциями. Они обычно применяются при программировании на

Python в функциональном стиле, потому что позволяют объявлять преобразования произвольной сложности, избегая при этом побочных эффектов.

В Python 2 все три функции были доступны как стандартные встроенные функции без дополнительного импортирования. В Python 3 функция `reduce()` перемещена в модуль `functools`, поэтому теперь ее надо импортировать.

Функция `map(func, iterable, ...)` применяет функцию-аргумент `function` к каждому элементу итерируемого объекта `iterable`. Если передать фупкции `map()` более одного итерируемого объекта, она будет обрабатывать элементы всех этих объектов одновременно. На каждом шаге фупкция `func` принимает по одному следующему аргументу от каждого итерируемого источника. Если у итерируемых объектов разная длина, то `map()` останавливается, когда заканчиваются элементы в самом коротком из них. Следует помнить, что `map()` не вычисляет весь результат сразу, а возвращает итератор, чтобы каждый элемент результата вычислялся только тогда, когда это необходимо.

В следующем примере с помощью `map()` вычисляются квадраты первых десяти целых чисел, начиная с 0:

```
>>> map(lambda x: x**2, range(10))
<map object at 0x10ea09cf8>
>>> list(map(lambda x: x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

А вот пример использования функции `map()` с несколькими итерируемыми объектами разных размеров:

```
>>> mapped = list(map(print, range(5), range(4), range(5)))
0 0 0
1 1 1
2 2 2
3 3 3
>>> mapped
[None, None, None, None]
```

Как и `map()`, функция `filter(func, iterable)` обрабатывает входные элементы по очереди. Но, в отличие от `map()`, функция `filter()` не преобразует их в новые значения, а отфильтровывает входные элементы, которые не удовлетворяют предикату, определенному аргументом `func`. Вот примеры использования фупкции `filter()`:

```
>>> evens = filter(lambda number: number % 2 == 0, range(10))
>>> odds = filter(lambda number: number % 2 == 1, range(10))
>>> print(f"Четные числа в диапазоне от 0 до 9: {list(evens)}")
```

```
Четные числа в диапазоне от 0 до 9: [0, 2, 4, 6, 8]
>>> print(f"Нечетные числа в диапазоне от 0 до 9: {list(odds)}")
Нечетные числа в диапазоне от 0 до 9: [1, 3, 5, 7, 9]
>>> animals = ["мартышка", "осел", "козел", "медведь"]
>>> animals_s = filter(lambda animal: animal.startswith('м'), animals)
>>> print(f"Животные на букву 'м': {list(animals_s)}")
Животные на букву 'м': ['мартышка', 'медведь']
```

Функция `reduce(func, iterable)` работает полностью противоположно `map()`. Как можно догадаться по названию, она выполняет свертку итерируемого объекта в одно значение. Вместо того чтобы отображать элементы `iterable` по одному на возвращаемые значения `func`, функция кумулятивно применяет операцию, определяемую `func`, ко всем элементам `iterable`. Таким образом, для следующих входных данных `reduce()`:

```
reduce(func, [a, b, c, d])
```

возвращаемое значение будет равно:

```
func(func(func(a, b), c), d)
```

Для примера рассмотрим, как можно использовать `reduce()`, чтобы суммировать значения элементов из различных итерируемых объектов:

```
>>> from functools import reduce
>>> reduce(lambda a, b: a + b, [2, 2])
4
>>> reduce(lambda a, b: a + b, [2, 2, 2])
6
>>> reduce(lambda a, b: a + b, range(100))
4950
```

Одна из интересных особенностей `map()` и `filter()` заключается в том, что они могут работать с бесконечными последовательностями. Конечно, если пытаться преобразовать бесконечную последовательность в объект типа `list` или перебирать ее в обычном цикле, программа никогда не завершится. `count()` из `itertools` — пример функции, которая возвращает бесконечные итерируемые объекты. Она просто ведет отсчет от 0 до бесконечности. Если пытаться перебрать сгенерированные элементы в цикле, как в следующем примере, программа будет выполняться бесконечно:

```
from itertools import count

for i in count():
    print(i)
```

Однако возвращаемые значения `map()` и `filter()` представляют собой итераторы. Вместо цикла `for` можно обращаться к последовательным элементам

итератора функцией `next()`. Вспомним предыдущий вызов `map()`, который генерировал квадраты последовательных целых чисел, начиная с 0:

```
map(lambda x: x**2, range(n))
```

Функция `range()` возвращает ограниченный итерируемый объект из `n` элементов. Если заранее неизвестно, сколько элементов надо сгенерировать, можно просто заменить ее вызовом `count()`:

```
map(lambda x: x**2, count())
```

После этого можно извлекать последовательные квадраты чисел. Использовать цикл `for` нельзя, потому что он никогда не закончится. Однако можно вызывать `next()` многократно и обрабатывать элементы по одному:

```
sequence = map(lambda x: x**2, count())
next(sequence)
next(sequence)
next(sequence)
...
```

В отличие от функций `map()` и `filter()`, функция `reduce()` должна вычислить все входные элементы для того, чтобы вернуть значение, и она не возвращает промежуточные результаты. Это означает, что ее нельзя использовать с бесконечными последовательностями.

Частичные объекты и частичные функции

Частичные объекты отдаленно связаны с концепцией частичных функций в математике. Частичная функция — это обобщение математической функции, которое не обязано отображать всю область отправления на область прибытия. В Python частичные объекты можно использовать, чтобы сегментировать диапазон входных значений заданной функции, присваивая фиксированное значение некоторым из ее аргументов.

В предыдущих разделах квадрат числа `x` вычислялся с помощью выражения `x ** 2`. В Python есть встроенная функция `pow(x, y)`, которая вычисляет произвольную степень произвольного числа. Таким образом, выражение `lambda x: x ** 2` является частичной функцией по отношению к `pow(x, y)`, потому что множество допустимых значений `y` ограничивается единственным числом `2`. С помощью функции `partial()` из модуля `functools` такие частичные функции можно определять альтернативным способом, не используя лямбда-выражения, которые иногда становятся слишком громоздкими.

Допустим, теперь мы хотим создать другую частичную функцию на базе `pow()`. В прошлый раз мы генерировали квадраты последовательных чисел. Теперь

сузим область допустимых значений других входных аргументов, чтобы получить последовательность степеней числа 2, то есть 1, 2, 4, 8, 16 и т. д.

Сигнатура конструктора частичного объекта имеет вид `partial(func, *args, **keywords)`. Частичный объект ведет себя в точности как `func`, но его входные аргументы предварительно заполняются значениями `*args` (слева направо) и `**keywords`. Функция `pow(x, y)` не поддерживает именованные аргументы, поэтому левый аргумент `x` приходится заполнять следующим образом:

```
>>> from functools import partial
>>> powers_of_2 = partial(pow, 2)
>>> powers_of_2(2)
4
>>> powers_of_2(5)
32
>>> powers_of_2(10)
1024
```

Обратите внимание: если вы не собираетесь использовать частичный объект повторно, его не обязательно связывать с каким-либо идентификатором. С его помощью можно успешно определять одноразовые функции по аналогии с тем, как вы бы использовали лямбда-выражения.

Модуль itertools — настоящая сокровищница вспомогательных средств для перебора итерируемых объектов любых типов всевозможными способами. В этом модуле доступны многочисленные функции, которые, помимо прочего, позволяют перебирать контейнеры в цикле, группировать их содержимое, разбивать итерируемые объекты на фрагменты и объединять несколько итерируемых объектов в один. Каждая функция в этом модуле возвращает итераторы. Если вам интересно программировать на Python в функциональном стиле, вам определенно стоит поближе познакомиться с itertools. Документация этого модуля доступна по адресу https://docs.python.org/3/library/itertools.html.

Генераторы

С помощью генераторов можно элегантно писать простой и эффективный код для функций, которые возвращают последовательность элементов. Благодаря инструкции `yield` они позволяют приостановить работу функции и вернуть промежуточный результат. Функция сохраняет контекст своего исполнения и может возобновить работу позже, если потребуется.

Например, можно использовать синтаксис генераторов, чтобы написать функцию, которая возвращает последовательные числа Фибоначчи. Следующий пример позаимствован из документа PEP 255 («Простые генераторы»):

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b
```

От генераторов можно получать новые значения так, как если бы они были итераторами, то есть с помощью функции `next()` или циклов `for`:

```
>>> fib = fibonacci()
>>> next(fib)
1
>>> next(fib)
1
>>> next(fib)
2
>>> for item in fibonacci():
...     print(item)
...     if item > 10:
...         break
...
1
1
2
3
5
8
13
```

Функция `fibonacci()` возвращает объект `generator` — итератор специального вида, который умеет сохранять контекст выполнения. Его можно вызывать сколько угодно раз, нри каждом вызове получая следующий элемент последовательности. Синтаксис генераторов весьма лаконичен, а бесконечная природа алгоритма не парушает удобочитаемость кода. В этой функции не обязательно обеспечивать возможность остановки. Собственно, код очень похож на то, как функция генерирования последовательности была бы описана на псевдокоде.

Как правило, для обработки одного элемента требуется меньше ресурсов, чем для хранения целых последовательностей. Чем меньше ресурсов тратит программа, тем она эффективнее. Например, последовательность Фибоначчи бесконечна, но генератору, который ее производит, пе нужен бесконечный объем памяти: он выдает значения одно за другим и теоретически может работать неограниченно долго. Типичный сценарий использования генераторов — потоковая передача буферов данных (например, из файлов). Геператоры можно приостанавливать, возобновлять и завершать па любой стадии обработки данных без необходимости загружать полные наборы данных в память программы.

В функциональном программировании с помощью генераторов можно реализовать функцию с сохранением состояния, которой иначе пришлось бы сохранять промежуточные результаты как побочный эффект, словно это функция без сохранения состояния.

Генераторные выражения

Генераторные выражения — еще один элемент синтаксиса, который позволяет писать код в более функциональном стиле. Они похожи на включения (comprehensions), которые используются с литералами словарей, множеств и списков. Генераторное выражение обозначается круглыми скобками, как в следующем примере:

```
(элемент for элемент in итерируемое_выражение)
```

Генераторные выражения можно использовать как входные аргументы любых функций, которые принимают итерируемые объекты. Эти выражения также позволяют условиям `if` фильтровать отдельные элементы по тем же принципам, что и включения списков, словарей и множеств. Таким образом, вместо сложных конструкций с `map()` и `filter()` часто можно использовать более удобочитаемые и компактные генераторные выражения.

Синтаксически генераторные выражения не отличаются от любых других выражений с включениями. Их главное преимущество в том, что они вычисляют только один элемент за раз. Таким образом, если вы обрабатываете итерируемое выражение произвольной длины, вам может пригодиться генераторное выражение, потому что оно не требует размещать весь набор промежуточных результатов в памяти программы.

Лямбда-выражения, `map`, `reduce`, `filter`, частичные функции и генераторы нацелены на то, чтобы представлять логику программы как процесс вычисления выражений вызовов функций. Другой важный аспект функционального программирования — наличие функций первого класса. В Python все функции являются объектами, и их, как и любые другие объекты, можно анализировать и изменять во время выполнения. На этом основан мощный синтаксический инструмент — **декораторы функций**.

Декораторы

В общем случае декоратор — это вызываемое выражение, которое принимает при вызове один аргумент (декорируемую функцию) и возвращает другой вызываемый объект.

До выхода Python 3.9 специальный синтаксис декораторов был доступен только для именованных выражений. Начиная с Python 3.9, в синтаксис декораторов можно подставить любое выражение, включая лямбда-выражения.

Декораторы часто рассматриваются в контексте методов и функций, однако не ограничиваются этим контекстом. Собственно, любой вызываемый объект (то есть объект, который реализует метод `__call__`) может использоваться как декоратор. Объекты, возвращаемые декораторами, часто представляют собой не простые функции, а экземпляры более сложных классов, которые реализуют собственный метод `__call__`.

Синтаксис декораторов — не более чем «синтаксический сахар». Рассмотрим следующий пример применения декоратора:

```
@some_decorator
def decorated_function():
    pass
```

Эту конструкцию всегда можно заменить явным вызовом декоратора и переприсвоением функции:

```
def decorated_function():
    pass
decorated_function = some_decorator(decorated_function)
```

Однако второй вариант хуже читается, и его очень трудно понять, если с одной функцией используются несколько декораторов.

Декоратор даже не обязан возвращать вызываемый объект!

Собственно, любую функцию можно использовать как декоратор, потому что Python не устанавливает ограничений на то, какой тип может возвращать декоратор. Таким образом, синтаксис абсолютно не запрещает использовать в качестве декоратора функцию, которая принимает один аргумент, а возвращает объект, не являющийся вызываемым,— допустим, `str`. Если попытаться вызвать объект, декорированный таким способом, это в конце концов завершится неудачей. Эта особенность синтаксиса декораторов открывает возможности для интересных экспериментов.

Декораторы — это элементы языка программирования, вдохновленные аспектно-ориентированным программированием и паттерном проектирования «Декоратор». Они предназначены прежде всего для того, чтобы элегантно расширять существующую реализацию функций дополнительным поведением, взятым из других частей приложения.

Рассмотрим следующий пример из документации фреймворка Flask:

```
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

`secret_page()` — функция представления, которая, по всей видимости, должна возвращать секретную страницу. Она снабжена двумя декораторами. `app.route()` назначает маршрут URI для функции представления, а `login_required()` требует аутентификации пользователя.

Согласно нринципу единственной ответственности, функции должны быть небольшими и предназначенными для одной цели. В нашем нриложении Flask функция представлепия `secret_page()` отвечает за подготовку ответа HTTP, который нозже может открыться в браузере. Вероятно, она не должпа заниматься посторонними задачами: разбирать запросы HTTP, проверять учетные данные пользователя и т. д.

Как подсказывает название, фупкция `secret_page()` возвращает некую конфиденциальную информацию, которая не должна быть достунна кому попало. Проверка учетных данных пользователя не входит в зону ответственности фупкции представления, однако является частью общей идеи «секретной страницы». Декоратор `@login_required` приближает аутентификацию пользователя к функции представления. Код приложения становится лаконичнее, а намерения программиста проявляются более наглядпо.

Рассмотрим реальный пример декоратора `@login_required` из документации фреймворка Flask:

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

Декоратор `@wraps` позволяет скопировать такие метаданные декорированной функции, как имя и аннотации типов. Использовать декоратор `@wraps` в ваших декораторах считается хорошим тоном, потому что это упрощает отладку и предоставляет доступ к аннотациям типов исходной функции.

Как видите, этот декоратор возвращает новую функцию `decorated_function()`, которая прежде всего выясняет, связан ли с глобальным объектом `g` действительный пользователь. Это распространенный способ проверить, прошел ли пользователь аутентификацию во Flask. Если проверка успешна, то декорируемая функция вызывает исходную функцию, возвращая `f(*args, **kwargs)`. Если проверка аутентификации не удается, то декорируемая функция перенаправляет браузер на страницу входа.

Таким образом, декоратор `login_required()` обеспечивает нечто большее, чем простую проверку «прошел/не прошел». Декораторы становятся отличным механизмом новторного использования кода. Аутентификация может быть стандартной составной частью приложений, но ее реализация может изменяться со временем. Декораторы предоставляют удобный способ упаковывать такие компоненты в переносимые «единицы поведения», которые можно легко добавлять поверх существующих функций.

Декораторы с примерами их практического применения будут подробнее представлены в главе 8 «Элементы метапрограммирования», где мы рассмотрим их как один из приемов метапрограммирования.

Перечисления

Некоторые распространенные концепции программирования встречаются во многих языках независимо от преобладающей в них парадигмы программирования. Одна из таких концепций — перечисляемые типы (перечисления), имеющие конечное число именованных значений. С их помощью особенно удобно кодировать замкнутое множество значений переменных или аргументов функций.

Среди вспомогательных типов стандартной библиотеки Python можно выделить класс `Enum` из модуля `enum`. Это базовый класс, который позволяет определять символические перечисления, подобные типам перечислений из прочих языков программирования (C, C++, C#, Java и многих других), где такие типы часто обозначаются ключевым словом `enum`.

Чтобы определить собственное перечисление в Python, нужно создать подкласс класса `Enum` и определить все элементы перечисления как атрибуты класса. Вот пример простого перечисления Python:

```
from enum import Enum

class Weekday(Enum):
    MONDAY = 0
    TUESDAY = 1
```

```
    WEDNESDAY = 2
    THURSDAY = 3
    FRIDAY = 4
    SATURDAY = 5
    SUNDAY = 6
```

В документации Python определена такая терминология перечислений:

- **перечисленне** (enum): подкласс базового класса `Enum` (в этом примере — `Weekday`).
- **элемент**: атрибут, определенный в подклассе `Enum` (в этом примере — `Weekday.MONDAY`, `Weekday.TUESDAY` и т. д.).
- **имя**: имя атрибута в подклассе `Enum`, определяющего элемент перечисления (в этом примере `MONDAY` для `Weekday.MONDAY`, `TUESDAY` для `Weekday.TUESDAY` и т. д.).
- **значенне**: значение, присвоенное атрибуту подкласса `Enum`, определяющему элемент перечислепия (в этом примере для `Weekday.MONDAY` значение равно 1, для `Weekday.TUESDAY` оно равно 2 и т. д.).

Значения элементов перечисления могут быть любого типа. Если значения элементов в вашем коде не играют роли, можно использовать тип `auto()`, который будет заменен автоматически генерируемыми значениями. Вот как можно записать предыдущий пример, используя `auto`:

```
from enum import Enum, auto

class Weekday(Enum):
    MONDAY = auto()
    TUESDAY = auto()
    WEDNESDAY = auto()
    THURSDAY = auto()
    FRIDAY = auto()
    SATURDAY = auto()
    SUNDAY = auto()
```

Перечисления в Python очепь полезны во всех ситуациях, когда переменная может принимать только конечное число значений. Например, по этой схеме можно определить статусы объектов, как в следующем примере:

```
from enum import Enum, auto

class OrderStatus(Enum):
    PENDING = auto()
    PROCESSING = auto()
    PROCESSED = auto()

class Order:
    def __init__(self):
```

```
        self.status = OrderStatus.PENDING

    def process(self):
        if self.status == OrderStatus.PROCESSED:
            raise ValueError(
                ""Невозможно обработать заказ, ""
                ""который уже был обработан""
            )

        self.status = OrderStatus.PROCESSING
        ...
        self.status = OrderStatus.PROCESSED
```

С помощью перечислений также можно хранить варианты множественного выбора. В языках, где поразрядные манипуляции с числами являются обычным делом (например, в C), нечто подобное часто реализуется с использованием флагов и масок. В Python это делается более выразительно и удобно с помощью базового класса перечислений `Flag`:

```
from enum import Flag, auto

class Side(Flag):
    GUACAMOLE = auto()
    TORTILLA = auto()
    FRIES = auto()
    BEER = auto()
    POTATO_SALAD = auto()
```

Можно комбинировать такие флаги поразрядными онераторами (`|` и `&`), а также проверять принадлежность элемента тому или иному флагу ключевым словом `in`. Вот примеры использования перечисления `Side`:

```
>>> mexican_sides = Side.GUACAMOLE | Side.BEER | Side.TORTILLA
>>> bavarian_sides = Side.BEER | Side.POTATO_SALAD
>>> common_sides = mexican_sides & bavarian_sides
>>> Side.GUACAMOLE in mexican_sides
True
>>> Side.TORTILLA in bavarian_sides
False
>>> common_sides
<Side.BEER: 8>
```

Символические перечисления в чем-то похожи на словари и именованные кортежи, потому что они тоже связывают имена (ключи) со значениями. Главное отличие состоит в том, что определение `Enum` является неизменяемым и глобальным. Перечисления имеет смысл использовать в тех случаях, когда существует замкнутый набор допустимых значений, который не может динамически изменяться во время выполнения программы, — особенно если этот набор должен

определяться однократно и глобально. В то же время словари и именованные кортежи — это контейнеры данных. Вы можете создать столько их экземпляров, сколько пожелаете.

Итоги

В этой главе мы рассмотрели язык Python сквозь призму разных парадигм программирования. Там, где это имело смысл, мы старались сравнивать Python с другими языками со сходной функциональностью, чтобы наглядно продемонстрировать сильные и слабые стороны Python.

Мы достаточно глубоко погрузились в конценции объектно-ориентированного программирования, а также затронули такую дополнительную парадигму, как функциональное нрограммирование. Теперь вы полностью готовы перейти к вопросам структурирования и формирования архитектуры целых приложений.

В следующей главе эта тема раскрывается достаточно полно, потому что глава будет целиком посвящена различным паттерпам и методологиям программирования.

5

Интерфейсы, паттерны и модульность

В этой главе мы погрузимся в архитектуру приложений и рассмотрим ее сквозь призму интерфейсов, паттернов и модульности. Мы уже приближались к этой теме, когда упоминали концепцию идиом программирования. Идиомы можно понимать как небольшие общепризнанные паттерны программирования для решения локальных задач. Ключевое свойство всякой идиомы состоит в том, что она относится к какому-то конкретному языку программирования. Хотя идиомы из одного языка часто можно перенести в другой, нельзя гарантировать, что итоговый код будет выглядеть естественно для «носителей» этого языка.

Идиомы обычно выражаются в небольших программных конструкциях — как правило, из нескольких строк кода. А паттерны проектирования имеют дело с гораздо бóльшими программными структурами — функциями и классами, и при этом они гораздо более универсальны. Паттерны — это типовые решения многих распространенных задач проектирования ПО. Часто они не зависят от языка программирования, а следовательно, могут выражаться на разных языках.

В этой главе тема паттернов проектирования рассматривается под довольно необычным углом. Многие книги по программированию начинают со ссылок на неофициальный первоисточник сведений о паттернах — книгу Гаммы, Влиссидеса, Хелма и Джонсона (Gamma, Vlissides, Helm, and Johnson) «Design Patterns: Elements of Reusable Object-Oriented Software»[1]. Затем обычно следует длинный

[1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Паттерны объектно-ориентированного проектирования». — СПб., издательство «Питер».

перечень классических паттернов с более или менее идиоматическими примерами их реализации в Python: Одиночка, Фабрика, Адаптер, Приспособлепец, Мост, Посетитель, Стратегия и т. д.

Еще существуют бесчисленные веб-статьи и блоги с точно такой же подачей материала, так что если вас интересуют классические паттерны проектирования, вы сможете без особых проблем найти источники информации в интернете.

Если вам интересно, как реализовывать «классические» паттерны проектирования на Python, посетите сайт https://python-patterns.guide. Там представлен обширный каталог паттернов с примерами кода на Python.

Мы пойдем другим путем и сосредоточимся на двух нервоосновах паттернов проектирования:

- интерфейсы;
- инверсия управления и внедрение зависимостей.

Эти концепции названы первоосновами, потому что без них даже не было бы необходимых языковых средств, чтобы реализовывать паттерны. Рассматривая темы интерфейсов и инверсии управления, мы сможем лучше понять, какие проблемы возникают при построении модульных приложений. И только глубокое понимание этих проблем позволит разобраться, для чего же па самом деле нужны паттерны.

Конечно, попутно мы будем использовать мпогие классические наттерны проектирования, но пе станем сосредоточиваться на каком-то конкретном паттерне.

Технические требования

Ниже перечислены пакеты Python, которые упоминаются в этой главе и доступны для загрузки из PyPI:

- `zope.interface`
- `mypy`
- `redis`
- `flask`
- `injector`
- `flask-injector`

Информация об установке пакетов приведепа в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%205.

Интерфейсы

В широком смысле интерфейс — это посредник во взаимодействии между двумя сущностями. Например, интерфейс автомобиля состоит в основном из руля, педалей, рычага переключения передач, приборной панели, различных переключателей и т. д. К интерфейсу компьютера традиционно относятся мышь, клавиатура и экран монитора.

В программировании термин «интерфейс» может обозначать два понятия:

- Полное пространство средств взаимодействия с кодом.
- Абстрактное определение возможных способов взаимодействия с кодом, умышленно отделенное от его реализации.

В первом значении интерфейс представляет собой конкретный набор символических имен, который используется для взаимодействия с программной единицей. Например, интерфейс функции состоит из имени этой функции, ее входных аргументов и возвращаемого результата. В интерфейс объекта входят все его методы, которые можно вызывать, и все атрибуты, к которым можно обращаться.

Программные единицы (функции, объекты, классы) часто объединяются в библиотеки. В Python библиотеки имеют форму модулей и пакетов (наборов модулей). Кроме того, у них есть интерфейсы. Содержимое модулей и пакетов обычно используется в разных комбинациях, и вам не обязательно взаимодействовать со всем этим содержимым. Интерфейсы библиотек, как и других программных систем, часто обозначаются термином **API** (Application Programming Interface, интерфейс прикладного программирования).

Термин «интерфейс» в этом значении можно расширить на другие компоненты компьютерных технологий. Интерфейсы операционных систем представлены в форме файловых систем и системных функций, а интерфейсы сетевых служб — в форме коммуникационных протоколов.

Второе значение термина «интерфейс» можно рассматривать как формализацию первого. Здесь интерфейс понимается как контракт, который тот или иной элемент кода обязуется выполнять. Такие формальные интерфейсы можно абстрагировать от реализации и рассматривать как автономные сущности. Это позволяет разрабатывать приложения, которые опираются на конкретный интерфейс, но не зависят от его фактической реализации — при условии, что она существует и выполняет контракт.

Формальный смысл интерфейса тоже можно расширить для более крупных программных концепций:

- **Библиотеки**: язык программирования C определяет API стандартной библиотеки этого языка, которая также называется библиотекой ISO C. В отличие от Python, у стандартной библиотеки C есть много реализаций. Самая распространенная реализация для Linux — вероятно, библиотека GNU C (glibc), но у нее есть альтернативы, например dietlibc или musl. В других операционных системах применяются собственные реализации ISO C.
- **Операционная система**: POSIX (Portable Operating System Interface) — это набор стандартов, определяющих единый интерфейс для операционных систем. Многие системы прошли сертификацию на соответствие этому стандарту (например, macOS или Solaris). Также существуют ОС, которые в основном совместимы с POSIX: Linux, Android, OpenBSD и многие другие. Вместо того чтобы называть такие системы «POSIX-совместимыми», можно сказать, что они реализуют интерфейс POSIX.
- **Веб-службы**: OIDC (OpenID Connect) — открытый стандарт для фреймворков аутентификации и авторизации, базирующийся на протоколе OAuth 2.0. Службы, которые хотят реализовать стандарт OIDC, должны предоставить конкретные четко определенные интерфейсы, описанные в этом стандарте.

Формальные интерфейсы — исключительно важная концепция в объектно-ориентированных языках программирования. В этом контексте интерфейс абстрагирует либо форму, либо назначение моделируемого объекта. Обычно он описывает набор методов и атрибутов, которые класс должен реализовать, обеспечив желаемое поведение.

При «академическом» подходе определение интерфейса не предоставляет никакой полезной реализации методов. Оно всего лишь описывает явный контракт для любого класса, который пожелает реализовать этот интерфейс. Интерфейсы часто компонуются, то есть один класс может реализовывать сразу несколько интерфейсов. При таком подходе интерфейсы становятся ключевыми структурными элементами паттернов проектирования: каждый паттерн можно рассматривать как результат компоновки определенных интерфейсов. У паттернов проектирования, как и у интерфейсов, нет нормативной реализации. По сути, это всего лишь обобщенные схемы, которые помогают разработчикам решать стандартные задачи.

Разработчики на Python предпочитают **утиную типизацию** явному определению интерфейсов. Однако когда есть четко определенные контракты взаимодействия между классами, это часто улучшает общее качество продукта и сокращает число потенциальных ошибок. В частности, создатели новой реализации интер-

фейса получают четкий список методов и атрибутов, которые тот или иной класс должен предоставлять. При грамотной реализации невозможно забыть о методе, если он явно значится в требованиях конкретного интерфейса.

Поддержка абстрактных интерфейсов — краеугольный камень многих языков со статической типизацией. Например, в Java существуют типажи (traits) — явные объявления о том, что класс реализует тот или иной интерфейс. Это позволяет программистам Java добиваться полиморфизма без наследования типов, которое иногда оказывается затруднительным. С другой стороны, в Go нет классов и не поддерживается наследование типов, но интерфейсы в Go позволяют реализовать некоторые паттерны ООП и полиморфизм без этого наследования. В обоих языках интерфейсы ведут себя как явная версия утиной типизации: как в Java, так и в Go они применяются, чтобы проверять безопасность типов во время компиляции, вместо того чтобы с помощью утиной типизации связывать компоненты во время выполнения.

В Python используется совершенно иная философия типизации, чем в этих языках, поэтому в нем отсутствует встроенная поддержка проверки интерфейсов во время компиляции. Впрочем, если вам нужен более явный контроль над прикладными интерфейсами, для этого есть несколько вариантов:

- Сторонние фреймворки (такие, как `zope.interface`), добавляющие концепцию интерфейсов.
- Абстрактные базовые классы (ABC, Abstract Base Classes).
- Аннотации типов, `typing.Protocol` и статические анализаторы типов.

Все эти решения будут подробно рассмотрены в следующих разделах.

Немного истории: zope.interface

Существует несколько фреймворков, которые позволяют строить явные интерфейсы в Python. Самый заметный из них — проект Zope, то есть пакет `zope.interface`. В наши дни Zope уже не так популярен, как десять лет назад, однако `zope.interface` остается одним из главных компонентов все еще востребованного фреймворка Twisted. `zope.interface` — также один из самых старых интерфейсных фреймворков, который при этом все еще активно поддерживается и широко используется в Python. Он появился до таких общепринятых средств Python, как ABC, поэтому мы начнем с него, а затем сравним его с другими решениями для работы с интерфейсами.

Пакет `zope.interface` был создан Джимом Фултоном, чтобы имитировать функциональность тогдашних интерфейсов Java.

Концепция интерфейса лучше всего подходит для ситуаций, в которых одна абстракция может иметь несколько реализаций или применяться к разным объектам, которые не стоит объединять в структуру наследования. Чтобы лучше продемонстрировать эту идею, рассмотрим пример задачи с участием нескольких сущностей, которые имеют ряд общих качеств, но при этом не сводятся к одной и той же сущности.

Попробуем разработать простую систему, которая обнаруживает геометрические перекрытия между несколькими объектами. Такая функциональность может пригодиться, например, в простой игре или симуляторе. Наше решение будет достаточно тривиальным и малоэффективным. Помните, что здесь наша цель — изучить концепцию интерфейсов, а не построить идеальный движок столкновений для игрового блокбастера.

Мы будем использовать алгоритм, который называется AABB (Axis-Aligned Bounding Box, «ограпичительный нрямоугольник, выровненпый по осям»). Это простой способ обнаружения перекрытий двух нрямоугольников, которые выровнены по координатным осям (то есть без поворотов). Предполагается, что каждый проверяемый элемент можно ограничить прямоугольной областью. Алгоритм довольно нрост: достаточпо сравпить четыре координаты прямоугольника (рис. 5.1).

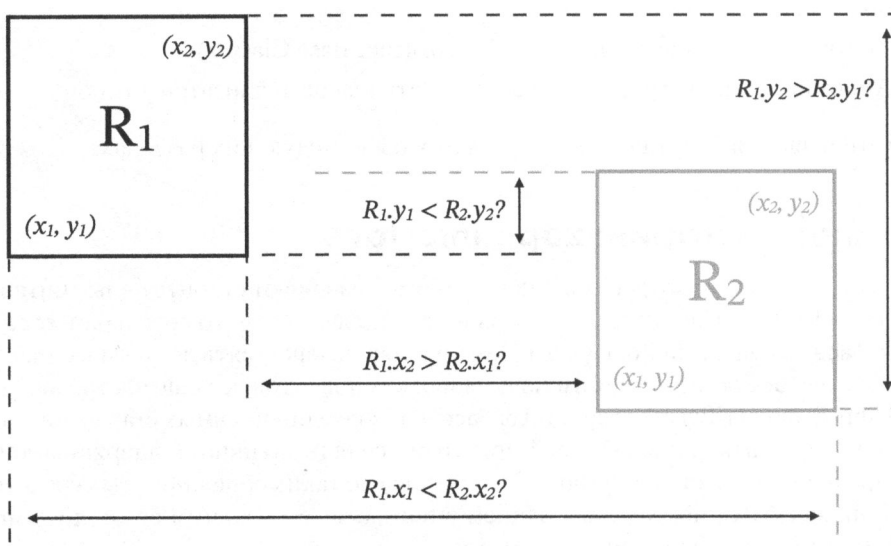

Рис. 5.1. Сравнение координат прямоугольников в алгоритме AABB

Начпем с простой функции, которая проверяет, перекрываются ли два прямоугольника:

```
def rects_collide(rect1, rect2):
    """Проверяет перекрытие прямоугольников

    Координаты прямоугольников:
        ┌─────(x2, y2)
        │           │
        (x1, y1) ────┘
    """
    return (
        rect1.x1 < rect2.x2 and
        rect1.x2 > rect2.x1 and
        rect1.y1 < rect2.y2 and
        rect1.y2 > rect2.y1
    )
```

Мы не определили аннотации типов, но из приведенного кода должно быть абсолютно ясно, что у обоих аргументов функции `rects_collide()` должно быть по четыре атрибута: `x1`, `y1`, `x2`, `y2`. Эти атрибуты соответствуют координатам нижнего левого и верхнего правого угла ограничительного прямоугольника.

Имея функцию `rects_collide()`, можно определить еще одну функцию, которая обнаруживает все перекрытия внутри группы объектов. Функция получается очень простой:

```
import itertools

def find_collisions(objects):
    return [
        (item1, item2)
        for item1, item2
        in itertools.combinations(objects, 2)
        if rects_collide(
            item1.bounding_box,
            item2.bounding_box
        )
    ]
```

Остается определить какие-нибудь классы или объекты, которые можно проверить на перекрытие. Смоделируем несколько фигур: квадрат, прямоугольник и круг. Все фигуры разные, поэтому их внутренняя структура различается. Трудно найти подходящий класс, который мог бы стать их общим предком. Чтобы не усложнять задачу, мы воспользуемся классами данных и свойствами. Вот все исходные определения:

```
from dataclasses import dataclass

@dataclass
class Square:
    x: float
    y: float
```

```
    size: float

    @property
    def bounding_box(self):
        return Box(
            self.x,
            self.y,
            self.x + self.size,
            self.y + self.size
        )

@dataclass
class Rect:
    x: float
    y: float
    width: float
    height: float

    @property
    def bounding_box(self):
        return Box(
            self.x,
            self.y,
            self.x + self.width,
            self.y + self.height
        )

@dataclass
class Circle:
    x: float
    y: float
    radius: float

    @property
    def bounding_box(self):
        return Box(
            self.x - self.radius,
            self.y - self.radius,
            self.x + self.radius,
            self.y + self.radius
        )
```

Эти классы объединяет только одно (помимо того факта, что они все являются классами данных): у них есть свойство `bounding_box`, которое возвращает экземпляр класса `Box`. Этот класс тоже является классом данных:

```
@dataclass
class Box:
    x1: float
    y1: float
    x2: float
    y2: float
```

Определения классов данных достаточно просты и не требуют особых пояснений. Чтобы проверить, работает ли система, можно передать групну экземнляров функции `find_collisions()`, как в следующем нримере:

```
for collision in find_collisions([
    Square(0, 0, 10),
    Rect(5, 5, 20, 20),
    Square(15, 20, 5),
    Circle(1, 1, 2),
]):
    print(collision)
```

Если все было сделано верно, этот код выдаст следующий результат с тремя перекрытиями:

```
(Square(x=0, y=0, size=10), Rect(x=5, y=5, width=20, height=20))
(Square(x=0, y=0, size=10), Circle(x=1, y=1, radius=2))
(Rect(x=5, y=5, width=20, height=20), Square(x=15, y=20, size=5))
```

Все хорошо, но давайте нроведем мыслешный эксперимент. Представьте, что приложение немного расширилось и обогатилось новыми элементами. Например, если это игра, то в ней могут появиться объекты, представляющие спрайты, предметы или частицы из эффектов. Допустим, кто-то определил такой класс `Point` (точка):

```
@dataclass
class Point:
    x: float
    y: float
```

Что произойдет, если включить экземпляр этого класса в список объектов для проверки нерекрытий? Скорее всего, возникнет исключение с трассировкой следующего вида:

```
Traceback (most recent call last):
  File "/.../simple_colliders.py", line 115, in <module>
    for collision in find_collisions([
  File "/.../simple_colliders.py", line 24, in find_collisions
    return [
  File "/.../simple_colliders.py", line 30, in <listcomp>
    item2.bounding_box
AttributeError: 'Point' object has no attribute 'bounding_box'
```

Трассировка дает представление о нричинах проблемы. Вопрос в том, можно ли было улучшить код, чтобы обнаруживать такие проблемы на более ранней стадии? Как минимум, мы могли бы проверить функции `bounding_box()` всех входных объектов и убедиться, что для них доступна проверка перекрытий. Но как это сделать?

Так как у интересующих нас классов нет общего предка, мы не сможем легко воспользоваться функцией `isinstance()`, чтобы проверить соответствие их типов. Функция `hasattr()` сможет проверить наличие атрибута `bounding_box`, но тогда придется еще убеждаться, что он имеет правильную структуру, и код быстро станет неуклюжим.

В таких ситуациях на помощь приходит `zope.interface`. Центральное место в этом пакете занимает класс `Interface`, который позволяет явно определить новый интерфейс. Определим класс `ICollidable` так, чтобы он представлял любой объект, который может участвовать в нашей системе перекрытий:

```
from zope.interface import Interface, Attribute

class ICollidable(Interface):
    bounding_box = Attribute("Ограничительный прямоугольник объекта")
```

По стандартным соглашениям в Zope имена классов интерфейсов начинаются с префикса `I`. Конструктор `Attribute` обозначает требуемый атрибут объектов, которые реализуют интерфейс. Любой метод, определенный в классе интерфейса, будет использоваться как объявление метода интерфейса. Эти методы должны быть пустыми. По общепринятому соглашению в них содержится только doc-строка тела метода.

После того как такой интерфейс определен, необходимо указать, какие конкретно из ваших классов его реализуют. Интерфейсы такого рода называются **явными интерфейсами** и по своей природе близки к типажам в Java. Чтобы обозначить реализацию конкретного интерфейса, нужно использовать декоратор класса `implementer()`. В нашем примере это выглядит так:

```
from zope.interface import implementer

@implementer(ICollidable)
@dataclass
class Square:
    ...

@implementer(ICollidable)
@dataclass
class Rect:
    ...

@implementer(ICollidable)
@dataclass
class Circle:
    ...
```

Тела классов данных в этом примере были сокращены, чтобы сэкономить место.

Часто говорят, что интерфейс определяет контракт, который должна выполнять конкретная реализация. Главное преимущество такого паттерна в том, что перед использованием объекта можно проверить, согласованы ли контракт и реализация друг с другом. При обычном подходе с утиной тинизацией вы узнаёте о проблемах, только когда во время выполнения обнаружится, что нужный атрибут или метод отсутствует.

Чтобы выявить расхождения заранее, можно проанализировать фактическую реализацию с номощью двух методов модуля `zope.interface.verify`:

- `verifyClass(interface, class_object)`: проверяет, есть ли в объекте класса требуемые методы и правильны ли их сигнатуры, но не обращается к атрибутам.
- `verifyObject(interface, instance)`: проверяет методы, их сигнатуры, а также атрибуты фактического объекта экземпляра.

Таким образом, функцию `find_collisions()` можно расширить, чтобы она выполняла нервичную проверку интерфейсов объектов до того, как выявлять пересечения. Это может выглядеть примерно так:

```
from zope.interface.verify import verifyObject

def find_collisions(objects):
    for item in objects:
        verifyObject(ICollidable, item)
    ...
```

Если тенерь передать функции `find_collisions()` экземпляр класса, у которого нет декоратора `@implementer(ICollidable)`, возникнет исключение с трассировкой:

```
Traceback (most recent call last):
  File "/.../colliders_interfaces.py", line 120, in <module>
    for collision in find_collisions([
  File "/.../colliders_interfaces.py", line 26, in find_collisions
    verifyObject(ICollidable, item)
  File "/.../site-packages/zope/interface/verify.py", line 172, in
verifyObject
    return _verify(iface, candidate, tentative, vtype='o')
  File "/.../site-packages/zope/interface/verify.py", line 92, in _
verify
    raise MultipleInvalid(iface, candidate, excs)
zope.interface.exceptions.MultipleInvalid: The object Point(x=100,
y=200) has failed to implement interface <InterfaceClass __main__.
ICollidable>:
    Does not declaratively implement the interface
    The __main__.ICollidable.bounding_box attribute was not provided
```

Две последние строки сообщают о двух ошибках:

- **Ошибка объявления**: у проблемного объекта нет явного объявления о том, что он реализует интерфейс.
- **Структурная ошибка**: у проблемного объекта не хватает компонентов, которых требует интерфейс.

Вторая ошибка защищает от неполных интерфейсов. Если бы у класса `Point` был декоратор `@implementer(ICollidable)`, но не было свойства `bounding_box()`, то это исключение все равно возникло бы.

Методы `verifyClass()` и `verifyObject()` проверяют интерфейс только на верхнем уровне и не отслеживают типы атрибутов. При желании можно выполнить более глубокую проверку методом `validateInvariants()`, который предоставляется всеми классами интерфейсов пакета `zope.interface` и позволяет функциям-перехватчикам анализировать значения интерфейсов. Таким образом, если вам особенно важна надежность кода, используйте следующий паттерн для интерфейсов и их проверки:

```
from zope.interface import Interface, Attribute, invariant
from zope.interface.verify import verifyObject

class IBBox(Interface):
    x1 = Attribute("нижняя левая координата x")
    y1 = Attribute("нижняя левая координата y")
    x2 = Attribute("верхняя правая координата x")
    y2 = Attribute("верхняя правая координата y")

class ICollidable(Interface):
    bounding_box = Attribute("Ограничительный прямоугольник объекта")
    invariant(lambda self: verifyObject(IBBox, self.bounding_box))

def find_collisions(objects):
    for item in objects:
        verifyObject(ICollidable, item)
        ICollidable.validateInvariants(item)

    ...
```

С помощью метода `validateInvariants()` можно не только убедиться, что входные данные содержат все атрибуты, необходимые для интерфейса `ICollidable`, но и проверить, удовлетворяет ли структура этих атрибутов (в данном случае `bounding_box`) более глубоким ограничениям. В этом примере `invariant()` используется для проверки вложенного интерфейса.

Пакет `zope.interface` примечателен тем, что позволяет ослабить связанность (coupling) в приложении. С его помощью можно добиться, чтобы объекты поддерживали нужные интерфейсы без чрезмерных сложностей, связанных с мно-

жественным наследованием, а также обнаруживать несоответствия на ранней стадии.

Самый большой недостаток `zope.interface` состоит в том, что требуется явно объявлять, какие объекты реализуют интерфейсы. Это особенно неудобно, когда нужно проверять экземпляры, происходящие из внешних классов встроенных библиотек. В библиотеке есть ряд средств для решения этой проблемы, хотя с ними код в конце концов становится излишне громоздким. Конечно, такие проблемы можно решить самостоятельно с помощью паттерна Адаптер и даже горячей подменой (monkey-patching) во внешних классах, однако элегантность таких решений в лучшем случае находится под вопросом.

Аннотоции функций и абстрактные базовые клоссы

Формальные интерфейсы предназначены для того, чтобы обеспечивать слабую связанность (loose coupling) в больших приложениях, а не для того, чтобы создавать новые уровни сложности. `zope.interface` — отличпая концепция, которая прекраспо подходит для пекоторых проектов, и все же это не папацея. С `zope.interface` вы рискуете обнаружить, что бóльшую часть времени решаете проблемы с несовместимыми интерфейсами для сторонних классов и конструируете бесконечные прослойки адаптеров, вместо того чтобы писать настоящие реализации.

Если вы оказались в такой ситуации, значит, что-то пошло не так. К счастью, Python поддерживает не только явные интерфейсы, но и упрощенный альтернативный механизм. Это не полноценное решение, как `zope.interface` или его аналоги, но опо обычно придает приложениям больше гибкости. Возможно, придется нанисать чуть больше кода, но в конечном итоге вы получите результат, который лучше расширяется, лучше справляется с внешними типами и, возможно, обеспечит лучший задел на будущее.

Следует учитывать, что в Python нет и, вероятно, никогда не будет явно выраженного попятия интерфейса как такового. Одпако в языке есть возможности, которые позволяют реализовать некое подобие функциональности иптерфейсов:

- **абстрактные базовые классы (ABC)**;
- **аннотации функцнй**;
- **аннотации тннов**.

В основу нашего решения заложены абстрактные базовые классы, поэтому пачнем с них.

Вероятно, вам уже известно, что напрямую сравнивать типы считается делом вредным и непитоническим. Всегда следует избегать сравнений вида:

```
assert type(instance) == list
```

Такое сравнение типов в методах или функциях полностью нарушает возможность передавать подтип класса в качестве аргумента функции. Чуть более эффективное решение использует функцию `isinstance()`, которая учитывает возможность наследования:

```
assert isinstance(instance, list)
```

Дополнительное преимущество функции `isinstance()` заключается в том, что она может проверять совместимость с широким диапазоном типов. Например, если ваша функция ожидает получить в аргументе какую-либо последовательность, можно сопоставить экземпляр со списком базовых типов:

```
assert isinstance(instance, (list, tuple, range))
```

Такая проверка совместимости типов эффективна в некоторых ситуациях, но и она не идеальна. Она работает с любыми подклассами `list`, `tuple` и `range`, но дает сбой, если пользователь передаст объект, который ведет себя точно так же, как один из типов последовательностей, но не наследует ни одному из них. Допустим, например, что мы ослабим свои требования и позволим принимать в аргументе любую разновидность итерируемого объекта. Что тогда делать?

Список базовых типов, которые относятся к категории итерируемых, весьма обширен. В него входят `list`, `tuple`, `range`, `str`, `bytes`, `dict`, `set` и многие другие тины. Набор подходящих встроенных типов получается объемным, и даже если вы перечислите их все, это все равно не обеспечит проверку для нестандартных классов, которые определяют метод `__iter__()`, но наследуют непосредственно классу `object`.

В таких ситуациях подходящим решением становятся абстрактные базовые классы (ABC). Класс такого рода не обязан предоставлять конкретную реализацию, а просто определяет схему класса, которую можно использовать при проверке совместимости тинов. Эта концепция очень похожа на абстрактные классы и виртуальные методы, которые применяются в языке C++.

Абстрактные базовые классы используются для двух целей:

- проверки полноты реализации;
- проверки совместимости неявных интерфейсов.

Использовать ABC несложно. Сначала вы определяете новый класс, который либо наследует базовому классу `abc.ABC`, либо имеет в качестве метакласса

`abc.ABCMeta`. Разговор о метаклассах мы отложим до главы 8, а здесь рассмотрим только классическое наследование.

Вот пример абстрактного базового класса, который определяет интерфейс, не делающий ничего особо полезного:

```
from abc import ABC, abstractmethod

class DummyInterface(ABC):

    @abstractmethod
    def dummy_method(self): ...

    @property
    @abstractmethod
    def dummy_property(self): ...
```

Декоратор `@abstractmethod` обозначает часть интерфейса, которая должна быть реализована (путем переопределения) в подклассах нашего ABC. Если у класса будут методы или свойства, которые не переопределены, то создать его экземпляр не удастся: возникнет исключение `TypeError`.

Такой подход отлично обеспечивает полноту реализации и по явному выражению не уступает `zope.interface`. Чтобы в примере из предыдущего раздела вместо `zope.interface` использовать абстрактные базовые классы, можно изменить определения классов следующим образом:

```
from abc import ABC, abstractmethod
from dataclasses import dataclass

class ColliderABC(ABC):
    @property
    @abstractmethod
    def bounding_box(self): ...

@dataclass
class Square(ColliderABC):
    ...

@dataclass
class Rect(ColliderABC):
    ...

@dataclass
class Circle(ColliderABC):
    ...
```

Тела и свойства классов `Square`, `Rect` и `Circle` не меняются, потому что суть интерфейса остается прежней. Изменился только способ объявления явного

интерфейса: теперь вместо декоратора класса `zope.interface.implementer()` используется наследование. Если вы все же хотите проверить, что входные данные `find_collisions()` соответствуют интерфейсу, необходимо использовать функцию `isinstance()`. Изменение будет весьма простым:

```
def find_collisions(objects):
    for item in objects:
        if not isinstance(item, ColliderABC):
            raise TypeError(f"{item} is not a collider")
    ...
```

Нам приходится использовать подклассы, поэтому связанность между компонентами слегка усугубляется, но не сильно отличается от решения с `zope.interface`. До тех пор пока мы опираемся на интерфейсы (то есть `ColliderABC`), а не на конкретные реализации (`Square`, `Rect` или `Circle`), связанность все равно считается слабой.

Однако схема может быть еще более гибкой. Python предоставляет в ваше распоряжение всю мощь интроспекции. Благодаря утиной типизации в Python любой объект, который «крякает как утка», можно использовать в качестве утки. К сожалению, обычно это делается в духе «попробуем и посмотрим, что получится». Предполагается, что объект в заданном контексте соответствует ожидаемому интерфейсу. А ведь смысл формальных интерфейсов именно в том, чтобы формулировать контракт, соответствие которому можно проверить. Можно ли проверить, соответствует ли объект интерфейсу, не попытавшись предварительно его использовать?

В некоторой степени — да. Абстрактные базовые классы предоставляют специальный метод `__subclasshook__(cls)`. Он позволяет внедрить собственную логику в процедуру, которая определяет, является ли объект экземпляром заданного класса. К сожалению, всю логику вам придется обеспечить самостоятельно, потому что создатели `abc` решили не ограничивать разработчиков, которые захотят переопределить механизм `isinstance()` целиком. Весь процесс оказывается под вашим полным контролем, но за это вам приходится писать шаблонный код.

И хотя в методе `__subclasshook__()` можно делать все что угодно, обычно единственный разумный вариант — следовать стандартному шаблону. Чтобы узнать, совместим ли тот или иной класс (неявно) с тем или иным абстрактным базовым классом, нужно выяснить, содержит ли он все методы абстрактного базового класса.

Стандартная процедура состоит в том, чтобы проверить, доступен ли набор определенных методов где-то в порядке разрешения методов (MRO) заданного класса. Если понадобится расширить интерфейс `ColliderABC` из нашего примера методом `__subclasshook__`, это можно сделать так:

```
class ColliderABC(ABC):
    @property
    @abstractmethod
    def bounding_box(self): ...

    @classmethod
    def __subclasshook__(cls, C):
        if cls is ColliderABC:
            if any("bounding_box" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

При таком онределении метода `__subclasshook__()` класс `ColliderABC` становится неявным интерфейсом. Это означает, что экземпляром `ColliderABC` будет считаться любой объект, чья структура проходит проверку `__subclasshook__()`. Благодаря этому можно добавлять новые компоненты, совместимые с интерфейсом `ColliderABC`, без явного наследования ему. Вот пример класса `Line` (отрезок линии), который будет считаться действительным подклассом `ColliderABC`:

```
@dataclass
class Line:
    p1: Point
    p2: Point

    @property
    def bounding_box(self):
        return Box(
            self.p1.x,
            self.p1.y,
            self.p2.x,
            self.p2.y,
        )
```

Как видите, нигде в коде класса данных `Line` не уноминается `ColliderABC`. Тем не менее экземпляры `Line` можно проверять на совместимость неявных интерфейсов, сравнивая их с `ColliderABC` с помощью функции `isinstance()`:

```
>>> line = Line(Point(0, 0), Point(100, 100))
>>> line.bounding_box
Box(x1=0, y1=0, x2=100, y2=100)
>>> isinstance(line, ColliderABC)
True
```

До сих пор мы имели дело со свойствами, но такой же подход можно использовать и для методов. К сожалению, в этом случае при проверке совместимости типов и полноты реализации не будут учитываться сигнатуры методов классов. Таким образом, даже если количество ожидаемых аргументов в реализации отличается, она все равно будет считаться совместимой. В большинстве слу-

чаев это не создает проблем, но если требуется именно детализированный контроль над интерфейсами, то вам поможет пакет `zope.interface`. Как уже упоминалось, метод `__subclasshook__()` позволяет добавить сколь угодно сложную логику в функцию `isinstance()`, чтобы добиться похожего уровня контроля.

collections.abc

Абстрактные базовые классы можно сравнить с маленькими кирпичиками, из которых строится более высокий уровень абстракции. Они позволяют реализовать по-настоящему полезные интерфейсы, однако по своей природе они универсальны, а их возможности гораздо шире, чем поддержка одного этого паттерна проектирования. ABC позволяют вам проявлять фантазию и создавать волшебные решения, однако на то, чтобы разработать нечто обобщенное и при этом практически ценное, могут понадобиться значительные трудозатраты, которые никогда не окупятся. Стандартная библиотека и встроенные типы Python в полной мере раскрывают потенциал абстрактных базовых классов.

Модуль `collections.abc` предоставляет множество заранее определенных ABC, которые позволяют проверять совместимость типов со стандартными интерфейсами Python. Например, с помощью базовых классов из этого модуля можно проверить, является ли заданный объект вызываемым, поддерживает ли он итерацию или является отображением. Использовать эти классы с функцией `isinstance()` намного удобнее, чем выполнять сравнение с базовыми типами Python. Стоит уметь пользоваться этими базовыми классами, даже если вы не определяете собственные интерфейсы с помощью `abc.ABC`.

Перечислим самые полезные абстрактные базовые классы из `collections.abc`, которые вы будете использовать достаточно часто:

- `Container`: этот интерфейс означает, что объект поддерживает оператор `in` и реализует метод `__contains__()`.
- `Iterable`: этот интерфейс означает, что объект поддерживает итерацию и реализует метод `__iter__()`.
- `Callable`: этот интерфейс означает, что объект может вызываться как функция и реализует метод `__call__()`.
- `Hashable`: этот интерфейс означает, что объект является хешируемым (то есть может быть элементом множества или ключом словаря) и реализует метод `__hash__`.
- `Sized`: этот интерфейс означает, что объект обладает размером (то есть может быть аргументом функции `len()`) и реализует метод `__len__()`.

Полный список абстрактных базовых классов из модуля `collections.abc` доступен в официальной документации Python по адресу https://docs.python.org/3/library/collections.abc.html.

Модуль `collections.abc` наглядно демонстрирует, для чего ABC годятся лучше всего: чтобы создавать коптракты небольших и простых протоколов объектов. Они вряд ли хорошо подойдут, чтобы сопровождать изощренную структуру большого интерфейса. У абстрактных базовых классов также нет механизмов, которые позволяли бы легко проверять атрибуты или глубоко анализировать аргументы фупкций и возвращаемые типы.

К счастью, для этих задач существует совершеппо другое решение: статический анализ типов и тип `typing.Protocol`.

Интерфейсы с аннотациями типов

Аннотации типов в Python оказались чрезвычайно удобным способом повысить качество программных продуктов. Все больше и больше профессиопальных разработчиков используют `mypy` или другие средства статического апализа типов по умолчанию, а традиционное нрограммирование без типов примепяют разве что для прототинов и коротких одноразовых сценариев.

За носледнне годы значительно развилась поддержка типизации в стандартной библиотеке и проектах сообщества. Благодаря этому гибкость аннотаций типов улучшается с каждым новым выпуском Python. Кроме того, появляется возможность использовать аннотации типов в совершенно новых контекстах.

Один из таких коптекстов — структурная подтипизация (она же статическая утипая типизация). Это всего лишь еще одна вариация на тему пеявных интерфейсов, которая также предоставляет минимальные наивные возможности проверки на стадии выполнения в духе `__subclasshook__()`.

Структурная подтипизация оспована на типе `typing.Protocol`. Объявляя его подклассы, можно создать определение иптерфейса. Ниже приведен пример базовых интерфейсов `Protocol`, которые можно было бы использовать в предыдущих примерах с обнаружением перекрытий:

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class IBox(Protocol):
    x1: float
    y1: float
    x2: float
```

```
    y2: float

@runtime_checkable
class ICollider(Protocol):
    @property
    def bounding_box(self) -> IBox: ...
```

На этот раз используются два интерфейса. Такие инструменты, как `mypy`, поддерживают глубокую проверку типов, так что можно задействовать дополнительные интерфейсы, чтобы повысить безопасность типов. Декоратор `@runtime_checkable` расширяет класс протокола проверками `isinstance()`. Для ABC с перехватчиками подклассов из предыдущего раздела нам приходилось реализовывать соответствующую функциональность вручную, а здесь она получается практически сама собой.

Средства статического анализа типов подробнее рассматриваются в главе 10 «Автоматизация тестирования и контроля качества».

Чтобы в полной мере воспользоваться преимуществами статического анализа типов, нужно также разметить остальной код соответствующими аннотациями. Ниже приведен полный код обнаружения перекрытий, где проверка интерфейсов на стадии выполнения основана на классах протоколов:

```
import itertools
from dataclasses import dataclass
from typing import Iterable, Protocol, runtime_checkable

@runtime_checkable
class IBox(Protocol):
    x1: float
    y1: float
    x2: float
    y2: float

@runtime_checkable
class ICollider(Protocol):
    @property
    def bounding_box(self) -> IBox: ...

def rects_collide(rect1: IBox, rect2: IBox):
    """Проверяет перекрытие прямоугольников

    Координаты прямоугольников:
        ┌──(x2, y2)
        │       │
      (x1, y1)──┘
    """
    return (
```

```
            rect1.x1 < rect2.x2 and
            rect1.x2 > rect2.x1 and
            rect1.y1 < rect2.y2 and
            rect1.y2 > rect2.y1
        )

    def find_collisions(objects: Iterable[ICollider]):
        for item in objects:
            if not isinstance(item, ICollider):
                raise TypeError(f"{item} is not a collider")

        return [
            (item1, item2)
            for item1, item2
            in itertools.combinations(objects, 2)
            if rects_collide(
                item1.bounding_box,
                item2.bounding_box
            )
        ]
```

Код классов `Rect`, `Square` и `Circle` не приводится, потому что их реализация не требует изменений. И в этом нроявляется настоящая элегантность неявных интерфейсов: в каждом конкретном классе нет никаких явных объявлений интерфейсов, а есть только неявный интерфейс, обусловленный фактической реализацией.

В результате можно использовать все нредыдущие версии классов `Rect`, `Square` и `Circle` (простые классы данных, классы с объявлениями `zope` или потомки ABC). Все они будут совместимы со структурной подтипизацией, реализованной через класс `typing.Protocol`.

Как видите, несмотря на то что в Python нет встроенной поддержки интерфейсов (на том уровне, на каком она присутствует, скажем, в Java или Go), существует множество способов стандартизировать контракты классов, методов и функций. Эта способность становится особенно полезной, когда требуется с помощью разных паттернов проектирования решать часто встречающиеся задачи разработки. Главное назначение паттернов проектирования — обеспечивать повторное использование кода, а интерфейсы помогают преобразовывать паттерны в шаблоны кода, которые можпо применять снова и снова.

Однако область применения интерфейсов (и аналогичных решений) не ограничивается паттернами. Создание четко определенных и проверяемых контрактов для программной единицы (функции, класса или метода) стало важнейшим элементом некоторых парадигм и приемов программирования. Типичные примеры — инверсия управления и внедрение зависимостей. Эти две концепции тесно связаны, поэтому в следующем разделе мы рассмотрим их одновременно.

Инверсия управления и внедрение зависимостей

Инверсия управления (IoC, Inversion of Control) — простое свойство некоторых программных архитектур. Согласно Wiktionary, если архитектура поддерживает IoC, это означает, что:

> *(...) поток управления в системе инвертируется по сравнению с традиционной архитектурой.*

Но что считать традиционной архитектурой? Идея IoC не нова, ее история прослеживается как минимум до статьи Дэвида Д. Кларка (David D. Clark) «The structuring of systems using of upcall» («Структурирование систем с использованием восходящих вызовов»), опубликованной в 1985 году. Вероятно, под традиционной архитектурой понимается архитектура ПО, которая была распространенной или считалась стандартной в 1980-х годах.

Полный текст статьи Кларка доступен по адресу https://groups.csail.mit.edu/ana/Publications/PubPDFs/The%20Structuring%20of%20Systems%20Using%20Upcalls.pdf.

Кларк описывает традиционную архитектуру программы как многоуровневую структуру процедур, в которой управление всегда направлено сверху вниз. Более высокие уровни вызывают процедуры с более низких уровней.

Вызванные процедуры получают управление и могут вызывать процедуры еще более низких уровней, прежде чем возвращать управление наверх. На практике управление традиционно передается от приложения библиотечным функциям. Библиотечные функции могут нередавать управление библиотекам еще более низкого уровня, но в конечном счете оно возвращается приложению.

Инверсия управления происходит, когда библиотека передает управление вверх приложению, чтобы оно могло повлиять на работу библиотеки. Чтобы лучше понять эту концепцию, рассмотрим элементарный пример сортировки списка целых чисел:

```
sorted([1,2,3,4,5,6])
```

Встроенная функция `sorted()` получает итерируемый объект и возвращает отсортированный список элементов. Управление переходит от вызывающей стороны (ваше приложение) непосредственно к функции `sorted()`. Завершив сортировку, она просто возвращает отсортированный результат и передает управление обратно вызывающей стороне. Здесь нет никаких сложностей.

Теперь допустим, что мы хотим отсортировать числа по необычному принципу — скажем, по абсолютному расстоянию до числа 3. Чем ближе число к 3, тем раньше оно будет следовать в списке результатов, а чем дальше — тем позже. Чтобы реализовать такую сортировку, можно определить простую функцию, которая будет определять порядок элемептов:

```
def distance_from_3(item):
    return abs(item - 3)
```

Тенерь эту функцию можно передать в аргументе `key` функции `sorted()`:

```
sorted([1,2,3,4,5,6], key=distance_from_3)
```

В результате функция `sorted()` будет вызывать фупкцию `key` для каждого элемента итерируемого аргумента. Вместо того чтобы сравнивать значения элементов, она теперь сравнивает возвращаемые значения функции `key`. Здесь и происходит ипверсия управления (IoC). Функция `sorted()` выполняет «восходящий вызов» фупкции `distance_from_3()`, которую приложение передало в аргументе. Получается, что библиотека вызывает функции приложения, а не наоборот; таким образом, поток управления идет в обратпом направлепии.

Инверсия управления на базе обратных вызовов также шуточно называется «принципом Голливуда». Имеется в виду знаменитая фраза «Не звоните нам, мы сами вам позвоним».

Следует учитывать, что сама по себе инверсия управления — характеристика архитектуры, а не паттерн проектирования. Пример с функцией `sorted()` — простейшая разновидность IoC на базе обратных вызовов, но инверсия управлепия может нринимать много других форм, например:

- **Полиморфизм**: производпый класс наследует базовому классу, а методы базового класса вызывают методы производного.
- **Передача аргументов**: функция, припявшая объект в качестве аргумента, вызывает методы этого объекта.
- **Декораторы**: функция-декоратор вызывает декорируемую функцию.
- **Замыкання**: вложенная фупкция вызывает фупкцию за пределами своей области видимости.

Нетрудно видеть, что инверсия управления — довольно распространенный аспект объектно-ориентированной или фупкциональной нарадигмы программирования. Она часто встречается даже там, где вы ее не замечаете. Хотя сама по себе IoC не является паттерном проектирования, она служит ключевым ингредиентом многих настоящих паттернов, парадигм и методологий. Среди пих наиболее заметно внедрение зависимостей, которое будет рассмотрено позднее в этой главе.

Традиционный поток управления в стиле процедурного программирования по Кларку встречается и в ООП. В объектно-ориентированных программах сами объекты становятся получателями управления. Можно сказать, что управление передается объекту при каждом вызове его метода. Таким образом, традиционный поток управления требовал бы, чтобы объект полностью владел всеми зависимыми объектами, которые необходимы для реализации его поведения.

Инверсия управления в приложениях

Чтобы лучше продемонстрировать различия между разными потоками управления, мы создадим небольшое, но вполне рабочее приложение. Изначально в нем будет реализован традиционный поток управления, но затем мы покажем, как улучшить приложение, если в отдельных местах применить инверсию управления.

Наш сценарий использования будет довольно простым и распространенным. Мы построим службу, которая отслеживает просмотры веб-страниц с использованием так называемых **пикселей отслеживания** (tracking pixels) и предоставляет статистику просмотра страниц через конечную точку HTTP. Этот прием широко используется, чтобы подсчитывать просмотры рекламы или открытия сообщений электронной почты. Также он полезен, если вы активно используете кэширование HTTP и хотите убедиться, что оно не влияет на статистику просмотра страниц.

Нашему приложению понадобится хранить счетчик просмотров страницы в каком-то постоянном хранилище. Заодно это позволит нам исследовать модульность приложения — характеристику, которую нельзя реализовать без IoC.

У приложения будут две конечные точки:

- `/track`: эта конечная точка будет возвращать ответ HTTP с GIF-изображением размером 1×1 пиксель. При поступлении запроса она будет сохранять заголовок `Referer` и увеличивать счетчик запросов, связанных с этим значением.
- `/stats`: эта конечная точка будет обнаруживать десять самых частых значений `Referer`, полученных через конечную точку `/track`, и возвращать ответ HTTP со сводкой результатов в формате JSON.

`Referer` — необязательный заголовок HTTP, с помощью которого браузеры передают веб-серверу URL-адрес исходной веб-страницы, с которой был запрошен ресурс. Этот заголовок был впервые стандартизирован в документе RFC 1945 «Протокол передачи гипертекста — HTTP/1.0» (см. https://tools.ietf.org/html/rfc1945). Обратите внимание на неправильное написание слова «referrer»: когда ошибку обнаружили, исправлять ее было уже поздно.

В главе 2 «Современные среды разработки для Python» уже был представлен простой веб-микрофреймворк Flask, который мы снова применим здесь. Для начала импортируем модули и настроим перемеппые, которые будут использоваться в программе:

```
from collections import Counter
from http import HTTPStatus

from flask import Flask, request, Response

app = Flask(__name__)
storage = Counter()

PIXEL = (
    b'GIF89a\x01\x00\x01\x00\x80\x00\x00\x00'
    b'\x00\x00\xff\xff\xff!\xf9\x04\x01\x00'
    b'\x00\x00\x00,\x00\x00\x00\x00\x01\x00'
    b'\x01\x00\x00\x02\x01D\x00;'
)
```

Переменная `app` — основной объект фреймворка Flask, представляющий веб-приложение Flask. Позже мы используем эту переменную, чтобы регистрировать маршруты конечных точек и запускать серверы разработки приложения.

В переменной `storage` хранится экземпляр `Counter`. Это удобная структура данных из стандартной библиотеки, которая позволяет хранить счетчики любых неизменяемых значений. Наша конечная цель — размещать статистику просмотров страницы в постоянном хранилище, но проще будет начать с чего-то менее амбициозного. Именпо поэтому в первоначальной версии приложения эта переменная будет хранить статистику просмотра страпицы в памяти.

Остается сказать о переменной `PIXEL`. Она содержит байтовое представление прозрачного изображения в формате GIF размером 1 × 1 пиксель. Реальный внешний вид пикселя отслеживания ни на что пе влияет и, скорее всего, никогда не будет измепяться. Также размер этих данных настолько мал, что нет смысла возиться с их загрузкой из файловой системы. Поэтому мы встраиваем данпые в модуль, чтобы все приложение помещалось в одном модуле Python.

Закончив подготовку, можно написать код обработчика конечной точки `/track`:

```
@app.route('/track')
def track():
    try:
        referer = request.headers["Referer"]
    except KeyError:
        return Response(status=HTTPStatus.BAD_REQUEST)

    storage[referer] += 1
```

```
    return Response(
        PIXEL, headers={
            "Content-Type": "image/gif",
            "Expires": "Mon, 01 Jan 1990 00:00:00 GMT",
            "Cache-Control": "no-cache, no-store, must-revalidate",
            "Pragma": "no-cache",
        }
    )
```

Заголовки `Expires`, `Cache-Control` и `Pragma` управляют механизмом кэширования HTTP. В этом примере они задаются так, чтобы отключить все виды кэширования в большинстве современных браузеров. Кроме того, кэширование отключается и на любых промежуточных узлах. Обратите особое внимание на то, что значение заголовка Expires устанавливается в прошлом. Это наименьшее допустимое время эпохи; на практике оно означает, что срок действия ресурса всегда считается истекшим.

Обработчики запросов Flask обычно начинаются с декоратора `@app.route(route)`, который регистрирует последующую функцию-обработчик для заданного маршрута HTTP. Обработчики запросов также называются **нредставленнямн** (views). В данном случае мы регистрируем представление `track()` в качестве обработчика конечной точки маршрута `/track`. Здесь в нашем приложении впервые встречается инверсия управления: мы регистрируем свою собственную реализацию обработчика во фреймворке Flask. Этот фреймворк будет обратно вызывать наши обработчики для входящих запросов, которые соответствуют связанным с ними маршрутам.

После сигнатуры идет простой код обработки запроса. Мы проверяем, есть ли во входящем запросе заголовок `Referer`. По его значению браузер определяет URI «родительского» ресурса (например, отслеживаемой HTML-страницы), в который был включен запрошенный ресурс (в нашем случае GIF-изображение). Если такого заголовка не существует, возвращается ошибка сервера с кодом состояния `400 Bad Request` («Некорректный запрос»).

Если во входящем запросе есть заголовок `Referer`, то значение счетчика в переменной `storage` будет увеличено. Структура `Counter` предоставляет интерфейс, подобный `dict`, и позволяет легко модифицировать значения счетчика для ключей, которые еще не зарегистрированы. В таких случаях предполагается, что исходное значение для заданного ключа равно 0. При этом подходе не нужно проверять, встречалось ли ранее конкретное значение `Referer`, и это ощутимо упрощает код. Увеличив значение счетчика, мы возвращаем ответ — изображение из одного пикселя, которое наконец-то можно вывести в браузере.

Хотя переменная `storage` определена за пределами функции `track()`, ее еще нельзя считать примером IoC. Дело в том, что сторона, вызывающая функцию

stats(), не может подменить реализацию storage. Мы попробуем изменить эту ситуацию в следующей версии приложения.

Код конечной точки /stats выглядит еще проще:

```
@app.route('/stats')
def stats():
    return dict(storage.most_common(10))
```

В представлении stats() мы снова пользуемся преимуществами удобного интерфейса объекта Counter. В нем есть метод most_common(n), который возвращает до n наиболее часто встречающихся пар «ключ — значение», хранящихся в структуре. Наша программа немедленно преобразует результат в словарь. Мы не используем класс Response, потому что без него Flask по умолчанию сериализует возвращаемые значения в JSON и предполагает статус 200 OK для ответа HTTP.

Чтобы протестировать приложение, мы завершим сценарий простым вызовом встроенного сервера разработки:

```
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8000)
```

Если сохранить приложение в файле tracking.py, сервер можно будет запустить командой python tracking.py и приложение начнет прослушивать порт 8000. Чтобы протестировать приложение в браузере, дополните его следующим обработчиком конечной точки:

```
@app.route('/test')
def test():
    return """
    <html>
    <head></head>
    <body><img src="/track"></body>
    </html>
    """
```

Если вы несколько раз откроете адрес http://localhost:8000/test в своем браузере, а затем перейдете по адресу http://localhost:8000/stats, результат будет выглядеть примерно так:

```
{"http://localhost:8000/test":6}
```

Недостаток текущей реализации в том, что счетчики запросов хранятся в памяти. При каждом перезапуске приложения счетчики сбрасываются, и теряются важные данные. Чтобы они сохранялись между перезапусками, storage нужно реализовать по-другому.

Существует много вариантов постоянного хранения данных, например:

- простой текстовый файл;
- встроенный модуль `shelve`;
- система управления реляционными базами данных — например, MySQL, MariaDB или PostgreSQL;
- служба хранения пар «ключ — значение» или структур данных в памяти, например Memcached или Redis.

Выбор хранилища зависит от контекста и объема рабочей нагрузки, которую должно выдерживать приложение. Если еще неизвестно, какое решение будет оптимальным, механизм хранения данных можно сделать переключаемым, чтобы переходить с одной подсистемы хранения на другую в зависимости от реальных потребностей пользователей. Для этого понадобится инвертировать поток управления в функциях `track()` и `stats()`.

Принципы качественного проектирования требуют подготовить в том или ином виде определение интерфейса объекта, который будет отвечать за IoC. По-видимому, удобный в использовании интерфейс класса `Counter` может послужить хорошей отправной точкой. Единственная проблема заключается в том, что операцию `+=` можно реализовать как специальным методом `__add__()`, так и методом `__iadd__()`, а нам такая неоднозначность определенно ни к чему. Кроме того, у класса `Counter` слишком много дополнительных методов, в то время как нам нужны только два:

- метод, который увеличивает значение счетчика на 1;
- метод, который предоставляет 10 наиболее часто запрашиваемых ключей.

Чтобы код оставался простым и хорошо читался, мы определим собственный интерфейс хранения количества просмотров в виде абстрактного базового класса:

```
from abc import ABC, abstractmethod
from typing import Dict

class ViewsStorageBackend(ABC):
    @abstractmethod
    def increment(self, key: str): ...

    @abstractmethod
    def most_common(self, n: int): Dict[str, int] ...
```

С этого момента можно создавать различные реализации подсистемы хранения количества просмотров. Следующая реализация адаптирует класс `Counter`, который использовался раньше, к интерфейсу `ViewsStorageBackend`:

```
from collections import Counter
from typing import Dict

from .tracking_abc import ViewsStorageBackend

class CounterBackend(ViewsStorageBackend):
    def __init__(self):
        self._counter = Counter()

    def increment(self, key: str):
        self._counter[key] += 1

    def most_common(self, n: int) -> Dict[str, int]:
        return dict(self._counter.most_common(n))
```

А если вы хотите обеспечить постоянное хранилище на основе службы хранения в памяти Redis, для этого можно реализовать другую подсистему хранения:

```
from typing import Dict
from redis import Redis

class RedisBackend(ViewsStorageBackend):
    def __init__(
        self,
        redis_client: Redis,
        set_name: str
    ):
        self._client = redis_client
        self._set_name = set_name

    def increment(self, key: str):
        self._client.zincrby(self._set_name, 1, key)

    def most_common(self, n: int) -> Dict[str, int]:
        return {
            key.decode(): int(value)
            for key, value in
            self._client.zrange(
                self._set_name, 0, n-1,
                desc=True,
                withscores=True,
            )
        }
```

Redis — это хранилище данных в памяти. Это означает, что по умолчанию данные размещены только в памяти. При перезагрузке Redis сохраняет их на диск, но может потерять при неожиданных сбоях (например, из-за аварийного отключения питания). Тем не менее это всего лишь поведение по умолчанию. Redis предлагает различные режимы хранения, в том числе сравнимые с другими базами данных. В итоге Redis вполне подходит для простого сценария из нашего примера. Дополнительная информация о том, как устроено постоянное хранение данных в Redis, доступна по адресу https://redis.io/topics/persistence.

Интерфейсы обеих подсистем одинаковы, и абстрактный базовый класс обеспечивает неформальное соблюдение этих интерфейсов. Это означает, что экземпляры обоих классов можно использовать взаимозаменяемо. Но остается вопрос: как инвертировать управление функций `track()` и `stats()`, чтобы можно было подключать разпые реализации хранилища количества просмотров?

Вспомните сигнатуры наших функций:

```
@app.route('/stats')
def stats():
   ...
@app.route('/track')
def track():
   ...
```

Во фреймворке Flask декоратор `app.route()` регистрирует функцию как обработчик конкретного маршрута. Ее можно рассматривать как функцию обратного вызова для путей занросов HTTP. Вы больше не вызываете эту функцию вручную, а Flask полностью управляет передаваемыми ей аргументами. Однако нам нужна возможность легко заменить реализацию хранилища данных. Один из способов добиться этого — откладывать регистрацию обработчика и передавать функциям дополнительный аргумент `storage`. Рассмотрим следующий пример:

```
def track(storage: ViewsStorageBackend):
    try:
        referer = request.headers["Referer"]
    except KeyError:
        return Response(status=HTTPStatus.BAD_REQUEST)

    storage.increment(referer)

    return Response(
        PIXEL, headers={
            "Content-Type": "image/gif",
            "Expires": "Mon, 01 Jan 1990 00:00:00 GMT",
            "Cache-Control": "no-cache, no-store, must-revalidate",
            "Pragma": "no-cache",
        }
    )

def stats(storage: ViewsStorageBackend):
    return storage.most_common(10)
```

Наш дополнительпый аргумент помечен аннотацией типа `ViewsStorageBackend`, чтобы тип можно было легко проверить средствами IDE или с помощью дополнительных инструментов. Благодаря этому удалось инвертировать управ-

ление для этих функций, а также улучшить модульность. Теперь реализацию хранилища данных можно легко заменить другим классом с совместимым интерфейсом. Дополнительное преимущество IoC заключается в том, что можно легко провести модульное тестирование методов `stats()` и `track()` независимо от реализации хранилища данных.

Тема модульного тестирования рассматривается в главе 10 «Автоматизация тестирования и контроля качества» вместе с подробными примерами тестов, использующих инверсию управления.

Единственное, чего здесь не хватает — регистрации реального маршрута. Декоратор `app.route()` больше не получится использовать напрямую с функциями, потому что Flask не сможет обработать аргумент `storage` самостоятельно. Чтобы решить проблему, можно предварительно внедрить нужные реализации хранилища в функции-обработчики и создать новые функции, которые легко регистрируются вызовом `app.route()`.

Для этого проще всего воспользоваться функцией `partial()` из модуля `functools`. Она принимает одну функцию с набором позиционных и именованных аргументов, а возвращает новую функцию, где отдельные аргументы заранее зафиксированы. Этот прием можно использовать, чтобы подготовить различные конфигурации нашей службы. Например, так настраивается конфигурация приложения, которая использует Redis в качестве подсистемы хранилища данных:

```
from functools import partial

if __name__ == '__main__':
    views_storage = RedisBackend(Redis(host="redis"), "my-stats")

    app.route("/track", endpoint="track")(
        partial(track, storage=views_storage))
    app.route("/stats", endpoint="stats")(
        partial(stats, storage=views_storage))

    app.run(host="0.0.0.0", port=8000)
```

Этот подход можно применять со многими другими веб-фреймворками, потому что в большинстве из них используется такая же структура привязки маршрутов к обработчикам. Эта схема особенно хорошо подходит для небольших служб с малым числом конечных точек. К сожалению, в крупных приложениях она не всегда эффективно масштабируется. Соответствующий код легко написать, но ощутимо труднее читать. Опытные разработчики Flask наверняка сочтут этот подход неестественным и излишне однообразным. В данном случае он просто

нарушает стандартные соглашения о том, как писать функции-обработчики Flask.

Окончательным решением будет то, которое позволит писать и регистрировать функции представлений без необходимости вручную внедрять зависимые объекты, например так:

```
@app.route('/track')
def track(storage: ViewsStorageBackend):
    ...
```

Чтобы достичь этой цели, от фреймворка Flask понадобятся следующие возможности:

- Распознавать дополнительные аргументы как зависимости представлений.
- Позволять определять реализацию по умолчанию для этих зависимостей.
- Автоматически разрешать зависимости и внедрять их в представления во время выполнения.

Такой механизм называется **внедрением зависимостей**, и мы уже упоминали о нем. В некоторых веб-фреймворках есть встроенные средства внедрения зависимостей, однако для экосистемы Python это скорее редкость. К счастью, существует множество облегченных библиотек внедрения зависимостей, которые можно добавить поверх любого фреймворка Python. Такая функциональность будет рассмотрена в следующем разделе.

Фреймворки внедрения зависимостей

Если использовать инверсию управления в больших масштабах, она быстро выходит из-под контроля. Пример из предыдущего раздела был довольно простым, поэтому он не требовал значительной настройки. К сожалению, нам пришлось в некоторой степени пожертвовать удобочитаемостью и выразительностью ради того, чтобы улучшить модульность и разделение ответственности. В больших приложениях это может стать серьезной проблемой.

На помощь приходят специализированные библиотеки внедрения зависимостей, которые позволяют простым способом помечать зависимости функций или объектов и разрешают зависимости во время выполнения. Все это обычно делается так, чтобы как можно меньше влиять на общую структуру кода.

Существует много библиотек внедрения зависимостей для Python, так что вам определенно не придется строить свой вариант с нуля. Эти библиотеки обычно похожи друг на друга по реализации и функциональности, поэтому мы возьмем одну из них и разберемся, как ее можно применить в приложении со счетчиком просмотров.

Мы воспользуемся библиотекой `injector`, свободно доступной в PyPI. Она выбрана по нескольким причинам:

- **Достаточно зрелая и активно развивается**: библиотека существует более 10 лет, новые версии выпускаются через каждые несколько месяцев.
- **Поддерживает фреймворки**: сообщество обеспечивает поддержку различных фреймворков, включая Flask (с помощью пакета `flask-injector`).
- **Поддерживает аннотации типов**: библиотека позволяет писать ненавязчивые аннотации зависимостей и применять средства статического анализа типов.
- **Простая**: `injector` обладает «питоническим» API, благодаря чему код легко читать и осмыслять.

Чтобы установить `injector` в вашей среде с помощью `pip`, используйте следующую команду:

```
$ pip install injector
```

Дополнительная информация об `injector` доступна по адресу https://github.com/alecthomas/injector.

В нашем примере будет использоваться пакет `flask-injector`, который предоставляет шаблонный код для бесшовной интеграции `injector` с Flask. Но сначала мы разделим приложение на несколько модулей, чтобы реалистичнее смоделировать крупный продукт. В конце концов, внедрение зависимостей по-настоящему проявляет себя в приложениях с большим количеством компонентов.

Мы создадим следующие модули Python:

- `interfaces`: модуль, содержащий наши интерфейсы. В нем будет размещен интерфейс `ViewsStorageBackend` из предыдущего раздела без каких-либо изменений.
- `backends`: модуль с конкретными реализациями подсистем хранения данных. Здесь будут размещены реализации `CounterBackend` и `RedisBackend` из предыдущего раздела без каких-либо изменений.
- `tracking`: модуль с конфигурацией приложения и функциями представлений.
- `di`: модуль с определениями библиотеки `injector`, которые позволяют автоматически разрешать зависимости.

Центральный класс библиотеки `injector` — `Module`. В нем определен так называемый **контейнер внедрения зависимостей** — атомарный блок отображений

между интерфейсами зависимостей и экземплярами их фактических реализаций. Минимальный подкласс `Module` может выглядеть так:

```
from injector import Module, provider

def MyModule(Module):
    @provider
    def provide_dependency(self, *args) -> Type:
        return ...
```

Декоратор `@provider` помечает методы класса `Module`, которые предоставляют реализацию для конкретного интерфейса `Type`. Создание некоторых объектов может быть достаточно сложным, поэтому `injector` позволяет модулям иметь также вспомогательные методы без декораторов.

У метода, который предоставляет зависимость, могут быть собственные зависимости. Они определяются как аргументы метода с аннотациями тинов. Это позволяет каскадно разрешать зависимости. `injector` поддерживает компоновку контекстов внедрения зависимостей из нескольких модулей, поэтому не обязательно определять все зависимости в одном модуле.

На основе приведенного выше шаблона можно создать наш первый модуль с впедрением зависимостей в файле di.py. Модуль будет пазываться `CounterStorage` и предоставлять реализацию `CounterBackend` для интерфейса `ViewsStorageBackend`. Онределение выглядит так:

```
from injector import Module, provider, singleton

from interfaces import ViewsStorageBackend
from backends import CounterBackend

class CounterStorage(Module):
    @provider
    @singleton
    def provide_storage(self) -> ViewsStorageBackend:
        return CounterBackend()
```

`CounterStorage` пе принимает аргументов, поэтому пе требуется онределять еще какие-то зависимости. От общего шаблона модуля он отличается только декоратором `@singleton`. Это явная реализация паттерна проектирования Одиночка (Singleton) — класса, у которого может быть только один экземпляр. В данном контексте это означает, что каждый раз при разрешении этой зависимости `injector` будет возвращать одип и тот же объект. Это необходимо из-за того, что `CounterStorage` хранит счетчики просмотров во внутреннем атрибуте `_counter`. Без декоратора `@singleton` каждый занрос к реализации `ViewsStorageBackend` возвращал бы совершенно новый объект, а следовательно, мы бы ностоянно теряли счетчики просмотров.

Класс `RedisStorage` реализуется немногим сложнее:

```
from injector import Module, provider, singleton
from redis import Redis

from interfaces import ViewsStorageBackend
from backends import RedisBackend

class RedisStorage(Module):
    @provider
    def provide_storage(self, client: Redis) -> ViewsStorageBackend:
        return RedisBackend(client, "my-set")

    @provider
    @singleton
    def provide_redis_client(self) -> Redis:
        return Redis(host="redis")
```

В файлах с кодом этой главы содержится полная среда docker-compose с предварительно настроенным образом Redis Docker, чтобы вам не требовалось устанавливать Redis на своем хосте.

В модуле `RedisStorage` используется возможность библиотеки `injector` разрешать каскадные зависимости. Конструктору `RedisBackend` нужен экземпляр клиента Redis, так что его можно нередать как еще один аргумент метода `provide_storage()`. Средства модуля `injector` распознают аннотации типов и автоматически подбирают метод, предоставляющий экземпляр класса Redis. Можно пойти еще дальше и извлечь аргумент хоста, чтобы отделить зависимость конфигурации. Мы не станем этого делать для простоты.

Теперь нужно связать все воедино в модуле `tracking`. Будем полагаться на то, что `injector` разрешит зависимости в представлениях. Это означает, что можно наконец определить обработчики `track()` и `stats()` с дополнительными аргументами `storage` и зарегистрировать их с декоратором `@app.route()`, как если бы они были обычными представлениями Flask. Обновленные сигнатуры будут выглядеть так:

```
@app.route('/stats')
def stats(storage: ViewsStorageBackend):
    ...

@app.route('/track')
def track(storage: ViewsStorageBackend):
    ...
```

Остается определить финальную конфигурацию приложения, в которой указано, из каких модулей надо получать реализации интерфейсов. Если вы хотите

использовать RedisBackend, то модуль `tracking` нужно завершить следующим кодом:

```
import di

if __name__ == '__main__':
    FlaskInjector(app=app, modules=[di.RedisStorage()])
    app.run(host="0.0.0.0", port=8000)
```

Ниже приведен полный код модуля `tracking`:

```
from http import HTTPStatus

from flask import Flask, request, Response
from flask_injector import FlaskInjector

from interfaces import ViewsStorageBackend
import di

app = Flask(__name__)

PIXEL = (
    b'GIF89a\x01\x00\x01\x00\x80\x00\x00\x00'
    b'\x00\x00\xff\xff\xff!\xf9\x04\x01\x00'
    b'\x00\x00\x00,\x00\x00\x00\x00\x01\x00'
    b'\x01\x00\x00\x02\x01D\x00;'
)

@app.route('/track')
def track(storage: ViewsStorageBackend):
    try:
        referer = request.headers["Referer"]
    except KeyError:
        return Response(status=HTTPStatus.BAD_REQUEST)

    storage.increment(referer)

    return Response(
        PIXEL, headers={
            "Content-Type": "image/gif",
            "Expires": "Mon, 01 Jan 1990 00:00:00 GMT",
            "Cache-Control": "no-cache, no-store, must-revalidate",
            "Pragma": "no-cache",
        }
    )

@app.route('/stats')
def stats(storage: ViewsStorageBackend):
    return storage.most_common(10)

@app.route("/test")
```

```
def test():
    return """
    <html>
    <head></head>
    <body><img src="/track"></body>
    </html>
    """

if __name__ == '__main__':
    FlaskInjector(app=app, modules=[di.RedisStorage()])
    app.run(host="0.0.0.0", port=8000)
```

Как видите, основной код приложения почти не изменился от механизма внедрения зависимостей. Этот код очень похож на первую и самую простую версию, где еще не использовалась IoC. Добавив несколько определений интерфейсов и модулей `injector`, мы получили заготовку для модульного приложения, которую теперь будет легко расширять. Например, можно дополнить ее механизмами хранения других аналитических данных или подключить дашборд, позволяющий просматривать данные в разных разрезах.

Другое преимущество внедрения зависимостей — слабая связанность (ослабление связей между компонентами). В нашем примере представления не создают никаких экземпляров подсистем хранения или клиентов используемой службы (в случае `RedisBackend`). Они зависят от общих интерфейсов, но не зависят от реализаций. Слабая связанность обычно служит хорошей основой для качественной архитектуры приложения.

Конечно, пользу от инверсии управления и внедрения зависимостей трудно продемонстрировать на компактном примере вроде того, который мы рассмотрели. Дело в том, что потенциал этих приемов в полной мере раскрывается в больших приложениях. Как бы то ни было, мы вернемся к примеру с пикселями отслеживания в главе 10 «Автоматизация тестирования и контроля качества», где будет показано, что IoC значительно улучшает тестируемость кода.

Итоги

Эта глава была своего рода путешествием во времени. Python считается современным языком, но чтобы лучше понять его паттерны, нам пришлось совершить ряд экскурсов в историю.

Мы начали с интерфейсов — концепции почти такой же древней, как ООП (первый объектно-ориентированный язык — Simula — появился в 1967 году!). Мы рассмотрели `zope.interface` — пожалуй, одну из самых старых активно сопровождаемых интерфейсных библиотек в экосистеме Python. Вы узнали о ее

достоинствах и недостатках. Все это позволило лучше понять ее две основные альтернативы в Python: абстрактные базовые классы и структурную подтипизацию через расширенные аннотации типов.

После знакомства с интерфейсами мы перешли к теме инверсии управления. Источники информации в интернете нередко сбивают с толку, путая эту концепцию с внедрением зависимостей. Чтобы расставить все точки над «i», мы отследили происхождение этого термина до 1980-х годов, когда никто еще даже не задумывался о контейнерах внедрения зависимостей. Вы узнали, как распознать инверсию управления в разных формах и как она позволяет улучшить модульность приложений. Сначала мы попытались вручную инвертировать управление в простом приложении. Мы убедились, что иногда за такие попытки приходится расплачиваться удобочитаемостью и выразительностью. Благодаря этому удалось понять, в чем состоит ценность готовых библиотек внедрения зависимостей и как они упрощают разработку.

В следующей главе мы полностью сменим панравление и отойдем от темы объектно-ориентированного программирования, языковых средств, наттернов проектирования и парадигм. Вся глава будет полностью посвящена теме конкурентного выполнения. Вы научитесь писать код, который выполняет разные операции параллельно, и, будем надеяться, делает это быстро.

6

Конкурентное выполнение

Конкурентное (concurrent) выполнение и одно из его воплощений — параллельная обработка — входят в число самых обширных областей в сфере разработки ПО. Эта тема настолько огромна, что о ней можно нанисать десятки книг и при этом все равно не охватить всех важных аспектов и моделей. В этой главе мы покажем, почему для вашего приложения может потребоваться конкурентное выполнение, когда его использовать и какие основные его модели представлены в Python.

Мы обсудим некоторые средства языка, встроенные модули и сторонние накеты, которые позволяют реализовать эти модели в вашем коде. Впрочем, мы не сможем глубоко погрузиться во все детали. Содержимое этой главы можно рассматривать как отправную точку для самостоятельных изысканий. Мы постараемся продемонстрировать ключевые идеи и помочь разобраться, действительно ли в вашем приложении нужна конкурентность. Будем надеяться, что к концу главы вы сможете решить, какой подход лучше отвечает вашим потребностям.

В этой главе рассматриваются следующие темы:

- Что такое конкурентное выполнение.
- Многоточность.
- Многопроцессная обработка.
- Асинхронное программирование.

Прежде чем браться за основные концепции конкурентности, рассмотрим технические требования.

Технические требования

Ниже перечислены пакеты Python, которые используются в этой главе и доступны для загрузки из PyPI:

- `requests`
- `aiohttp`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%206.

Но перед тем как углубиться в различные реализации конкурентности, доступные программистам на Python, поговорим о том, что же обозначают термином «конкурентное выполнение».

Что такое конкурентное выполнение?

Конкурентное выполненне, или **конкурентность** (concurrency), часто путают с конкретными способами его реализации. Некоторые программисты считают этот термин синонимом параллельной обработки. Вот почему нужно начать с полпоценного определепия конкурентности. Только после этого можно будет правильно нонять разные модели конкурентного выполнения и ключевые различия между ними.

Прежде всего, конкурентное выполнение — не то же самое, что параллелизм (parallelism). Конкурентность также не является особепностью реализации приложения. Конкурентность — это свойство программы, алгоритма или задачи, тогда как параллелизм — всего лишь один из возможных подходов к решению задач, которые являются конкурептными.

В статье Лесли Лампорта (Leslie Lamport) «Time, Clocks, and the Ordering of Events in Distributed Systems» («Время, часы и порядок событий в распределенной системе») (1976) конценция конкурентного выполнения определена так:

> *Два события считаются конкурентными, если ни одно из них не влияет на другое через причинно-следственную связь.*

Экстраполируя это определение па программы, алгоритмы или задачи, можно сказать, что их уместно пазывать конкурентными, если они полпостью или

частично раскладываются на компоненты (единицы), очередность следования которых не важна. Такие единицы можно обрабатывать независимо друг от друга, и порядок обработки не повлияет на конечный результат. Это означает, что они, в частности, могут обрабатываться одновременно или параллельно: в этом случае действительно можно говорить о параллелизме. Однако это необязательная схема.

Естественным следствием конкурентной природы задач становится распределенное выполнение работы, желательно с помощью многоядерных процессоров или вычислительных кластеров. Тем не менее это не единственный эффективный механизм конкурентного выполнения. Во многих сценариях конкурентные задачи решаются асинхронными методами, которые не требуют параллельного выполнения. Другими словами, если задача конкурентна, это дает возможность решать ее специальным (желательно — более эффективным) образом.

Как правило, мы склонны решать задачи классическим способом, выполняя некую последовательность шагов. Именно так большинство из нас мыслит и обрабатывает информацию — с помощью синхронных алгоритмов, которые в каждый момент времени делают что-то одно, шаг за шагом. Однако этот способ обработки информации плохо подходит для крупномасштабных задач или для ситуаций, в которых приходится одновременно обслуживать нескольких пользователей или программных агентов:

- если время обработки задания ограничивается быстродействием одной вычислительной единицы (одного компьютера, ядра процессора и т. д.);
- если вы не можете принимать и обрабатывать новые входные данные, пока ваша программа не завершит обработку предыдущих.

Эти проблемы порождают три основных сценария работы приложений, в которых конкурентное выполнение помогает удовлетворить потребности пользователей:

- **Распределенная обработка**: масштаб задачи настолько огромен, что ее можно завершить за приемлемое время (с ограниченными ресурсами), только если распределить выполнение по нескольким вычислительным единицам, которые могут обрабатывать нагрузку параллельно.
- **Отзывчивость приложения**: приложение должно реагировать на действия пользователя (принимать новые входные данные), даже если обработка предыдущих данных еще не завершена.
- **Фоновая обработка**: не все задачи должны выполняться синхронно. Если не требуется, чтобы результаты того или иного действия были доступны немедленно, может иметь смысл отложить его выполнение на более позднее время.

Сценарий распределенной обработки напрямую соответствует параллельному выполнению. Именно поэтому в нем обычно применяются многопоточные и многопроцессные модели. Для сценария отзывчивости часто не требуется параллельная обработка, так что реальное решение в конечном итоге зависит от специфики задачи. Проблема отзывчивости также охватывает ситуации, в которых приложение должно независимо обслуживать нескольких клиентов (пользователей или программных агентов) так, чтобы ни один из них не ждал, пока обслужат другого.

Интересно, что эти группы задач не исключают друг друга. Часто требуется обеспечивать отзывчивость приложения, при этом не имея возможности обрабатывать все входные данные на одном процессоре. Именно поэтому разные и даже на первый взгляд конфликтующие подходы к конкурентному выполнению часто применяются одновременно. Такая ситуация особенно характерна для разработки веб-серверов, где иногда приходится использовать асинхронные циклы событий или программные потоки в сочетании с множественными процессами, чтобы задействовать все доступные ресурсы и при этом обеспечить низкую задержку при высокой нагрузке.

Python поддерживает несколько способов конкурентного выполнения. Вот важнейшие из них:

- **Многопоточность** (multithreading): несколько активных потоков выполнения совместно используют контекст памяти родительского процесса. Это одна из самых популярных и старых моделей конкурентного выполнения. Она лучше всего работает в приложениях, которые выполняют интенсивные операции ввода/вывода или должны обеспечить высокую отзывчивость пользовательского интерфейса. Многопоточность не требует особых затрат ресурсов, но создает множество скрытых ловушек и угроз безопасности при работе с памятью.
- **Многопроцессность** (multiprocessing): несколько независимых процессов выполняют работу распределенно. Эта схема похожа на многопоточность, хотя в ней нет общего контекста памяти. В применении к Python многопроцессность лучше подходит для приложений с интенсивным использованием ЦП. Это решение более ресурсоемко, чем многоточность, и требует реализовывать паттерны межпроцессных коммуникаций, чтобы координировать работу процессов друг с другом.
- **Асинхронное программирование**: несколько взаимодействующих задач запускаются в рамках одного процесса приложения. Взаимодействующие задачи работают как программные потоки, хотя переключение между ними обеспечивается самим приложением, а не ядром операционной системы. Такая схема подходит для приложений, ориентированных на ввод/вывод, особенно для программ, которые должны одновременно обслуживать мно-

жество сетевых нодключений. Недостаток асинхронного программирования в том, что приходится использовать снециальные асинхронные библиотеки.

Перейдем к подробному обсуждению многопоточности.

Многопоточность

Разработчики часто считают многопоточность очень сложной темой. Хотя это чистая правда, Python обеспечивает высокоуровневые классы и функции, которые значительно упрощают работу с потоками. CPython страдает от отдельных неприятных особенностей реализации, которые снижают эффективность потоков но сравнению с другими языками программирования (такими, как C или Java). Но это не означает, что они полностью бесполезны в Python. Существует достаточно широкий дианазон задач, которые эффективно и удобно решаются именно с помощью потоков Python.

В этом разделе мы обсудим недостатки многопоточпости в CPython, а также популярные конкурентные задачи, для которых потоки Python бывают полезны.

Что такое многопоточность?

Программный поток (или просто поток) — это отдельная ветвь выполнения. Программист может разбить работу на потоки, которые выполняются одновременно. Потоки связаны с родительским процессом и могут легко взаимодействовать друг с другом, потому что они совместно используют один и тот же контекст памяти. Выполнение потоков координируется ядром ОС.

Многопоточные программы работают быстрее на многопроцессорных или многоядерных компьютерах, где каждый поток может выполняться на отдельном ядре. Это общее правило действует в большинстве языков программирования. В Python выигрыш в быстродействии от многопоточного выполнения на многоядерных процессорах ограничен, о чем мы поговорим позже. Для простоты пока будем считать, что упомянутое выше правило также верно п для Python.

Простейший способ запустить новый поток в Python — это использовать класс `threading.Thread()`, как в следующем примере:

```
def my_function():
    print("печатаем из потока")

if __name__ == "__main__":
    thread = Thread(target=my_function)
    thread.start()
    thread.join()
```

`my_function()` — функция, которая должна выполняться в новом потоке. Она передается конструктору класса `Thread` как именованный аргумент `target`. Экземпляры этого класса инкапсулируют потоки приложений и управляют ими.

Чтобы запустить новый поток, недостаточно создать новый экземпляр класса `Thread`, нужно еще вызвать метод `start()`. После того как новый поток запущен, он продолжит выполняться параллельно с основным потоком, пока функция `target` не завершится. В приведенном примере с помощью метода `join()` мы явно заставили программу ожидать, пока новый поток завершится.

Метод join() называется блокирующей операцией. Это означает, что поток ничего не делает (и не расходует процессорное время), а просто ожидает, пока наступит некоторое событие.

Методы `start()` и `join()` позволяют создать и запустить сразу несколько потоков. Эта простая модификация предыдущего примера запускает несколько потоков и присоединяет их к основной программе:

```
from threading import Thread

def my_function():
    print("печатаем из потока")

if __name__ == "__main__":
    threads = [Thread(target=my_function) for _ in range(10)]
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()
```

Все потоки совместно используют один контекст памяти. Это означает, что нужно очень внимательно следить за тем, как ваши потоки обращаются к одним и тем же структурам данных. Если два параллельных потока обновляют одну переменную без какой-либо защиты, то возможна ситуация, в которой небольшие временны́е расхождения при выполнении потоков приведут к тому, что результат изменится неожиданным образом. Чтобы лучше понять суть проблемы, рассмотрим небольшую программу, где несколько потоков читают и обновляют одно значение:

```
from threading import Thread

thread_visits = 0

def visit_counter():
```

```
    global thread_visits
    for i in range(100_000):
        value = thread_visits
        thread_visits = value + 1

if __name__ == "__main__":
    thread_count = 100
    threads = [
        Thread(target=visit_counter)
        for _ in range(thread_count)
    ]
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    print(f"{thread_count=}, {thread_visits=}")
```

Программа запускает 100 потоков, каждый из которых пытается прочитать и увеличить переменную `thread_visits` 100 000 раз. Если бы эти операции выполнялись последовательно, то итоговое значение переменной `thread_visits` равнялось бы 10 000 000. Но когда потоки переплетаются, результат становится менее предсказуемым. Сохраним этот фрагмент кода в файле thread_visits.py и запустим его несколько раз, чтобы увидеть реальные результаты:

```
$ python3 thread_visits.py
thread_count=100, thread_visits=6859624
$ python3 thread_visits.py
thread_count=100, thread_visits=7234223
$ python3 thread_visits.py
thread_count=100, thread_visits=7194665
```

При каждом запуске мы получили повое число, и все эти числа очепь далеки от ожидаемого зпачения 10 000 000. Но эти результаты пе соответствуют реальпо-му количеству обновлений переменной `thread_visits`. При таком большом количестве потоков они пачипают переплетаться друг с другом, и результат искажается.

Эта ситуация называется **гонкой** (race hazard или race condition) и считается одной из самых ненавистных причип ошибок в мпогопоточных приложениях. Очевидно, между операциями чтения и записи переменной `thread_visits` существует промежуток времени, в течепие которого другой поток может вмешаться и исказить результат.

Можно подумать, будто проблема решается оператором `+=`, который выглядит как атомарная операция:

```
def visit_counter():
    global thread_visits
    for i in range(100_000):
        thread_visits += 1
```

Но и это не поможет! Оператор `+=` — это всего лишь сокращенная запись для увеличения переменной, но на самом деле в интерпретаторе Python ей соответствуют несколько операций. Между этими операциями образуются промежутки, куда могут «вклиниться» другие потоки.

Чтобы предотвратить гонку, принято использовать **примитивы блокировки** (locking primitives). Python предоставляет несколько классов блокировки в модуле `threading`. Здесь мы воспользуемся простейшим из них — `threading.Lock`. Вот потоково-безопасная модификация функции `visit_counter()`:

```
from threading import Lock

thread_visits = 0
thread_visits_lock = Lock()

def visit_counter():
    global thread_visits
    for i in range(100_000):
        with thread_visits_lock:
            thread_visits += 1
```

Запустив измененную версию кода, вы убедитесь, что при использовании блокировок обращения потоков подсчитываются правильно. Однако за это приходится расплачиваться быстродействием. Объект `Lock()` гарантирует, что в каждый момент времени к блоку кода может обращаться только один поток. Это означает, что защищенный блок не может выполняться параллельно в нескольких потоках. Более того, установка и снятие блокировки — это операции, которые требуют дополнительных затрат. Если много потоков пытаются работать с блокировкой, быстродействие заметно снижается. Позже в этой главе вы увидите другие примеры того, как блокировки используются для защиты параллельного доступа к данным.

Многопоточность обычно поддерживается на уровне ядра ОС. Если компьютер оснащен одним процессором с одним ядром, система использует механизм квантования времени: процессор переключается с одного потока на другой так быстро, что возникает иллюзия их одновременного выполнения.

Одноядерные процессоры сейчас редко встречаются в настольных компьютерах, но еще могут попадаться в других областях. У небольших дешевых машин в облачных вычислительных платформах, а также недорогих встроенных систем часто бывают одноядерные или виртуальные процессоры.

Очевидно, что на единственном вычислительном устройстве параллелизм может быть только виртуальным, и на таком оборудовании трудно оценить его пользу с точки зрения быстродействия. Тем не менее иногда имеет смысл реализовать код с потоками, даже если он предназначен для выполнения на одном ядре. Такие сценарии будут продемонстрированы позднее.

Все меняется, если в вашей среде исполнения доступно несколько процессоров или процессорных ядер. В таких случаях ядро ОС может распределять потоки между ними, что позволяет программам выполняться намного быстрее. Это справедливо для многих языков программирования, но не всегда верно для Python. Чтобы понять, почему так происходит, необходимо глубже разобраться в том, как работают потоки в Python.

Как работают потоки в Python

В отличие от некоторых других языков, Python использует несколько потоков уровня ядра, которые могут запускать любые потоки уровня интерпретатора. За выполнение и планирование потоков уровня ядра отвечает ядро ОС. Чтобы создавать и присоединять потоки, CPython использует системные вызовы ОС. Он не может в полной мере контролировать, когда и на каких ядрах процессора будут выполняться потоки, — за это отвечает исключительно системное ядро. Более того, ядро может в любой момент вытеснить выполняемый поток, — например, чтобы запустить поток с более высоким приоритетом.

К сожалению, стандартная реализация Python (интерпретатор CPython) устанавливает принциниальное ограничение, которое во многих контекстах уменьшает практическую пользу от потоков. Все операции, которые обращаются к объектам Python, **сериализуются** одной глобальной блокировкой. Это происходит из-за того, что многие внутренние структуры интерпретатора не являются потоково-безопасными и нуждаются в защите. Не каждая операция требует блокировки, и в некоторых ситуациях потоки снимают блокировку.

В контексте параллельной обработки под термином **«сериализация»** подразумевается последовательное выполнение действий (одно за другим). Как правило, в конкурентных программах лучше избегать непреднамеренной сериализации.

Этот механизм интерпретатора Python называется **глобальной блокировкой интерпретатора**, или **GIL** (Global Interpreter Lock). Тема отказа от GIL время от времени всплывает в списке рассылки Python-dev, и создатели Python неоднократно подтверждали, что в перспективе планируют избавиться от этого механизма. К сожалению, на момент написания книги никому не удалось предложить разумное и простое решение, которое позволило бы устранить это

ограничение. Крайне маловероятно, что в ближайшее время ситуация изменится. Рациональнее всего предположить, что GIL останется в CPython, и мы должны научиться жить с этим.

Как же работает многопоточность в Python? Если потоки содержат только чистый код на Python и не выполняют никаких операций ввода/вывода (например, взаимодействия через сокеты), то использование потоков мало поможет ускорить работу программы: GIL с большой вероятностью будет глобально сериализовать выполнение всех потоков. Но помните, что GIL защищает только объекты Python. На практике GIL отключается на некоторых блокирующих системных вызовах, например на вызовах сокетов. Также GIL может не действовать на фрагментах расширений на языке C, где не используются никакие функции Python/C API. Это означает, что несколько потоков могут работать в полностью параллельном режиме, если они выполняют операции ввода/вывода или код расширения C, специально приспособленный для этого режима.

Взаимодействие с GIL в C-расширениях Python подробно рассматривается в главе 9 «Интеграция Python с C и C++».

Многопоточность позволяет эффективно использовать время, когда программа ожидает доступности внешнего ресурса. Если поток в спящем режиме отключил GIL (это часто происходит во внутренней реализации CPython), он может находиться в состоянии ожидания и «проснуться», когда поступят результаты. Наконец, если программа должна предоставлять интерфейс с небольшим временем отклика, то многопоточность может помочь даже в одноядерных средах, где ОС приходится квантовать время. А в многопоточных средах программа может легко взаимодействовать с пользователем, параллельно выполняя интенсивные вычисления в фоновом режиме.

Учтите, что GIL бывает не во всех реализациях языка Python. Это ограничение есть в CPython, Stackless Python и PyPy, но не в Jython (Python для JVM) и IronPython (Python для .NET). Также предпринималась попытка разработать версию PyPy без GIL. Она была основана на программно реализованной транзакционной памяти и называлась PyPy-STM.

В следующем разделе мы рассмотрим более конкретные примеры ситуаций, в которых потоки могут принести пользу.

Когда использовать многопоточность?

Несмотря на все ограничения GIL, потоки могут пригодиться в следующих случаях:

- **Отзывчивые приложеиия**: приложения, которые могут принимать новые входные данные и реагировать в пределах задапного временного окна, даже если они не завершили обработку предыдущих входных данных.
- **Миогопользовательские приложения и сетевые коммуникации**: приложения, которые должны принимать ввод от нескольких пользователей одновременно, часто взаимодействуют с ними по сети. Это означает, что влияние блокировки можно существенно ослабить, задействуя те компоненты CPython, где GIL отключена.
- **Делегироваиие работы и фоиовая обработка**: приложения, в которых большую часть «черной работы» выполняют внешние приложения или службы, а ваш код играет роль шлюза для работы с этими ресурсами.

Начнем с отзывчивых приложений, потому что именно в них разработчики часто отдают предпочтение многопоточности перед другими моделями совместного выполнения.

Отзывчивые приложения

Допустим, вы с помощью графического интерфейса заставляете ОС скопировать большой файл из одной папки в другую. Вероятно, задача копирования станет выполняться в фоновом режиме, а окно пользовательского интерфейса будет отображать постояпно обновляемый индикатор выполнепия. Таким образом вы будете непрерывно нолучать обратную связь по ходу всего процесса, а также сможете отменить операцию. Во время конирования можно выполпять другую работу, например просматривать веб-страницы или редактировать документы, и графический интерфейс системы будет реагировать на ваши действия. Этот подход пе вызывает такого раздражения, как команда `cp` или `copy` командной оболочки, которая не нредоставляет никакой обратной связи, пока вся работа не завершится.

С отзывчивым интерфейсом также можно работать над песколькими задачами одновремеппо. Например, Gimp (популярный графический редактор с открытым исходным кодом) позволит вам редактировать одпо изображение в то время, пока другое обрабатывается фильтром, потому что эти две задачи не зависят друг от друга.

Разрабатывая отзывчивые интерфейсы, желательно вынести продолжительные задачи в фоповый режим или по крайней мере постоянно предоставлять пользователям обратную связь. Для этого проще всего воспользоваться нотоками, чтобы пользователь мог работать с интерфейсом, даже если приложению нужно выполнять свои задачи в течение продолжительного времени.

Этот подход часто используется в сочетании с событийным программированием, при котором главный поток приложения отправляет фоновым рабочим

потокам события, подлежащие обработке (см. главу 7). Веб-браузеры — характерный пример приложений, где используется такой архитектурный паттерн.

Не путайте отзывчивость приложения (application responsiveness) с концепцией адаптивного веб-дизайна, который иногда тоже называется отзывчивым (RDW, Responsive Web Design). Это популярный подход в веб-разработке, который позволяет приложению хорошо отображаться на разных устройствах (например, настольных браузерах, смартфонах или планшетах).

Многопользовательские приложения

Одновременное обслуживание нескольких пользователей — особый случай отзывчивости приложения. Особенность этого сценария в том, что приложение должно обслуживать параллельный ввод от многих пользователей, и у каждого пользователя могут быть свои представления о том, как быстро приложение должно реагировать на его действия. Проще говоря, ни один пользователь не должен дожидаться, пока будет обработан ввод других пользователей, чтобы получить свой результат.

Многопоточность — популярная модель конкурентного выполнения для многопользовательских приложений, которая чрезвычайно распространена в веб-приложениях. Например, главный поток веб-сервера может принимать все входящие подключения, но передавать обработку каждого запроса специально выделенному потоку. Обычно это позволяет обрабатывать несколько подключений и запросов одновременно; их количество ограничивается только скоростью, с которой главный поток способен принимать подключения и передавать запросы новым потокам. Ограничение такого подхода заключается в том, что он может быстро израсходовать слишком много ресурсов. Потоки не бесплатны: они используют общую память, но каждому потоку необходимо как минимум выделить собственный стек. Если потоков слишком много, суммарное потребление памяти может быстро выйти из-под контроля.

Другая модель многопоточных многопользовательских приложений предполагает, что всегда существует ограниченный пул рабочих потоков, которые способны обрабатывать входные данные пользователей. В этом случае главный поток отвечает только за выделение рабочих потоков в пределах этого пула. Этот паттерн тоже часто применяется в веб-приложениях. Например, веб-сервер может создать ограниченное количество потоков, каждый из которых будет принимать подключения самостоятельно и обрабатывать все запросы, поступающие по этим подключениям. Этот подход обычно позволяет обслуживать меньше пользователей одновременно (по сравнению с созданием потока для каждого запроса), зато при этом удается эффективнее контролировать потребление ресурсов. Два популярных веб-сервера, совместимых с Python WSGI, —

Gunicorn и uWSGI — работают примерно по описанной схеме, обслуживая запросы HTTP с помощью рабочих потоков.

Сокращение WSGI означает «Web Server Gateway Interface», то есть «шлюзовый интерфейс веб-сервера». Это обычный стандарт Python (определенный в документе PEP 3333 по адресу https://www.python.org/dev/peps/pep-3333/), который регламентирует взаимодействие между веб-серверами и приложениями и способствует переносимости веб-приложений между веб-серверами. Большинство современных веб-фреймворков и веб-серверов Python основаны на WSGI.

Многопоточность как средство конкурентного выполнения в многопользовательских приложениях обычно требует меньше ресурсов, чем многопроцессность. Отдельные процессы Python используют больше памяти, чем потоки, потому что для каждого из них приходится загружать новый интерпретатор. С другой стороны, избыток потоков тоже обходится слишком дорого. Мы знаем, что GIL не создает особых проблем в приложениях с интенсивным вводом/выводом, однако рано или поздно настает момент, когда требуется выполнить код на Python. Так как на одних только потоках невозможно параллелизировать все компоненты приложения, вам не удастся в полной мере задействовать все ресурсы на машине с многоядерными процессорами и одним-единственным процессом Python. Вот почему оптимальным решением иногда будет гибрид многопроцессного и многопоточного выполнения: несколько рабочих процессов выполняются с несколькими потоками. К счастью, некоторые WSGI-совместимые веб-серверы допускают такую конфигурацию (например, Gunicorn с типом исполнителей `gthread`).

Многопользовательские приложения часто делегируют работу потокам, чтобы обеспечить отзывчивость для большого количества пользователей. Однако делегирование работы само по себе можно рассматривать как отдельный практический сценарий для применения многопоточности.

Делегирование работы и фоновая обработка

Если приложение зависит от множества внешних ресурсов, потоки могут сильно ускорить его работу. Рассмотрим функцию, которая индексирует файлы в папке и сохраняет созданные индексы в базе данных. В зависимости от типа файла функция выполняет разные процедуры обработки. Например, для файлов PDF может применяться одна процедура, а для файлов OpenOffice — другая.

Вместо того чтобы обрабатывать все файлы последовательно, функция может создать по одному потоку для каждой процедуры и передавать задания каждому потоку через очередь. Общее время работы функции будет ближе ко време-

ни выполнения самого медленного индексатора, чем к суммарной продолжительности всей индексации.

Другой распространенный сценарий для применения потоков — множественные сетевые запросы к внешней службе. Например, если нужно получить несколько результатов от удаленного веб-API, в синхронном режиме это может занять много времени, особенно если удаленный сервер расположен на большом расстоянии.

Если перед тем, как отправить новый запрос, вы будете дожидаться завершения предыдущего, то слишком много времени уйдет только на ожидание ответа от внешней службы, а к каждому запросу еще будет добавляться круговая задержка.

Если вы взаимодействуете с достаточно эффективной службой (например, Google Maps API), то она с большой вероятностью сможет обрабатывать большинство ваших запросов параллельно без ущерба для времени реакции на отдельные запросы. Поэтому имеет смысл выполнять несколько запросов в отдельных потоках. Выполняя запрос HTTP, приложение, скорее всего, будет тратить бóльшую часть времени на чтение из сокета TCP. Если делегировать такую работу потокам, это позволит значительно улучшить быстродействие приложения.

Пример многопоточного приложения

Чтобы продемонстрировать, как потоки в Python работают на практике, давайте создадим приложение, которое могло бы от них выиграть. Рассмотрим простую задачу, которую мы уже описывали в предыдущем разделе как распространенный сценарий для многопоточности: отправка параллельных запросов HTTP некоторой удаленной службе.

Допустим, вы хотите получить информацию от веб-службы с помощью нескольких запросов, которые нельзя объединить в один пакетный запрос HTTP. Чтобы пример был реалистичнее, мы используем справочную службу по курсам валют из бесплатного API, доступную по адресу https://www.vatcomply.com. Эта служба выбрана по нескольким причинам:

- Служба общедоступна и не требует ключей аутентификации.
- Интерфейс службы очень прост, и к ней можно легко обращаться с запросами с помощью популярной библиотеки `requests`.
- API использует такой же финансовый формат данных, как многие подобные API. Если служба окажется недоступной (или перестанет быть бесплатной), вы сможете легко заменить базовый URL этого API адресом другой службы.

Бесплатные API появляются и исчезают. Возможно, через какое-то время URL-адреса из этой книги перестанут работать или API будет требовать платной подписки. В таких случаях хорошим вариантом может стать запуск собственной службы.

По адресу https://github.com/exchangeratesapi/exchangeratesapi можно найти код службы валютных курсов, которая использует тот же формат данных, что и API из этой главы.

Наша программа будет получать курсы обмена нескольких валют по отношению друг к другу. Результаты будут выводиться в матрице следующего вида:

```
1 USD =    1.0 USD,  0.887 EUR,    3.8 PLN,   8.53 NOK,  22.7 CZK
1 EUR =   1.13 USD,    1.0 EUR,   4.29 PLN,   9.62 NOK,  25.6 CZK
1 PLN =  0.263 USD,  0.233 EUR,    1.0 PLN,   2.24 NOK,  5.98 CZK
1 NOK =  0.117 USD,  0.104 EUR,  0.446 PLN,    1.0 NOK,  2.66 CZK
1 CZK =  0.044 USD,  0.039 EUR,  0.167 PLN,  0.375 NOK,   1.0 CZK
```

Выбранный нами API предоставляет ряд способов запросить курс одной валюты по отношению к нескольким другим, но, к сожалению, не позволяет запрашивать данные сразу для нескольких базовых валют. Получить курсы для одной базовой валюты очень просто:

```
>>> import requests
>>> response = requests.get("https://api.vatcomply.com/rates?base=USD")
>>> response.json()
{'base': 'USD', 'rates': {'BGN': 1.7343265053, 'NZD': 1.4824864769,
'ILS': 3.5777245721, 'RUB': 64.7361000266, 'CAD': 1.3287221779, 'USD':
1.0, 'PHP': 52.0368892436, 'CHF': 0.9993792675, 'AUD': 1.3993970027,
'JPY': 111.2973308504, 'TRY': 5.6802341048, 'HKD': 7.8425113062,
'MYR': 4.0986077858, 'HRK': 6.5923561231, 'CZK': 22.7170346723,
'IDR': 14132.9963642813, 'DKK': 6.6196683515, 'NOK': 8.5297508203,
'HUF': 285.09355325, 'GBP': 0.7655848187, 'MXN': 18.930477964, 'THB':
31.7495787887, 'ISK': 118.6485767491, 'ZAR': 14.0298838344, 'BRL':
3.8548372794, 'SGD': 1.3527533919, 'PLN': 3.8015429636, 'INR':
69.3340427419, 'KRW': 1139.4519819101, 'RON': 4.221867518, 'CNY':
6.7117141084, 'SEK': 9.2444799149, 'EUR': 0.8867606633}, 'date': '2019-
04-09'}
```

Чтобы примеры оставались компактными, мы будем выполнять запросы HTTP с помощью пакета `requests`. Он не входит в стандартную библиотеку, но его можно легко загрузить из PyPI с помощью `pip`. Дополнительная информация о `requests` доступна по адресу https://requests.readthedocs.io/.

Поскольку наша цель — сравнить многопоточное решение конкурентной задачи с классическим синхронным решением, мы начнем с реализации, которая

вообще не использует потоки. Вот код программы, которая перебирает список базовых валют, занрашивает у API их курсы и выводит результаты в стандартный вывод в виде отформатированной текстовой таблицы:

```
import time

import requests

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]

    # Курс валюты по отношению к самой себе равен 1:1
    rates[base] = 1.

    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")

def main():
    for base in BASES:
        fetch_rates(base)

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("затраченное время: {:.2f}s".format(elapsed))
```

Функция `main()` перебирает кортеж базовых валют и вызывает функцию `fetch_rates()`, которая получает курсы обмена этих валют. Впутри `fetch_rates()` отправляется один запрос HTTP с помощью функции `requests.get()`. Метод `response.raise_for_status()` выдает исключение, если сервер возвращает ответ с кодом состояния, обозначающим ошибку сервера или клиента. Пока мы не ожидаем никаких исключений и просто предполагаем, что после отправки запроса можпо успешно прочитать полезные данные ответа методом `response.json()`. О том, как правильно обрабатывать исключения, возникающие в потоках, рассказано в разделе «Обработка ошибок в потоках».

Вокруг вызова функции `main()` мы добавили несколько предложений, которые измеряют, сколько времени понадобилось па всю работу. Сохраните код в файле synchronous.py и запустите его, чтобы носмотреть, как он работает:

```
$ python3 synchronous.py
```

На моем компьютере выполнение этой программы уложилось в несколько секунд:

```
1 USD =     1.0 USD,   0.823 EUR,    3.73 PLN,     8.5 NOK,    21.5 CZK
1 EUR =    1.22 USD,     1.0 EUR,    4.54 PLN,    10.3 NOK,    26.2 CZK
1 PLN =   0.268 USD,    0.22 EUR,     1.0 PLN,    2.28 NOK,    5.76 CZK
1 NOK =   0.118 USD,  0.0968 EUR,   0.439 PLN,     1.0 NOK,    2.53 CZK
1 CZK =  0.0465 USD,  0.0382 EUR,   0.174 PLN,   0.395 NOK,     1.0 CZK
затраченное время: 4.08s
```

Каждый запуск сценария занимает разпое время. Дело в том, что время обработки в основном зависит от удаленной службы, к которой программа обращается по сети. На окончательный результат влияет множество недетерминированных факторов. Если бы мы собирались действовать по-настоящему методично, следовало бы нровести более тщательные тесты, повторить их многократно и вычислить среднее по полученпым результатам. Но для простоты мы так поступать не будем. Позже вы увидите, что и этого упрощенного подхода достаточпо для демопстрационных целей.

Итак, у нас есть базовая реализация. Пришло время добавить в нее потоки. В следующем разделе мы попытаемся ввести но одному потоку на каждый вызов функции `fetch_rates()`.

Использование одного потока на единицу работы

Попробуем улучшить написанную программу. В ней сам Python производит не так уж много работы, а долгое время выполнения обусловлено взаимодействием с внешней службой. Запрос HTTP отправляется удаленному серверу, тот вычисляет ответ, а затем мы ожидаем, пока ответ придет обратно.

Во всем этом широко задействован ввод/вывод, так что многопоточность выглядит разумным вариантом. Все запросы можно отправить одновременно в разных потоках, а затем просто дожидаться нолучения данных по каждому из них. Если служба, с которой мы взаимодействуем, умеет обрабатывать запросы конкурентно, то мы определенно увидим, что быстродействие улучшилось.

Начнем с простейшего подхода. В модуле `threading` Python предоставляет чистую и простую в использовапии абстракцию системных потоков. Центральное место в этом модуле занимает класс `Thread`, который представляет одиночный

экземпляр потока. Ниже приведена измененная версия функции `main()`, которая создает и занускает новый ноток для каждой обрабатываемой базовой валюты, после чего ожидает завершения всех нотоков:

```
from threading import Thread

def main():
    threads = []
    for base in BASES:
        thread = Thread(target=fetch_rates, args=[base])
        thread.start()
        threads.append(thread)

    while threads:
        threads.pop().join()
```

Это решение на скорую руку, которое нодходит к задаче несколько легкомысленно. У него есть серьезные недостатки, и нам придется заняться ими нозднее. Но, так или иначе, оно работает. Сохраните измененный сценарий в файле one_thread_per_item.py и запустите его, чтобы посмотреть, улучшилось ли быстродействие:

```
$ python3 one_thread_per_item.py
```

У себя на комньютере я вижу, что общее время обработки заметно сократилось:

```
1 EUR =    1.22 USD,     1.0 EUR,    4.54 PLN,    10.3 NOK,    26.2 CZK
1 NOK =   0.118 USD,  0.0968 EUR,   0.439 PLN,     1.0 NOK,    2.53 CZK
1 CZK =  0.0465 USD,  0.0382 EUR,   0.174 PLN,   0.395 NOK,     1.0 CZK
1 USD =     1.0 USD,   0.823 EUR,    3.73 PLN,     8.5 NOK,    21.5 CZK
1 PLN =   0.268 USD,    0.22 EUR,     1.0 PLN,    2.28 NOK,    5.76 CZK

затраченное время: 1.34s
```

Из-за того что функция `print()` используется в потоке, итоговый вывод может оказаться слегка искаженным. Это одна из проблем многопоточности, которой мы займемся позднее в этом разделе.

Как видите, нотоки благотворно влияют на приложение. Тенерь пора использовать их более рационально. Для начала отметим недостатки в предыдущем коде:

- Для каждого параметра запускается новый поток. Инициализация потока занимает некоторое время, но эти незначительные затраты — не единственная нроблема. Потоки также потребляют другие ресурсы, такие как память или файловые дескрипторы. В нашем примере ввод состоит из четко опре-

деленного количества элементов, но что, если бы такого предела не было? Не лучшая идея — запускать неограниченное количество потоков, зависящее от произвольного размера входных данных.

- Функция `fetch_rates()`, выполняемая в потоках, вызывает встроенную функцию `print()`. На практике крайне маловероятно, что она будет уместна где-нибудь за пределами главного потока приложения. В основном это связано с тем, как буферизуется стандартный вывод в Python. Если вызовы `print()` чередуются между потоками, компоненты вывода могут оказаться перемешанными. Кроме того, эта функция считается медленной. Если бездумно использовать ее в нескольких нотоках, это может привести к сериализации, которая сведет на нет преимущества многоноточности.
- Наконец, если делегировать каждый вызов функции отдельному потоку, становится крайне сложно контролировать скорость обработки ввода. Конечно, мы хотим, чтобы обработка происходила как можпо быстрее, но часто внешние службы устанавливают жесткие ограничения на частоту запросов от одного клиента. Иногда разумно спроектировать программу так, чтобы частоту обработки можно было лимитировать: тогда внешние API не заблокируют ваше нриложение за нопытки превышения ограничений.

В следующем разделе вы увидите, как пулы номогают решать проблему слишком большого количества нотоков.

Пулы потоков

Первая нроблема, которую мы нопробуем решить, заключается в том, что в нашей программе никак не ограничено количество нотоков, которые она запускает. Хорошее решение — сформировать пул из четко определенного количества рабочих потоков, которые будут параллельно обрабатывать все задачи и взаимодействовать с главным потоком через некоторую потоково-безонасную структуру данных. Кроме того, нри этом будет легче снравиться с двумя другими недостатками из нредыдущего раздела.

Общая идея заключается в том, чтобы занустить фиксированное количество потоков, которые будут нотреблять единицы работы из очереди, пока та не опустеет. Когда работы не останется, нотоки завершатся, и можно будет выйти из программы. Для коммуникационной структуры данных хорошо подойдет класс `Queue` из встроенного модуля `queue`. Он нредставляет собой реализацию очереди FIFO (First-In First-Out, «нервым пришел — первым ушел»), которая очень похожа на коллекцию `deque` из модуля `collections` и разработана специально для коммуникации между потоками. Ниже приведена изменепная версия функции `main()`, которая занускает ограниченное число рабочих нотоков с помощью новой целевой функции `worker()` и взаимодействует с ними через потоково-безопасную очередь:

```
from queue import Queue
from threading import Thread

THREAD_POOL_SIZE = 4

def main():
    work_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(target=worker, args=(work_queue,))
        for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()
```

В Python есть встроенные средства для работы с пулами потоков. Мы рассмотрим их позднее в разделе «Использование multiprocessing.dummy как интерфейса многопоточности».

Функция `main` инициализирует экземпляр `Queue` как переменную `worker_queue` и помещает все базовые валюты в эту очередь как единицы работы, которые должны быть обслужены рабочими потоками. Затем опа инициализирует потоки в количестве, равном `THREAD_POOL_SIZE`, с целевой функцией `worker()` и входным аргументом `work_queue`. Затем с помощью `work_queue.join()` функция ожидает, пока все элементы будут обработаны, а с помощью вызова метода `join` для каждого экземпляра `Thread` — пока все нотоки завершатся.

Обработка единиц работы из очереди выполняется в функции `worker`:

```
from queue import Empty

def worker(work_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break
        else:
            fetch_rates(item)
            work_queue.task_done()
```

Функция `worker()` запускает цикл `while` до тех пор, пока `work_queue.empty()` не вернет `True`. На каждой итерации она пытается получить новый элемент без блокирования, для чего используется метод `work_queue.get_nowait()`. Если очередь уже пуста, возникает исключение `Empty`, а функция прерывает цикл и завершается. Если же из очереди можно извлечь следующий элемент, то функция `worker()` передает его функции `fetch_rates(item)` и помечает элемент как обработанный — для этого используется `work_queue.task_done()`. Когда все элементы из очереди помечены как обработанные, функция `work_queue.join()` из главного потока возвращает управление.

Остальная часть сценария, а именно функция `fetch_rates()` и код в условии `if __name__ == "__main__"`, остается неизменной. Ниже приведен полный код сценария, который можно сохранить в файле thread_pool.py:

```
import time
from queue import Queue, Empty
from threading import Thread

import requests

THREAD_POOL_SIZE = 4
SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )

    response.raise_for_status()
    rates = response.json()["rates"]
    # Курс валюты по отношению к самой себе равен 1:1
    rates[base] = 1.

    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")

def worker(work_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break
        else:
            fetch_rates(item)
            work_queue.task_done()

def main():
```

```
    work_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(target=worker, args=(work_queue,))
        for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("затраченное время: {:.2f}s".format(elapsed))
```

Теперь можно запустить сцепарий и посмотреть, отличается ли он по быстродействию от предыдущей версии:

```
$ python thread_pool.py
```

На моем компьютере получается такой результат:

```
1 NOK =   0.118 USD,  0.0968 EUR,   0.439 PLN,     1.0 NOK,    2.53 CZK
1 PLN =   0.268 USD,    0.22 EUR,     1.0 PLN,    2.28 NOK,    5.76 CZK
1 USD =     1.0 USD,   0.823 EUR,    3.73 PLN,     8.5 NOK,    21.5 CZK
1 EUR =    1.22 USD,     1.0 EUR,    4.54 PLN,    10.3 NOK,    26.2 CZK
1 CZK =  0.0465 USD,  0.0382 EUR,   0.174 PLN,   0.395 NOK,     1.0 CZK

затраченное время: 1.90s
```

Программа может выполняться медленнее, чем в случае, когда создается отдельный поток па аргумент, но по крайней мере нам теперь не грозит исчерпание всех вычислительных ресурсов, если размер ввода пе будет ограничен. Кроме того, параметр `THREAD_POOL_SIZE` можно отрегулировать, чтобы достичь более эффективного баланса между затратами времени и ресурсов.

В этой версии использовалась исходная модификация функции `fetch_rates()`, которая направляет результат API в стандартный вывод прямо из потока. В пе-

которых случаях это может исказить вывод, если два потока попытаются выводить результаты одновременно. В следующем разделе мы попробуем улучшить реализацию за счет двусторонних очередей.

Двусторонние очереди

Теперь мы готовы решить проблему потенциальных искажений результатов при выводе из потока. Намного лучше поручить вывод главному потоку, который запускает рабочие потоки. Для этого можно настроить еще одну очередь, которая будет собирать результаты от рабочих нотоков. Ниже приведен полный код, в котором все связывается воедино (основные изменения выделены):

```
import time
from queue import Queue, Empty
from threading import Thread

import requests

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

THREAD_POOL_SIZE = 4

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]

    # Курс валюты по отношению к самой себе равен 1:1
    rates[base] = 1.
    return base, rates

def present_result(base, rates):
    rates_line = ", ".join([
        f"{rates[symbol]:7.03} {symbol}"
        for symbol in SYMBOLS
    ])
    print(f"1 {base} = {rates_line}")

def worker(work_queue, results_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break
        else:
            results_queue.put(fetch_rates(item))
```

```
            work_queue.task_done()

def main():
    work_queue = Queue()
    results_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(
            target=worker,
            args=(work_queue, results_queue)
        ) for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()

    while not results_queue.empty():
        present_result(*results_queue.get())

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("затраченное время: {:.2f}s".format(elapsed))
```

Новая версия отличается прежде всего тем, что в ней ноявился экземпляр `results_queue` класса `Queue` и функция `present_result()`. Фупкция `fetch_rates()` больше не нанравляет свои результаты в стандартный вывод. Вместо этого она возвращает обработанные результаты API непосредственно функции `worker()`, а рабочие потоки передают эти результаты без изменений повой выходной очереди `results_queue`.

Теперь только главный поток отвечает за нечать результатов в стандартный вывод. После того как вся работа будет помечена как выполненная, функция `main()` извлекает результаты из `results_queue` и передает их функции `present_result()`.

Тем самым исключается риск перемешивапия данных, которое могло бы возникнуть, если бы функция `present_result()` выполняла больше вызовов `print()`. При небольших входных данных от этой меры пе стоит ожидать особого улуч-

шения быстродействия, но на самом деле мы также снижаем риск сериализации потоков, которая бывает из-за медленного выполнения `print()`.

Во всех приведенных примерах предполагалось, что используемый API всегда будет выдавать содержательный и корректный ответ. Для простоты мы не рассматривали сценарии сбоев, но в реальном приложении они могут нарушить работу программы. В следующем разделе вы увидите, что произойдет, если в потоке возникает исключение, и как это влияет на взаимодействие между очередями.

Обработка ошибок в потоках

Метод `raise_for_status()` объекта `requests.Response` выдает исключение, если код состояния ответа HTTP соответствует ошибке. Мы использовали этот метод во всех предыдущих итерациях функции `fetch_rates()`, но нока не обрабатывали потепциальные исключения.

Если служба, к которой мы обращаемся с помощью метода `requests.get()`, отвечает с кодом состояния ошибки, то исключение возникнет в отдельном потоке и не приведет к сбою всей программы. Конечно, соответствующий рабочий поток немедленно завершится. Однако главный поток будет ожидать, пока завершатся все задачи, хранящиеся в очереди `work_queue` (с номощью вызова `work_queue.join()`). Если не принять меры, то может возникнуть ситуация, когда часть рабочих процессов завершилась аварийно, и нрограмма никогда не закончит работу. Значит, следует позаботиться о том, чтобы рабочие потоки корректно обрабатывали возможные исключения, и проверить, что все элементы из очереди были обработаны.

Внесем в код небольшие изменения, чтобы подготовиться к возможным проблемам. Если в рабочем потоке возникает исключение, можно номестить экземпляр ошибки в очередь `results_queue`, чтобы главный поток мог определить, каким задачам не удалось выполнить обработку. Кроме того, можно нометить текущую задачу как выполненную — так же, как если бы ошибка не возникла. Тогда главный поток не окажется навсегда заблокирован в ожидании вызова `work_queue.join()`.

После этого главный поток может проанализировать результаты и заново возбудить любые исключепия, обнаруженные в очереди результатов. Вот усовершепствованные версии функций `worker()` и `main()`, которые могут работать с исключениями на более безонасном уровне (изменепия выделены жирным шрифтом):

```
def worker(work_queue, results_queue):
    while not work_queue.empty():
        try:
```

```
            item = work_queue.get_nowait()
        except Empty:
            break

        try:
            result = fetch_rates(item)
        except Exception as err:
            results_queue.put(err)
        else:
            results_queue.put(result)
        finally:
            work_queue.task_done()

def main():
    work_queue = Queue()
    results_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(target=worker, args=(work_queue, results_queue))
        for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()

    while not results_queue.empty():
        result = results_queue.get()
        if isinstance(result, Exception):
            raise result

        present_result(*result)
```

Чтобы продемонстрировать обработку ошибок в действии, попробуем смоделировать реалистичный сценарий с возникновением ошибок. Поскольку мы не контролируем используемый API, будем внедрять ошибки случайным образом в функцию `fetch_rates()`. Ниже приведена измененная версия этой функции:

```
import random

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )
```

```
        if random.randint(0, 5) < 1:
            # Ошибки моделируются переопределением кода статуса
            response.status_code = 500

        response.raise_for_status()
        rates = response.json()["rates"]
        # Курс валюты по отношению к самой себе равен 1:1
        rates[base] = 1.
        return base, rates
```

Присваивая `response.status_code` значение `500`, мы моделируем ситуацию, в которой API возвращает ответ, указывающий на ошибку сервера. В таких ситуациях подробная информация об ошибке доступна не всегда. Такого кода состояния вполне достаточно для того, чтобы метод `response.raise_for_status()` выдал исключение.

Сохраните измененную версию кода в файле **error_handling.py** и запустите его, чтобы увидеть, как обрабатываются исключения:

```
$ python3 error_handling.py
```

Ошибки внедряются случайным образом, так что, возможно, нрограмму придется запустить неоднократно. После нескольких попыток вы получите результат, который выглядит примерно так:

```
1 PLN =   0.268 USD,    0.22 EUR,     1.0 PLN,    2.28 NOK,    5.76 CZK
Traceback (most recent call last):
  File ".../error_handling.py", line 92, in <module>
    main()
  File ".../error_handling.py", line 85, in main
    raise result
  File ".../error_handling.py", line 53, in worker
    result = fetch_rates(item)
  File ".../error_handling.py", line 30, in fetch_rates
    response.raise_for_status()
  File ".../.venv/lib/python3.9/site-packages/requests/models.py", line
943, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 500 Server Error: OK for url: https://
api.vatcomply.com/rates?base=NOK
```

Наш код не смог успешно получить все данные, но по крайней мере мы увидели четкую информацию о причине ошибки — код состояния 500 «Ошибка сервера».

В следующем разделе мы внесем последнее улучшение в эту многопоточную программу: механизм лимитирования, который защитит программу от принудительного ограничения частоты занросов и предотвратит непамеренные злоупотребления бесплатной службой.

Лимитирование

Последняя проблема, упомянутая в разделе «Использование одного потока на единицу работы», — потенциальные ограничения частоты запросов, которые может установить провайдер внешней службы. В случае с API курсов валют специалисты по сопровождению не сообщали нам о каких-либо ограничениях частоты запросов или механизмах лимитирования. Тем не менее многие службы (и даже платные) часто применяют подобные ограничения.

Как правило, когда служба ограничивает частоту запросов, то после превышения квоты она начинает возвращать ответы с сообщениями об ошибке. В предыдущем разделе мы уже подготовились к таким ответам, но этого часто недостаточно для корректной обработки ограничений. Дело в том, что многие службы подсчитывают запросы, сделанные с превышением лимита, и при хроническом превышении вы рискуете никогда не вернуться к разрешенной частоте.

При использовании нескольких потоков очень легко превысить все возможные ограничения частоты запросов, а если служба не лимитирует входящий трафик, то и перегрузить ее до такого состояния, когда она вообще перестанет реагировать на запросы. Если такую перегрузку устраивают намеренно, она называется **DoS-атакой** (Denial of Service, «отказ в обслуживании»).

Чтобы вписываться в ограничения и не устраивать случайных DoS-атак, нужно ограничить темпы отправки запросов к удаленной службе. Такое ограничение часто называется **лимитированием** (throttling). В PyPI есть несколько простых и удобных пакетов, позволяющих лимитировать работу любого вида. Впрочем, здесь внешний код использоваться не будет. Лимитирование дает возможность представить несколько примитивов блокировки для многопоточных приложений, поэтому мы попробуем построить свое решение с нуля.

Мы применим очень простой алгоритм, который называется **маркерной корзиной** (token bucket). Он основан на следующих принципах:

- Существует «корзина» с заранее определенным количеством маркеров.
- Каждый маркер соответствует одному разрешению на обработку одной единицы работы.
- Каждый раз, когда рабочий поток запрашивает один или несколько маркеров (разрешений), алгоритм работает так:

    1. Проверить, сколько времени прошло с момента последнего заполнения корзины.
    2. Если прошло достаточно времени, заполнить корзину количеством маркеров, соответствующим разности во времени.

3. Если количество хранимых маркеров больше либо равно запрошенному количеству, уменьшить количество хранимых маркеров до запрошенной величины и вернуть ее.
4. Если количество хранимых маркеров меньше запрошенного, вернуть 0.

Пара важных моментов: всегда инициализируйте корзину нулевым количеством маркеров и следите за тем, чтобы она никогда не переполнялась. Может показаться, что это противоречит здравому смыслу, но если нарушить эти условия, маркеры могут начать выдаваться сериями так, что предельная частота окажется превышена. Поскольку в нашей ситуации ограничение частоты выражается в запросах в секунду, нам не придется возиться с произвольными промежутками времени. Примем за базовое значение 1 секунду и не будем хранить в корзине больше маркеров, чем количество запросов, разрешенных за этот промежуток времени. Вот пример реализации класса, который обеспечивает лимитирование по принципу маркерной корзины:

```
from threading import Lock

class Throttle:
    def __init__(self, rate):
        self._consume_lock = Lock()
        self.rate = rate
        self.tokens = 0
        self.last = None

    def consume(self, amount=1):
        with self._consume_lock:
            now = time.time()

            # Отсчет времени инициализируется при первом запросе
            # маркера, чтобы предотвратить всплеск запросов
            if self.last is None:
             self.last = now

         elapsed = now - self.last

         # Проверяем, что прошедший квант времени
         # достаточно велик для добавления новых маркеров
         if elapsed * self.rate > 1:
             self.tokens += elapsed * self.rate
             self.last = now

         # Корзина никогда не должна переполняться
         self.tokens = min(self.rate, self.tokens)

         # Выдать маркеры, если они доступны
         if self.tokens >= amount:
```

```
            self.tokens -= amount
            return amount

        return 0
```

Пользоваться этим классом очень просто. Нужно создать один экземпляр `Throttle` (например, `Throttle(10)`) в главном потоке и передавать его каждому рабочему потоку как позиционный аргумент:

```
def main():
    work_queue = Queue()
    results_queue = Queue()
    throttle = Throttle(10)

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(
            target=worker,
            args=(work_queue, results_queue, throttle)
        ) for _ in range(THREAD_POOL_SIZE)
    ]
    ...
```

Все потоки будут совместно использовать один и тот же экземпляр `throttle`, но это безопасно, потому что манипуляции с его внутренним состоянием защищены экземпляром класса `Lock` из модуля `threading`. Теперь можно обновить реализацию функции `worker()`, чтобы она ожидала с каждым элементом, пока объект `throttle` не освободит новый маркер:

```
import time

def worker(work_queue, results_queue, throttle):
    while True:
        try:
            item = work_queue.get_nowait()
        except Empty:
            break

        while not throttle.consume():
            time.sleep(0.1)

        try:
            result = fetch_rates(item)
        except Exception as err:
            results_queue.put(err)
        else:
            results_queue.put(result)
            finally:
                work_queue.task_done()
```

Блок `while not throttle.consume()` не позволяет обрабатывать элементы рабочей очереди, если объект `throttle` не освободил маркеры (нулевое количество маркеров интерпретируется как `False`). Мы добавили короткую паузу, чтобы немного стимулировать потоки в случае пустой корзины. Вероятно, существует и более элегантное решение, но этот простой способ хорошо справляется со своей задачей.

Когда `throttle.consume()` возвращает ненулевое значение, маркер считается использованным. Поток может выйти из цикла `while` и начать обрабатывать элемент рабочей очереди. Когда обработка будет завершена, поток считывает следующий элемент из рабочей очереди и снова пытается использовать маркер. Весь этот процесс продолжается, пока очередь не опустеет.

Это было очень краткое введение в тему потоков. Мы не рассмотрели все возможные аспекты многопоточных приложений, но вы уже знаете достаточно, чтобы приступить к изучению других моделей конкурентного выполнения и сравнить их с потоками. Следующей моделью, которую мы рассмотрим, будет многопроцессность.

Многопроцессность

Откровенно говоря, многопоточность — непростая тема. Для безопасной и эффективной работы с потоками требуется огромный объем кода по сравнению с синхронными решениями. Выше нам приходилось создавать пулы потоков и коммуникационные очереди, корректно обрабатывать исключения из потоков, а также заботиться о безопасности потоков, реализуя ограничение частоты запросов. Десятки строк кода нужны лишь для того, чтобы выполнить одну функцию из внешней библиотеки в параллельном режиме! При этом мы надеемся, что создатели внешнего пакета обеспечили потоковую безопасность своей библиотеки. Создается впечатление, что решение, которое приносит реальную пользу только в задачах с интенсивным вводом/выводом, обходится слишком дорого.

Альтернативный подход, который позволяет добиться параллелизма, — многопроцессная обработка. Отдельные процессы Python, которые не ограничивают друг друга с помощью GIL, позволяют более эффективно использовать ресурсы. Это особенно важно для приложений, которые работают на многоядерных процессорах и выполняют задачи с интенсивными вычислениями. На текущий момент это единственное встроенное решение из области конкурентного выполнения, которое доступно разработчикам Python (с использованием интерпретатора CPython) и в любой ситуации позволяет извлечь пользу из наличия нескольких ядер процессора.

У многопроцессности есть и другое преимущество перед многопоточностью: процессы не связаны общим контекстом памяти, следовательно, меньше ве-

роятность повредить данные, а также устроить гонки или взаимные блокировки. Отсутствие общего контекста памяти означает, что понадобятся дополнительные усилия, чтобы обмениваться данными между отдельными процессами. К счастью, существует много хороших механизмов для надежных межпроцессных коммуникаций. Более того, в Python есть несколько примитивов, при наличии которых взаимодействие между процессами на практике не сложнее, чем между потоками.

Как правило, простейший способ запустить новый процесс в любом языке программирования заключается в ветвлении программы в некоторой точке. В POSIX-совместимых системах (таких, как UNIX, macOS и Linux) ветвление — это системный вызов, который создает новый дочерний процесс. В Python этот вызов выражается функцией `os.fork()`. После ветвления каждый процесс продолжает выполняться сам по себе. Рассмотрим пример сценария, в котором ветвление происходит ровно один раз:

```
import os

pid_list = []

def main():
    pid_list.append(os.getpid())
    child_pid = os.fork()

    if child_pid == 0:
        pid_list.append(os.getpid())
        print()
        print("CHLD: привет, я дочерний процесс")
        print("CHLD: все PID'ы, которые я знаю: %s" % pid_list)

    else:
        pid_list.append(os.getpid())
        print()
        print("PRNT: привет, я родительский процесс")
        print("PRNT: PID дочернего процесса: %d" % child_pid)
        print("PRNT: все PID'ы, которые я знаю: %s" % pid_list)


if __name__ == "__main__":
    main()
```

Функция `os.fork()` порождает новый процесс. Оба процесса обладают одинаковым состоянием памяти до момента вызова `fork()`, но после этого вызова память каждого процесса развивается по-своему — в этом и заключается ветвление. `os.fork()` возвращает целое значение. Если оно равно 0, мы знаем, что текущий процесс является дочерним. Родительский процесс получает идентификатор процесса (PID) своего дочернего процесса.

Сохраним сценарий в файле forks.py и занустим его в сеансе командной оболочки:

```
$ python3 forks.py
```

На моем компьютере результат выглядит так:

```
PRNT: привет, я родительский процесс
PRNT: PID дочернего процесса: 9304
PRNT: все PID'ы, которые я знаю: [9303, 9303]

CHLD: привет, я дочерний процесс
CHLD: все PID'ы, которые я знаю: [9303, 9304]
```

Обратите внимание: перед вызовом `os.fork()` у обоих процессов состояние данных полностью совпадает. В первом элементе коллекции `pid_list` оба процесса содержат одно и то же значение PID (идентификатор процесса).

Позднее их состояния расходятся. Мы видим, что дочерпий процесс добавил значение 9304, тогда как родитель продублировал свой PID 9303. Это произошло потому, что у каждого процесса свой контекст памяти. Они имеют одинаковые начальные условия, но не могут влиять друг на друга после вызова `os.fork()`.

После ветвления каждый процесс получает собствеппое адресное прострапство. Чтобы взаимодействовать друг с другом, процессы должны работать с ресурсами системного уровня или использовать такие низкоуровпевые средства, как сигналы.

К сожалению, функция `os.fork` недоступна в Windows, где для имитации ветвления приходится порождать новый интерпретатор. Таким образом, реализация многопроцессности зависит от платформы. В модуле `os` также есть функции для норождения новых процессов в Windows. Python предоставляет превосходный модуль `multiprocessing`, который создает высокоуровневый интерфейс для многопроцессной обработки.

Огромное преимущество модуля `multiprocessing` заключается в том, что он предоставляет в готовом виде некоторые абстракции, которые пам приходилось кодировать с нуля при обсуждении мпогопоточности. Этот модуль позволяет сократить объем шаблопного кода, отчего приложение становится проще и удобнее в сопровождении. Как ни страпно, несмотря па имя, модуль `multiprocessing` предоставляет сходный иптерфейс для потоков, так что ничто не мешает использовать один иптерфейс для обоих подходов.

В следующем разделе мы глубже изучим встроенный модуль `multiprocessing`.

Встроенный модуль multiprocessing

Модуль `multiprocessing` предоставляет переносимые средства для работы с процессами, как если бы они были потоками. Этот модуль содержит класс `Process`, который очень похож на класс `Thread` и может использоваться на любой платформе:

```
from multiprocessing import Process
import os

def work(identifier):
    print(
        f'Привет, я процесс '
        f'{identifier}, pid: {os.getpid()}'
    )

def main():
    processes = [
        Process(target=work, args=(number,))
        for number in range(5)
    ]
    for process in processes:
        process.start()

    while processes:
        processes.pop().join()

if __name__ == "__main__":
    main()
```

В классе `Process` есть методы `start()` и `join()`, похожие на методы класса `Thread`. Метод `start()` порождает новый процесс, а вызов `join()` ожидает выхода из дочернего процесса.

Чтобы увидеть возможности `multiprocessing` в действии, сохраните сценарий в файле с именем basic_multiprocessing.py и запустите его:

```
$ python3 basic_multiprocessing.py
```

На моем компьютере результат выглядит примерно так:

```
Привет, я процесс 3, pid: 9632
Привет, я процесс 1, pid: 9630
Привет, я процесс 2, pid: 9631
Привет, я процесс 0, pid: 9629
Привет, я процесс 4, pid: 9633
```

При создании процессов происходит ветвление памяти (в POSIX-совместимых системах). Помимо того что класс `Process` копирует состояние намяти, в его

конструкторе также предусмотрен дополнительный аргумент `args` для передачи данных.

Взаимодействие между процессами требует дополнительных усилий, потому что их локальная память по умолчанию не является общей. Чтобы обойти это затруднение, модуль `multiprocessing` предоставляет следующие способы межпроцессного взаимодействия:

- Класс `multiprocessing.Queue` — функциональный эквивалент класса `queue.Queue`, который использовался ранее для взаимодействия между потоками.
- Класс `multiprocessing.Pipe` — двусторонний коммуникационный канал, напоминающий сокет.
- Модуль `multiprocessing.sharedctypes`, позволяющий создавать произвольные типы C (из модуля `ctypes`) в специальном пуле памяти, который совместно используется процессами.

У классов `multiprocessing.Queue` и `queue.Queue` практически одинаковый интерфейс. Единственное различие заключается в том, что первый предназначен для использования в многопроцессных, а не многопоточных средах, поэтому у него другие внутренние средства передачи данных и примитивы блокировки. О том, как использовать `Queue` в многопоточных программах, уже рассказывалось в разделе «Многопоточность», поэтому не будем повторяться.

Более интересный коммуникационный паттерн предоставляется классом `Pipe`. Это дуплексный (двусторонний) коммуникационный канал, концептуально очень похожий на каналы (pipes) UNIX. Интерфейс `Pipe` напоминает простой сокет из встроенного модуля `socket`. От низкоуровневых системных каналов и сокетов он отличается тем, что автоматически применяет сериализацию объектов с помощью модуля `pickle`. С точки зрения разработчика все выглядит как передача обычных объектов Python. А с простыми системными каналами или сокетами вам пришлось бы настраивать сериализацию вручную, чтобы реконструировать отправленные объекты из байтовых потоков.

Модуль `pickle` легко сериализует и десериализует объекты Python в байтовые потоки и обратно. Он обрабатывает различные типы объектов, включая экземпляры классов, определенных пользователем. За дополнительной информацией о модуле pickle и объектах, с которыми он может работать, обращайтесь по адресу https://docs.python.org/3/library/pickle.html.

Это заметно упрощает взаимодействие между процессами, потому что можно передавать практически все базовые типы Python. Рассмотрим следующий класс `worker()`, который читает объект из объекта `Pipe` и выводит его представление в стандартный вывод:

```
def worker(connection):
    while True:
        instance = connection.recv()
        if instance:
            print(f"CHLD: recv: {instance}")
        if instance is None:
            break
```

Класс `Pipe` можно использовать в функции `main()`, чтобы передавать различные объекты (в том числе пользовательские классы) дочернему процессу:

```
from multiprocessing import Process, Pipe

class CustomClass:
    pass

def main():
    parent_conn, child_conn = Pipe()

    child = Process(target=worker, args=(child_conn,))

    for item in (
        42,
        'some string',
        {'one': 1},
        CustomClass(),
        None,
    ):
        print(
            "PRNT: send: {}".format(item)
        )
        parent_conn.send(item)

    child.start()
    child.join()

if __name__ == "__main__":
    main()
```

Результаты этого сценария показывают, что можно легко передавать экземпляры собственных классов и у них будут разные адреса в зависимости от процесса:

```
PRNT: send: 42
PRNT: send: some string
PRNT: send: {'one': 1}
PRNT: send: <__main__.CustomClass object at 0x101cb5b00>
PRNT: send: None
CHLD: recv: 42
CHLD: recv: some string
```

```
CHLD: recv: {'one': 1}
CHLD: recv: <__main__.CustomClass object at 0x101cba400>
```

Чтобы совместно использовать состояние между процессами, есть и другой способ: применение низкоуровневых типов в общем пуле памяти с классами, предоставляемыми в `multiprocessing.sharedctypes`. Самые важные из них — `Value` и `Array`. Вот примеры кода из официальной документации модуля `multiprocessing`:

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]


if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

Этот пример выводит следующий результат:

```
CHLD: recv: {'one': 1}
CHLD: recv: <__main__.CustomClass object at 0x101cba400>
```

Работая с `multiprocessing.sharedctypes`, не забывайте, что вы имеете дело с общей памятью, поэтому нужно использовать примитивы блокировки, чтобы предотвратить риск гонок. В модуле `multiprocessing` есть некоторые классы, сходные с классами модуля `threading`, такие как `Lock`, `RLock` и `Semaphore`. Недостаток классов из `sharedctypes` заключается в том, что они обеспечивают общий доступ только к базовым типам C из модуля `ctypes`. Если требуется передавать более сложные структуры или экземпляры классов, необходимо использовать `Queue`, `Pipe` или другие каналы межпроцесспых взаимодействий. В большинстве случаев от типов `sharedctypes` лучше держаться подальше, потому что они усложняют код и привносят все опасности многопоточности.

Мы уже упоминали о том, что дополнительные возможности модуля `multiprocessing` позволяют сократить объем шаблонного кода. Одна из таких возможностей — встроенные пулы процессов. В следующем разделе мы поговорим о том, как ими пользоваться.

Пулы процессов

Использование нескольких процессов вместо потоков требует дополнительных ресурсов. Прежде всего растут затраты памяти, потому что у каждого процесса — свой собственный независимый контекст памяти. Это означает, что создавать дочерние процессы в неограниченных количествах более накладно, чем дочерние потоки в многоточных приложениях.

Если ОС поддерживает системный вызов `fork()` с механизмом копирования при записи (COW, Copy on Write), то затраты памяти на запуск новых подпроцессов значительно сокращаются. COW позволяет ОС не дублировать одинаковые страницы памяти, а копировать их только тогда, когда один из подпроцессов попытается их изменить. Например, в Linux есть системный вызов `fork()` с COW, а в Windows — нет. Кроме того, преимущества COW могут нивелироваться в долго выполняющихся процессах.

Лучший способ контролировать расход ресурсов в приложениях, основанных на многопроцессном выполнении, — создать пул процессов по аналогии с тем, как это делалось для нотоков в разделе «Пулы потоков».

А самая замечательная особенность модуля `multiprocessing` — готовый класс `Pool`, который берет на себя все сложности управления множественными рабочими процессами. Реализация нула значительно сокращает объем необходимого нодготовительного кода и уменьшает проблемы, связанные с двусторонней передачей данных. Также не обязательно вручную использовать метод `join()`, потому что `Pool` может работать как диспетчер коптекстов (с помощью инструкции `with`). Ниже приведен один из предыдущих примеров, нереписанный с применением класса `Pool` из модуля `multiprocessing`:

```
import time
from multiprocessing import Pool

import requests

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

POOL_SIZE = 4

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )

    response.raise_for_status()
```

```
    rates = response.json()["rates"]
    # Курс валюты по отношению к самой себе равен 1:1
    rates[base] = 1.
    return base, rates

def present_result(base, rates):
    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")

def main():
    with Pool(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("затраченное время: {:.2f}s".format(elapsed))
```

Как видите, работать с пулом процессов стало проще, потому что нам не приходится управлять рабочими очередями и возиться с методами `start()` и `join()`. Такой код будет проще сопровождать и отлаживать, если возникнут проблемы. Собственно, единственная часть кода, которая явно имеет дело с многопроцессностью, — это функция `main()`:

```
def main():
    with Pool(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)
```

Теперь нам не требуется передавать результаты через явно заданные очереди, а также беспокоиться о том, что произойдет в случае исключений в подпроцессах. Это намного лучше ситуации, в которой приходилось строить пул нотоков с нуля. Вам даже не нужно заботиться о коммуникационных каналах, потому что они создаются неявно в реализации класса `Pool`.

Это не означает, что многоточность всегда вызывает проблемы. В следующем разделе будет показано, как использовать `multiprocessing.dummy` в качестве интерфейса многопоточности.

Использование multiprocessing.dummy в качестве интерфейса многопоточности

У высокоуровневых абстракций из модуля `multiprocessing` (таких, как класс `Pool`) есть множество преимуществ перед простыми инструментами из модуля `threading`. Но отсюда пе следует, что многонроцессность всегда лучше многоноточности. Во многих случаях нотоки оказываются более эффективными, чем нроцессы. Это особенно справедливо для ситуаций, где требуется низкая задержка и/или экономия ресурсов.

Тем не менее это не означает, что нридется жертвовать всеми полезными абстракциями из модуля `multiprocessing` каждый раз, когда вы соберетесь использовать потоки вместо процессов. Модуль `multiprocessing.dummy` имитирует многопроцессный API, но использует множественные потоки вместо ветвления и порождения новых процессов.

Это нозволяет сократить объем подготовительного кода, а также сформировать более модульную структуру кода. Например, вернемся к функции `main()` из предыдущего раздела. Можно нозволить пользователю выбрать подсистему обработки — процессы или потоки. Для этого достаточно заменить класс конструктора объекта `Pool`:

```
from multiprocessing import Pool as ProcessPool
from multiprocessing.dummy import Pool as ThreadPool

def main(use_threads=False):
    if use_threads:
        pool_cls = ThreadPool
    else:
        pool_cls = ProcessPool

    with pool_cls(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)
```

Пул потоков также можно импортировать из модуля multiprocessing.pool как класс ThreadPool. У него будет такая же реализация, так что способ импортирования — не более чем вопрос личных предпочтений.

Этот аспект модуля `multiprocessing` ноказывает, что у многопроцессной и многопоточной обработки много общего. Оба механизма опираются на средства конкурентного вынолнения, нредоставляемые ОС. Они унравляются похожим образом и часто применяют сходные абстракции, чтобы обеспечивать безопасность коммуникаций или намяти.

Совершенно иной подход к конкурентному выполнению — асинхронное программирование, не зависящее от возможностей ОС в части конкурентной обработки информации. Эта модель конкурентности рассматривается в следующем разделе.

Асинхронное программирование

Асинхронное программирование набрало обороты за последние годы. В Python 3.5 наконец-то появились некоторые средства синтаксиса, поддерживающие концепции асинхронного выполнения. Впрочем, это не означает, что до Python 3.5 асинхронное программирование было недоступно. И раньше существовало множество библиотек и фреймворков, большинство из которых восходит к старым версиям Python 2. Существовала даже полностью альтернативная реализация Python, которая называлась Stackless Python и была ориентирована на эту методологию программирования.

Асинхронное программирование на Python проще всего рассматривать как что-то вроде многопоточной модели, но без участия системного планирования. Это означает, что асинхронная программа может обрабатывать информацию конкурентно, но контекст выполнения нереключается внутренними средствами, а не системным планировщиком.

Но конечно, для конкурентного выполнения работы в асинхронной программе потоки не используются. В разных решениях асинхронного программирования применяются разные концепции, и они называются по-разному в зависимости от реализации. Вот несколько названий асинхронных сущностей в различных реализациях:

- Зеленые потоки, или гринлеты (проекты `greenlet`, `gevent` и `eventlet`).
- Корутины, или сопрограммы («родное» асинхронное программирование Python 3.5).
- Мини-задачи, или тасклеты (Stackless Python).

Название «зеленые потоки» происходит от исходной библиотеки потоков для языка Java, которую реализовала группа Green Team («Зеленая команда») в компании Sun Microsystems. Зеленые потоки появились в Java 1.1 и перестали существовать в Java 1.3.

Все это — в целом одна и та же концепция, но реализованная несколько разными способами.

По очевидным причинам в этом разделе мы сосредоточимся только на корутинах, которые поддерживаются во встроенном виде в Python, начиная с версии 3.5.

Кооперативная многозадачность и асинхронный ввод/вывод

Кооперативная многозадачность занимает центральное место в асинхронном программировании. В этой модели многозадачности ОС не отвечает за переключение контекста на другой процесс или поток. Вместо этого каждый процесс добровольно уступает управление в случае бездействия, чтобы несколько программ могли выполняться одновременно. Становится понятно, почему эта многозадачность называется коонеративной: чтобы она работала гладко, необходима кооперация со стороны всех нроцессов.

Эта модель многозадачности раньше применялась в некоторых ОС, но сейчас редко встречается в решениях системного уровня: ведь есть риск, что одна плохо спроектированная служба нарушит стабильность всей системы. В конкурентном выполнении на уровне ОС сейчас доминирует планирование потоков и процессов с прямым переключением контекста онерационной системой. Тем не менее кооперативная многозадачность все еще остается отличным инструментом конкурентости на прикладном уровне.

Реализуя кооперативную многозадачность на уровне приложения, мы не имеем дела с потоками или процессами, которые должны уступать управление, потому что все выполнение происходит в единственном нроцессе и потоке. Вместо этого создаются отдельные задачи (корутины, тасклеты или зеленые потоки), которые уступают управление одной координирующей функции. Такая функция — обычно некоторая разновидность цикла событий.

Чтобы избежать путаницы в будущем (из-за терминологии, принятой в Python), с этого момента мы будем называть такие конкурентные задачи корутинами.

Самый важный вопрос в коонеративной многозадачности — в какой момент передавать управление. В большинстве асинхронных приложений управление нередается планировщику или циклу событий в операциях ввода/вывода. При этом не так важно, читает ли нрограмма данные из файловой системы или обменивается ими через сокет, потому что такие операции ввода/вывода всегда сопровождаются некоторым временем ожидания, пока процесс бездействует. Это время зависит от внешнего ресурса, ноэтому появляется хорошая возможность передать управление другим корутинам, чтобы они выполняли свою работу, пока не попадут в режим ожидания.

Такое поведение отчасти напоминает реализацию многоноточности в Python. Мы знаем, что GIL сериализует нотоки Python, но также освобождается при

каждой операции ввода/вывода. Главное отличие заключается в том, что потоки в Python реализуются потоками системного уровня, так что ОС в любой момент может попытаться вытеснить текущий выполняемый поток и передать управление другому потоку. В асинхроппом программировании задачи пикогда не вытесняются главным циклом событий; вместо этого опи должны явно вернуть управление. Поэтому такой стиль также называется невытесняющей многозадачностью. Он сокращает затраты времепи на переключение контекста и лучше сочетается с реализацией GIL в CPython.

Конечно, каждое приложение на Python выполняется в ОС, где другие процессы конкурируют с ним за ресурсы. Это означает, что ОС всегда имеет право вытеснить весь нроцесс и нередать управление другому процессу. Но когда асинхронное приложение продолжит вынолняться, это произойдет с того же места, где оно было приостановлено при вмешательстве системного планировщика. Поэтому корутины также считаются невытесняющими.

В следующем разделе рассматриваются ключевые слова `async` и `await`, лежащие в основе всей кооперативной многозадачности в Python.

Ключевые слова async и await

Ключевые слова `async` и `await` — основные структурные элементы асинхронного программирования в языке Python.

Ключевое слово `async` перед инструкцией `def` определяет повую корутипу. Выполнение функции корутины может приостанавливаться и возобновляться в строго определепных обстоятельствах. По синтаксису и поведению корутины очень похожи па генераторы. Собственно, в старых версиях Python приходилось использовать генераторы, когда требовалось реализовать корутину. Вот пример объявления фупкции с ключевым словом `async`:

```
async def async_hello():
    print("hello, world!")
```

Фупкции, определенные с ключевым словом `async`, устроены по-особенному. При вызове они не выполняют содержащийся в них код, а возвращают объект корутипы. Рассмотрим следующий пример из интерактивного сеанса Python:

```
>>> async def async_hello():
...     print("hello, world!")
...
>>> async_hello()
<coroutine object async_hello at 0x1014129e8>
```

Объект корутины ничего не делает, пока его выполнение не будет запланировано в цикле событий. Существует модуль `asyncio`, который предоставляет базовую реализацию цикла событий, а также много других асинхронных средств. В следующем примере мы попытаемся вручную запланировать выполнение корутины в интерактивном сеансе Python:

```
>>> import asyncio
>>> async def async_hello():
...     print("hello, world!")
...
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(async_hello())
hello, world!
>>> loop.close()
```

Разумеется, поскольку мы создали только одну нростую корутину, в этом коде нет копкурентного выполнения. Чтобы увидеть его в действии, необходимо создать другие задачи, которые будут вынолняться циклом событий.

Чтобы добавить повые задачи в цикл, можно вызвать метод `loop.create_task()` или передать «ожидаемый» (awaitable) объект функции `asyncio.wait()`. Если у вас есть несколько таких «ожидаемых» задач или сопрограмм, их можно объединить в одип объект с помощью `asyncio.gather()`. Мы используем последний способ и попробуем асинхронпо вывести последовательность чисел, сгенерированную функцией `range()`:

```
import asyncio
import random

async def print_number(number):
    await asyncio.sleep(random.random())
    print(number)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()

    loop.run_until_complete(
        asyncio.gather(*[
            print_number(number)
            for number in range(10)
        ])
    )
    loop.close()
```

Сохраните сцепарий в файле **async_print.py** и занустите его:

```
$ python async_print.py
```

Результат может выглядеть примерно так:

```
0
7
8
3
9
4
1
5
2
6
```

Функция `asyncio.gather()` принимает несколько объектов сопрограмм и немедленно возвращает управление. Так как она может принимать переменное количество позиционных аргументов, мы используем оператор `*`, чтобы распаковать список сопрограмм в аргументах. Как следует из имени, `asyncio.gather()` собирает несколько сопрограмм для конкурентного выполнения. В результате получается так называемый объект Future, который представляет собой будущий результат выполпепия всех собранпых корутин. Метод `loop.run_until_complete()` выполняет цикл событий, пока объект Future не будет завершен.

Мы использовали `asyncio.sleep(random.random())`, чтобы подчеркнуть асинхронную природу корутин. Благодаря этому корутины смогли перемешиваться друг с другом.

С обычной функцией `time.sleep()` не удастся добиться такого же эффекта перемешивапия. Корутины начинают перемешиваться, когда они уступают управление. Для этого используется ключевое слово `await`, которое приостанавливает выполпение корутины, ожидающей результатов другой корутины или объекта Future.

Когда выполнение функции с `await` приостанавливается, она передает управление циклу событий. Чтобы лучше понять, как это работает, придется рассмотреть более сложный пример кода.

Допустим, вы хотите создать две корутины, которые в цикле выполняют одну и ту же простую задачу:

- Ожидать в течение случайного количества секунд.
- Вывести текст, переданпый в аргументе, и время ожидания.

Начнем с простой реализации, в которой не используется `await`:

```
import time
import random
```

```
async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        time.sleep(time_to_sleep)
        print(f"{name} ждал { time_to_sleep } с")
```

Можно заплапировать выполнение нескольких корутин `waiter()` с помощью `asyncio.gather()` так же, как это делалось в сценарии async_print.py:

```
import asyncio

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        asyncio.gather(waiter("первый"), waiter("второй"))
    )
    loop.close()
```

Сохраним код в файле waiters.py и посмотрим, как две корутины `waiter()` выполняются в цикле событий:

```
$ time python3 waiters.py
```

Обратите внимание, что мы использовали утилиту `time`, чтобы измерить общее время выполнепия. Результат может выглядеть так:

```
$ time python waiters.py
первый ждал 0.25 с
первый ждал 0.75 с
первый ждал 0.5 с
первый ждал 0.25 с
второй ждал 0.75 с
второй ждал 0.5 с
второй ждал 0.5 с
второй ждал 0.75 с

real    0m4.337s
user    0m0.050s
sys     0m0.014s
```

Как видите, обе корутины завершили свое выполнение, но не асинхронно. Дело в том, что они обе используют функцию `time.sleep()`, которая блокирует выполнение, но не возвращает управление циклу событий. В многоноточной конфигурации этот механизм работал бы лучше, но сейчас мы не хотим использовать потоки. Как же решить проблему?

Следует использовать `asyncio.sleep()` — асинхронную версию `time.sleep()` — и ожидать ее результата с помощью ключевого слова `await`. Рассмотрим улучшенную версию корутины `waiter()`, в которой используется инструкция `await asyncio.sleep()`:

```
async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        await asyncio.sleep(time_to_sleep)
        print(f"{name} ждал {time_to_sleep} с")
```

Сохраните измененную версию сценария в файле waiters_await.py и занустите ее в командной оболочке. На этот раз вывод двух функций будет перемешан:

```
$ time python waiters_await.py
```

Результат должен выглядеть примерно так:

```
первый ждал 0.5 с
второй ждал 0.75 с
второй ждал 0.25 с
первый ждал 0.75 с
второй ждал 0.75 с
первый ждал 0.75 с
второй ждал 0.75 с
первый ждал 0.5 с

real    0m2.589s
user    0m0.053s
sys     0m0.016s
```

У этого простого улучшения есть еще одно преимущество: с ним код работает быстрее. Общее время выполнения меньше суммы всех времен ожидания, потому что корутины добровольно уступают управление.

Более реалистичный пример асинхронного программирования рассматривается в следующем разделе.

Практический пример асинхронного программирования

Как уже неоднократно упоминалось в этой главе, асинхронное программирование отлично подходит для операций с интенсивным вводом/выводом. Пришло время создать что-то более практичное, чем простой вывод последовательностей или асинхронное ожидание.

Чтобы сохранить логику изложения, мы попробуем решить ту же задачу, которой занимались раньше с использованием многопоточности и многопроцессности. То есть мы будем асинхронно загружать информацию о текущих курсах обмена валют из внешнего ресурса по сети. Было бы замечательно, если бы при этом удалось использовать ту же библиотеку `requests`, что и в нредыдущих

разделах. К сожалению, это невозможно — или, выражаясь точнее, это нельзя сделать эффективно.

К сожалению, библиотека `requests` не поддерживает асинхронный ввод/вывод с ключевыми словами `async` и `await`. Есть другие проекты, которые стараются наделить `requests` возможностями совместного выполнения, но они зависят либо от Gevent (как `grequests`, доступный по адресу https://github.com/kennethreitz/grequests), либо от пулов потоков/процессов (как `requests-futures`, доступный по адресу https://github.com/ross/requests-futures). Ни один из этих вариантов не решает нашу проблему.

С учетом ограничений библиотеки, которая оказалась так полезна в предыдущих примерах, нам понадобится решение, способное заполнить этот пробел. API курсов валют очень прост в использовании, так что нужно найти всего лишь библиотеку HTTP, асинхронную по своей природе. В стандартной библиотеке Python 3.9 все еще нет средств, с помощью которых можно было бы совершать асинхронные вызовы HTTP так же просто, как с помощью `urllib.urlopen()`.

Строить всю поддержку протокола с нуля определенно не хочется, поэтому мы воспользуемся небольшой помощью со стороны пакета `aiohttp`, доступного в PyPI. Это перспективная библиотека, которая добавляет реализацию клиента и сервера для асинхронного HTTP. Ниже приведен небольшой модуль, построенный поверх `aiohttp`: этот модуль создает вспомогательную функцию `get_rates()`, которая отправляет запросы к API службы курсов валют:

```
import aiohttp

async def get_rates(session: aiohttp.ClientSession, base: str):
    async with session.get(
        f"https://api.vatcomply.com/rates?base={base}"
    ) as response:
        rates = (await response.json())['rates']
        rates[base] = 1.

        return base, rates
```

Мы сохраним этот модуль в файле asyncrates.py, чтобы позже его можно было импортировать как модуль `asyncrates`.

Теперь можно переписать пример, который использовался в разделах о многоточности и многопроцессности. Раньше вся работа была разделена на два этапа:

- выполнить все запросы к внешней службе в параллельном режиме с помощью функции `asyncrates.get_rates()`;
- вывести все результаты в цикле с помощью функции `present_result()`.

Основой нашей программы станет простая функция `main()`, которая собирает результаты от нескольких сопрограмм `get_rates()` и передает их функции `present_result()`:

```
async def main():
    async with aiohttp.ClientSession() as session:
        for result in await asyncio.gather(*[
            get_rates(session, base)
            for base in BASES
        ]):
            present_result(*result)
```

Полный код вместе с инструкциями импортирования и инициализацией цикла событий выглядит так:

```
import asyncio
import time

import aiohttp

from asyncrates import get_rates

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

def present_result(base, rates):
    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")

async def main():
    async with aiohttp.ClientSession() as session:
        for result in await asyncio.gather(*[
            get_rates(session, base)
            for base in BASES
        ]):
            present_result(*result)

if __name__ == "__main__":
    started = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    elapsed = time.time() - started

    print()
    print("затраченное время: {:.2f}s".format(elapsed))
```

Результат программы похож на вывод версий, основанных на многоточности и многопроцессном выполнении:

```
$ python async_aiohttp.py
1 USD =     1.0 USD,   0.835 EUR,    3.81 PLN,    8.39 NOK,    21.7 CZK
1 EUR =     1.2 USD,     1.0 EUR,    4.56 PLN,    10.0 NOK,    25.9 CZK
1 PLN =   0.263 USD,    0.22 EUR,     1.0 PLN,     2.2 NOK,    5.69 CZK
1 NOK =   0.119 USD,  0.0996 EUR,   0.454 PLN,     1.0 NOK,    2.58 CZK
1 CZK =  0.0461 USD,  0.0385 EUR,   0.176 PLN,   0.387 NOK,     1.0 CZK

затраченное время: 0.33s
```

Преимущество `asyncio` перед многопоточностью и многопроцессностью заключается в том, что вам не приходится возиться с пулами процессов и безопасностью памяти, чтобы обеспечить сетевое взаимодействие. К недостаткам можно отнести то, что нельзя использовать популярные библиотеки синхронного взаимодействия, например пакет `requests`. Вместо этого мы применили библиотеку `aiohttp`, и для простого API все получилось несложно. Но иногда требуется специализированная клиентская библиотека, которая не является асинхронной и которую невозможпо легко портировать. Такая ситуация рассматривается в следующем разделе.

Интеграция неасинхронного кода с async при помощи объектов Future

Асинхронное программирование — отличная штука, особенно для бэкенд-разработчиков, которых интересует создание масштабируемых приложений. На практике это один из важнейших инструментов для создания серверов с высокой степенью конкурентного выполнения.

Однако реальность сурова. Многие популярные пакеты, предназначенные для задач с интенсивным вводом/выводом, не рассчитаны на использование с асинхронным кодом. Основные причины:

- Современные возможности Python 3 (особепно асинхронное программирование) еще недостаточпо распространены.
- Начинающие программисты на Python плохо понимают различные концепции конкурентного выполнения.

Это означает, что часто миграция существующих синхронных многопоточных приложений и пакетов либо невозможна (из-за архитектурных ограничений), либо обходится слишком дорого. Многие проекты могли бы сильно выиграть от перехода на асинхронную многозадачпость, по лишь немногим из них со временем это удается.

Это означает, что пока еще вы будете сталкиваться со множеством трудностей, пытаясь разрабатывать асинхронные приложения с нуля. Как правило, возник-

нут проблемы вроде тех, которые мы встречали с библиотекой `requests` из раздела «Практический пример асинхронного нрограммирования», — несовместимые интерфейсы и синхронное блокирование операций ввода/вывода.

Конечно, когда вы сталкиваетесь с подобной несовместимостью, иногда можно отказаться от `await` и просто загружать занрашиваемые ресурсы синхронно. Но при этом во время ожидания результатов все остальные корутины будут блокироваться и их код не будет выполняться. Формально такое решепие работает, но с ним теряются все преимущества асинхронного программирования. Таким образом, в копечном итоге комнромиссное сочетание асинхронного ввода/вывода с синхронным нельзя считать эффективным решением. Приходится выбирать что-то одно.

Другая проблема — продолжительные операции с интенсивным использованием ЦП. Когда выполняется операция ввода/вывода, можно без особых проблем уступить управление из корутины. При записи в сокет или чтении из него рано или поздно наступит ожидание, так что использовать `await` — лучшее, что можно сделать. Но как быть, если нужно провести какие-то вычисления и вы знаете, что на них нотребуется некоторое время? Конечно, можно разбить задачу на части и уступать управление вызовом `asyncio.wait(0)` каждый раз, когда работа немного продвигается внеред. Но вскоре вы обнаружите, что это не лучший нодход. Оп приводит к хаосу в коде и при этом пе гарантирует хороших результатов. Квантованием времени должен заниматься интерпретатор или ОС.

Что же делать, если у вас есть код с долгими синхропными операциями ввода/вывода, который вы не можете (или не хотите) перенисать? Или что делать, если у вас происходят интенсивные вычисления в приложении, которое спроектировано в основном с расчетом на асинхронный ввод/вывод? Вам понадобится обходное решение, под которым я подразумеваю многопоточность или многопроцессное выполнение.

Как ни странно, иногда лучшим решением оказывается то, от которого вы пытались уйти. Параллельная обработка задач с интенсивными вычислениями в Python всегда лучше работает с многопроцессностью. А многопоточность справляется с операциями ввода/вывода настолько же эффективно (быстро и без перерасхода ресурсов), как `async` и `await`, если ее правильно настроить и применять с умом.

Таким образом, если что-то просто не вписывается в ваше асинхронное приложение, переместите эту часть кода в отдельный поток или процесс. Можно представить себе, что это была корутина, и вернуть управление в цикл событий с помощью `await`. В конечном итоге результаты будут обрабатываться по мере готовпости. К счастью, в стандартной библиотеке Python есть модуль `concurrent.futures`, также интегрированный в модуль `asyncio`. Эти два модуля позволяют планировать блокирующие функции для выполнения в потоках или дополни-

тельных процессах, как если бы они были асинхронными неблокирующими корутинами.

В следующем разделе мы поближе познакомимся с исполнителями и объектами Future.

Исполнители и объекты Future

Прежде чем разбираться, как внедрять нотоки или процессы в асинхронный цикл событий, стоит более внимательно изучить модуль `concurrent.futures`, который впоследствии станет главным компонентом так называемого обходного решения. Самые важные классы модуля `concurrent.futures` — `Executor` и `Future`.

Исполнитель (`Executor`) представляет пул ресурсов, которые могут обрабатывать единицы работы параллельно. Может показаться, что но своему назначению он очень похож на классы `Pool` и `dummy.Pool` из модуля `multiprocessing`, однако у него совершенно другой интерфейс и семантика. `Executor` — базовый класс, который не нредназначен для создания экземпляров и имеет две конкретные реализации:

- `ThreadPoolExecutor`: представляет пул потоков.
- `ProcessPoolExecutor`: представляет пул процессов.

У каждого исполнителя есть такие три метода:

- `submit(func, *args, **kwargs)`: планирует функцию `func` для выполнения в нуле ресурсов и возвращает объект `Future`, который представляет исполнение вызываемого объекта.
- `map(func, *iterables, timeout=None, chunksize=1)`: выполняет функцию `func` над итерируемым объектом по аналогии с методом `multiprocessing.Pool.map()`.
- `shutdown(wait=True)`: завершает работу исполнителя и освобождает все его ресурсы.

Здесь наиболее интересен метод `submit()` из-за объекта `Future`, который он возвращает. Этот объект представляет асинхронное выполнение вызываемого объекта и лишь косвенно представляет его результат. Чтобы получить фактическое возвращаемое значение переданного вызываемого объекта, нужно вызвать метод `Future.result()`. Если вызываемый объект уже завершился, метод `result()` не блокируется и просто возвращает результат фупкции. В противном случае вызов блокируется, пока результат не будет готов. Рассматривайте его как «обещание» результата (кстати, эта же концепция используется в обещаниях (promises) JavaScript). Объект Future не обязательно распаковывать сразу же после получения (методом `result()`), но если вы попытаетесь это сделать, метод в конечном итоге что-то вернет.

Рассмотрим следующее взаимодействие с `ThreadPoolExecutor` в интерактивном сеансе Python:

```
>>> def loudly_return():
...     print("работаю")
...     return 42
...
>>> from concurrent.futures import ThreadPoolExecutor
>>> with ThreadPoolExecutor(1) as executor:
...     future = executor.submit(loudly_return)
...
работаю
>>> future
<Future at 0x33cbf98 state=finished returned int>
>>> future.result()
42
```

Как видите, `loudly_return()` немедленно вывела строку «работаю» после того, как эта функция была передана исполнителю. Это означает, что выполнение началось еще до того, как мы решили распаковать его значение методом `future.result()`.

В следующем разделе вы узнаете, как использовать исполнители в цикле событий.

Исполнители в цикле событий

Экземпляры класса `Future`, которые возвращает метод `Executor.submit()`, концептуально очень близки к корутинам из асинхронного программирования. Поэтому можно использовать исполнители, чтобы создать гибрид между кооперативной многозадачностью и многопроцессным либо многопоточным выполнением.

В этом обходном решении центральное место занимает метод `run_in_executor(executor, func, *args)` в классе цикла событий. Он позволяет спланировать выполнение функции `func` в пуле процессов или потоков, представленном аргументом `executor`. Ценность этого метода в том, что он возвращает новый ожидаемый объект (объект, которого можно ожидать инструкцией `await`). В результате блокирующую функцию, которая не является корутиной, можно выполнять точно так же, как если бы она была корутиной. А самое важное, что она не блокирует цикл событий и не мешает ему обрабатывать другие корутины, сколько бы времени ни потребовалось для ее завершения. Она останавливает только функцию, которая ожидает результатов такого вызова, но весь цикл событий при этом продолжает работать.

Еще один полезный факт: вам даже не обязательно создавать собственный экземпляр исполнителя. Если в аргументе `executor` передать `None`, будет исполь-

зоваться класс `ThreadPoolExecutor` с числом потоков по умолчанию (для Python 3.9 оно равно количеству процессоров, умноженному на 5).

Допустим, вы не хотите переписывать проблемную часть кода, обращенную к API, из-за которой возникли все неприятности. Блокирующий вызов можно легко вынести в отдельный поток методом `loop.run_in_executor()`, оставив функцию `fetch_rates()` в качестве ожидаемой корутины:

```
async def fetch_rates(base):
    loop = asyncio.get_event_loop()
    response = await loop.run_in_executor(
        None, requests.get,
        f"https://api.vatcomply.com/rates?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]
    # Курс валюты по отношению к самой себе равен 1:1
    rates[base] = 1.
    return base, rates
```

Такое решение уступает полноценной асинхронной библиотеке, но это лучше, чем ничего.

Асинхронное программирование — отличный инструмент для разработки высокопроизводительных конкурентных приложений, которые интенсивно взаимодействуют с другими службами по сети. Это легко делается без проблем с безопасностью памяти, которые обычно присущи многопоточным и (до некоторой степени) многопроцессным решениям. При этом нет принудительного переключения контекста, в результате чего требуется меньше примитивов блокировки, потому что можно легко предсказать, когда корутины вернут управление циклу событий.

К сожалению, есть и оборотная сторона: приходится использовать специализированные асинхронные библиотеки. Для синхронных и многопоточных приложений существует широкий ассортимент клиентских и коммуникационных библиотек, который лучше покрывает потребности этих приложений во взаимодействии с популярными службами. Исполнители и объекты Future позволяют заполнить пробел, но они уступают по эффективности истинно асинхронным решениям.

Итоги

Путь был долгим, но нам успешно удалось рассмотреть все основные механизмы конкурентного программирования, доступные для разработчиков на Python.

Разобравшись, что же такое конкурентное выполнение, мы взялись за дело и проанализировали одну из типичных конкурентных задач с использованием

многопоточности. После того как мы выявили основные недостатки кода и исправили их, мы обратились к многопроцессному выполнению и выяснили, как оно работает в нашем случае. Выяснилось, что работать с множественными процессами средствами модуля `multiprocessing` намного проще, чем использовать простые потоки из модуля `threading`. Но сразу же после этого оказалось, что благодаря модулю `multiprocessing.dummy` тот же API можно использовать и для потоков. Таким образом, выбор между многопроцессностью и многопоточностью теперь определяется лишь тем, какой вариант лучше подходит для задачи, а не тем, у какого решения лучше интерфейс.

А когда речь зашла о выборе решений, мы опробовали асинхронное программирование, которое казалось лучшим подходом для приложений с интенсивным вводом/выводом, — только чтобы убедиться, что нельзя полностью отказаться от потоков и процессов. Круг замкнулся: мы вернулись к тому, с чего начали.

И здесь мы приходим к завершающему выводу этой главы. В области конкурентного выполнения не существует панацеи. Бывают решения, которые нравятся вам больше или меньше. Для разных наборов задач могут лучше подходить разные механизмы, и стоит ориентироваться в них, чтобы успешно программировать. В реальных ситуациях вам, возможно, придется пользоваться всем арсеналом средств и стилей конкурентного выполнения в рамках одного приложения, и такие случаи не редкость.

В следующей главе рассматривается тема, отчасти связанная с моделью конкурентного выполнения: событийное программирование. В этой главе мы сосредоточимся на различных паттернах взаимодействия, которые лежат в основе распределенных асинхронных систем, а также систем с высокой степенью конкурентности.

7

Событийно-ориентированное программирование

В предыдущей главе обсуждались различные модели реализации конкурентного выполнения, доступные в Python. Чтобы лучше объяснить концепцию конкурентности, мы использовали следующее определение:

> *Два события считаются конкурентными, если ни одно из них не влияет на другое через причинно-следственную связь.*

Мы часто представляем себе события как упорядоченные точки на шкале времени, которые происходят друг за другом и передко объединены причинно-следственными связями. Но в программировании события рассматриваются немного иначе: это пе обязательно «то, что происходит». Чаще события понимаются как независимые единицы информации, которые могут обрабатываться программой. И сама эта концепция событий ложится в оспову конкурентного выполнения.

Конкурентное выполпение — парадигма программирования для обработки конкурентных событий. И у этой парадигмы есть обобщение, ориентированное на концепцию событий так таковых, пезависимо от того, используются ли они для конкурентного выполнения. Этот подход, который рассматривает программы как потоки событий, называется **событийным**, или **событийно-ориентированным программированием** (event-driven programming).

Эта парадигма важна, потому что она позволяет легко уменьшить связанность даже в больших и сложных системах. Она помогает проводить четкие границы между независимыми компонентами и улучшает изоляцию между ними.

В этой главе рассматриваются следующие темы:

- Что такое событийное программирование.
- Разные стили событийного программирования.
- Событийные архитектуры (или архитектуры, управляемые событиями).

Прочитав эту главу, вы освоите стандартные приемы событийного программирования и узнаете, как экстранолировать их на событийные архитектуры. Также вы научитесь быстро распознавать задачи, которые эффективно решаются с помощью событийных программ.

Технические требования

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `flask`
- `blinker`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%207.

В этой главе мы разработаем простое приложение с помощью пакета графического интерфейса (GUI), который называется `tkinter`. Для запуска примеров `tkinter` вам понадобится библиотека Tk для Python. В большинстве дистрибутивов Python она доступна по умолчанию, но в некоторых операционных системах понадобится установить дополнительные системные пакеты. В дистрибутивах Linux на базе Debian этот пакет обычно называется `python3-tk`. Экземпляры Python, поступившие из официальных программ установки macOS и Windows, должны изначально содержать библиотеку Tk.

Что такое событийное программирование?

Событийное программирование ориентируется на события (часто называемые **сообщениями**), которые передаются между разными программными компонентами. На самом деле оно встречается во многих видах программных продуктов. Уже несколько десятилетий подряд событийное программирование остается самой распространенной парадигмой для программных продуктов, которые

напрямую взаимодействуют с пользователем. Таким образом, эта парадигма естественно подходит для графических интерфейсов. Везде, где программа должна ожидать ввода от пользователя, этот ввод можно смоделировать в виде событий или сообщений. В таком контексте событийная программа часто представляет собой набор обработчиков событий/сообщений, которые реагируют на действия пользователя.

Конечно, события не обязаны возникать только в результате взаимодействия с пользователем. Архитектура любого веб-приложения тоже является событийной. Веб-браузеры отправляют запросы веб-серверам по инициативе пользователя, и эти запросы часто обрабатываются как отдельные события взаимодействия. Некоторые запросы действительно являются результатом прямого ввода пользователя (например, когда он отправляет форму или щелкает по ссылке), однако бывают запросы совершенно другой природы. Многие современные приложения умеют асинхронно синхронизировать данные с веб-сервером без какого-либо участия пользователя, и это взаимодействие происходит незаметно для него.

Вкратце, событийное программирование — это обобщенный способ связывания (coupling) программных компонентов различного размера на разных уровнях программной архитектуры. В зависимости от масштаба и типа архитектуры, с которой вы имеете дело, оно может принимать разные формы:

- Модель конкурентного выполнения, напрямую поддерживаемая семантическими средствами языка программирования (например, `async/await` в Python).
- Способ структурирования кода приложения с помощью диспетчеров/обработчиков событий, сигналов и т. д.
- Обобщенная архитектура коммуникации между процессами или службами, которая позволяет объединять независимые программные компоненты в более крупные системы.

В следующем разделе мы разберемся, чем событийное программирование отличается от асинхронного.

Событийное != асинхронное

Хотя парадигма событийного программирования в высшей степени типична для асинхронных систем, это не означает, что каждое событийное приложение должно быть асинхронным. Это также не означает, что событийное программирование подходит только для конкурентных и асинхронных приложений. Вообще говоря, событийный подход чрезвычайно полезен даже для того, чтобы уменьшать связанность в чисто синхронных и совершенно неконкурентных задачах.

Например, возьмем триггеры, которые встречаются практически во всех реляционных СУБД. Триггер — это хранимая процедура, которая выполняется в ответ на определенное событие, произошедшее в базе данных. Этот распространенный структурный элемент СУБД среди прочего позволяет базе данных поддерживать целостность данных в сценариях, которые не удается легко смоделировать с помощью ограничений баз данных.

Например, база данных PostgreSQL различает три типа событий на уровне строк, которые могут происходить в таблицах или представлениях:

- `INSERT`: происходит при вставке новой строки.
- `UPDATE`: происходит при обновлении существующей строки.
- `DELETE`: происходит при удалении существующей строки.

В случае строк таблицы можно определять триггеры, которые должны выполняться до (`BEFORE`) или после (`AFTER`) конкретного события.

Таким образом, с точки зрения связанности «событие — процедура» каждый триггер `AFTER/BEFORE` можно рассматривать как отдельное событие. Чтобы лучше это понять, рассмотрим пример триггеров баз данных в PostgreSQL:

```
CREATE TRIGGER before_user_update
    BEFORE UPDATE ON users
    FOR EACH ROW
    EXECUTE PROCEDURE check_user();

CREATE TRIGGER after_user_update
    AFTER UPDATE ON users
    FOR EACH ROW
    EXECUTE PROCEDURE log_user_update();
```

В этом примере определяются два триггера, которые выполняются при обновлении строки таблицы `users`. Первый триггер выполняется до начала обновления, а второй — после того, как обновление завершится. Это означает, что события `BEFORE UPDATE` и `AFTER UPDATE` связаны причинно-следственной связью и не могут обрабатываться конкурентно.

С другой стороны, похожие наборы событий, происходящих с разными строками в разных сеансах, могут оставаться конкурентными, хотя это зависит от многих факторов (присутствие или отсутствие транзакций, уровень изоляции, область видимости триггеров и т. д.). Это «живой» пример ситуации, в которой изменение данных в СУБД может моделироваться событийной обработкой, хотя система в целом не является полностью асинхронной.

В следующем разделе рассматривается событийное программирование в графическом интерфейсе пользователя.

Событийное программирование в графическом интерфейсе пользователя

Графические интерфейсы (GUI) — это первое, что приходит на ум многим, кто слышит термин «событийное программирование». Событийное программирование — это элегантный механизм связывания пользовательского ввода и кода GUI, потому что оно естественным образом отражает взаимодействие пользователя с графическим интерфейсом. Такие интерфейсы часто предоставляют пользователю разнообразные компоненты, с которыми можно взаимодействовать, и это взаимодействие почти всегда нелинейно. В сложных интерфейсах оно часто моделируется набором событий, которые вызывает пользователь с помощью разных компонентов интерфейса.

Концепция событий присуща большинству библиотек и фреймворков пользовательского иптерфейса, хотя разные библиотеки реализуют событийное взаимодействие с помощью разных паттернов. Некоторые библиотеки даже онисывают свою архитектуру другими терминами (например, **сигиалы** в библиотеке Qt). Тем не менее общий принцип почти всегда один и тот же: все компоненты интерфейса (часто называемые **виджетами**) могут порождать события при взаимодействии. Другие компопенты получают эти события либо через механизм подписки, либо напрямую нрисоединяя себя к источникам событий в качестве обработчиков. В зависимости от библиотеки GUI события могут быть как нростыми именованными сигналами, которые сообщают о чем-то произошедшем (например, «пользователь щелкнул на виджете A»), так и более сложными сообщениями с дополнительной информацией о природе взаимодействия. Нанример, такие сообщения могут информировать, какая конкретно клавиша была нажата или в какой точке находился указатель мыши в момент события.

Мы рассмотрим различия между реальными паттернами проектирования позднее в разделе «Стили событийного программирования», а сейчас разберем пример GUI-приложения на языке Python, которое можно создать с номощью встроенного модуля `tkinter`.

Библиотека Tk, лежащая в основе модуля `tkinter`, обычно поставляется с дистрибутивами Python. Если она по какой-либо причине недоступна в вашей операционной системе, вы легко можете установить эту библиотеку из системного менеджера пакетов. Например, в дистрибутивах Linux на базе Debian ее можно установить как пакет `python3-tk` следующей командой:

```
sudo apt-get install python3-tk
```

Мы создадим GUI-приложение, которое отображает единственную кнопку с надписью Python Zen. Если щелкнуть па кпопке, приложение открывает новое окно с текстом документа «Дзен Python», импортированным из модуля `this` — «пасхалки» Python. После импортировапия в стандартный вывод выводятся 19 афоризмов, которые описывают ключевые принципы языка Python.

```
import this
from tkinter import Tk, Frame, Button, LEFT, messagebox

rot13 = str.maketrans(
    "ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz",
    "NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm"
)

def main_window(root: Tk):
    frame = Frame(root)
    frame.pack()

    zen_button = Button(frame, text="Python Zen", command=show_zen)
    zen_button.pack(side=LEFT)

def show_zen():
    messagebox.showinfo("Zen of Python", this.s.translate(rot13))

if __name__ == "__main__":
    root = Tk()
    main_window(root)
    root.mainloop()
```

В начале сценария импортируются нужные компонепты и определяется простая таблица преобразования строк. Она нужна потому, что текст документа «Дзеп Python» зашифрован в модуле `this` подстановочным шифром ROT13 (также называемым **шифром Цезаря**). Этот простой алгоритм шифрования заменяет каждую букву другой, отстоящей от нее в алфавите на 13 позиций.

Привязка событий происходит пепосредственно в конструкторе виджета `Button`:

```
Button(frame, text="Python Zen", command=show_zen)
```

Именованный аргумент `command` определяет обработчик события, который будет выполпяться по щелчку па кпопке. В нашем примере это функция `show_zen()`, которая выводит расшифрованный текст документа «Дзен Python» в отдельном окне.

У каждого виджета `tkinter` также есть метод `bind()`, с помощью которого можно определять обработчики конкретных событий — нажата/отпущена кнопка мыши, наведен указатель мыши и т. д.

Большинство GUI-фреймворков работают аналогичным образом: обычно разработчик не использует низкоуровневый ввод от клавиатуры и мыши, а связывает команды или функции обратного вызова с высокоуровневыми событиями, например:

- Изменилось состояние флажка.
- Пользователь щелкнул по кнопке.
- Выбран пункт меню.
- Закрыто окно.

В следующем разделе рассматривается механизм событийного взаимодействия.

Событийное взаимодействие

Событийное программирование — чрезвычайно распространенная практика разработки распределенных сетевых приложений. С событийным программированием проще разбить сложные системы на изолированные компоненты с ограниченным набором обязанностей, и поэтому оно особенно популярно в сервис-ориентированных и микросервисных архитектурах. В таких архитектурах поток событий проходит не между классами или функциями, существующими внутри одного вычислительного процесса, а между различными сетевыми службами. В крупных распределенных архитектурах поток событий между службами обычно координируется с помощью специальных коммуникационных протоколов (например, AMQP или ZeroMQ), часто с использованием особых служб, действующих как брокеры сообщений. Некоторые решения такого рода будут рассмотрены позднее в разделе «Событийные архитектуры».

Однако для того, чтобы ваш сетевой код считался событийным приложением, вам не понадобится ни формализованный механизм координации событий, ни выделенная служба их обработки. Вообще говоря, если более внимательно присмотреться к типичному веб-приложению на Python, вы заметите, что у большинства веб-фреймворков Python есть много общего с GUI-приложениями. Например, возьмем простое веб-приложение, написанное с использованием микрофреймворка Flask:

```
import this

from flask import Flask

app = Flask(__name__)

rot13 = str.maketrans(
    "ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz",
    "NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm"
)
```

```
def simple_html(body):
    return f"""
    <!DOCTYPE html>
    <html lang="en">
      <head>
        <meta charset="utf-8">
        <title>Book Example</title>
      </head>
      <body>
        {body}
      </body>
    </html>
    """


@app.route('/')
def hello():
    return simple_html("<a href=/zen>Python Zen</a>")

@app.route('/zen')
def zen():
    return simple_html(
        "<br>".join(this.s.translate(rot13).split("\n"))
    )

if __name__ == ,__main__':
    app.run()
```

Примеры написания и запуска простых приложений на Flask представлены в главе 2 «Современные среды разработки для Python».

Если сравнить этот код с примером приложения `tkinter` из предыдущего раздела, можно заметить, что структурно они очень похожи. Конкретные маршруты (пути) запросов HTTP преобразуются в специализированные обработчики. Если считать приложение событийным, то путь запроса можно рассматривать как привязку конкретного типа события (например, щелчка по ссылке) к обработчику действия. Подобно событиям в GUI-приложениях, запросы HTTP могут содержать дополнительные данные о контексте взаимодействия. И конечно, эта информация структурирована. Протокол HTTP определяет несколько методов запросов (например, POST, GET, PUT и DELETE) и несколько способов передачи дополнительных данных (строка запроса, тело запроса и заголовки).

Пользователь не взаимодействует с нашим приложением напрямую, как при работе с GUI; вместо этого он использует интерфейс браузера. В этом отношении приложение тоже отчасти похоже на традиционные графические приложения, ведь многие кросс-платформенные библиотеки пользовательского интерфейса

(такие, как Tcl/Tk, Qt и GTK+) на самом деле просто посредники между приложением и оконными API операционной системы пользователя. Таким образом, в обоих случаях мы имеем дело со взаимодействием и событиями, которые проходят через несколько системных уровней. Просто в веб-приложениях уровни более очевидны, а взаимодействие всегда выражено явно.

У современных веб-приложений часто бывают интерактивные интерфейсы на базе JavaScript. Многие из них разработаны с помощью событийных клиентских фреймворков, которые асинхронно взаимодействуют с серверной стороной приложения через серверные API. Это лишний раз подчеркивает событийную природу веб-приложений.

Итак, мы убедились, что в зависимости от сценария использования событийное программирование может применяться в приложениях разных типов. Оно также принимает разные формы. В следующем разделе мы рассмотрим три основных стиля событийного программирования.

Разные стили событийного программирования

Как упоминалось ранее, событийное программирование можно реализовать на разных уровнях программной архитектуры. Оно часто применяется в специфических областях разработки: сетевых коммуникациях, системном программировании, GUI-программировании и т. д. Таким образом, событийное программирование — это не единая целостная методология программирования, а скорее коллекция разнообразных паттернов, инструментов и алгоритмов, которые складываются в общую парадигму, где программирование опирается на поток событий.

Поэтому у событийного программирования существуют разные стили и разновидности. Конкретные реализации событийного программирования могут опираться на разные паттерны и приемы. Некоторые из этих приемов и инструментов даже не используют термин «событие». Несмотря на все разнообразие, можно выявить три основных стиля событийного программирования, которые лежат в основе более конкретных паттернов:

- **Стиль с обратными вызовами** (callback-based): привязка источников событий к их обработчикам по схеме «один к одному». В этом стиле источники событий сами указывают, какие действия должны происходить при возникновении того или иного события.
- **Стиль с субъектами** (subject-based): различные компоненты подписываются на события, порождаемые конкретными источниками, по схеме «один ко

многим». В этом стиле источники становятся **субъектами** подписки. Тот, кто хочет получать события, должен подписаться на их источник напрямую.

- **Стиль с топиками** (topic-based): центральное место занимают типы событий, а не их происхождение и место назначения. При таком стиле источники событий могут не знать о подписчиках, и наоборот. Вместо этого коммуникации осуществляются через независимые каналы событий (топики), в которых любая сторона может публиковать события или подписываться на них.

В ближайших разделах приводится краткий обзор трех основных стилей событийного программирования, с которыми вы можете столкнуться при программировании на Python.

Стиль с обратными вызовами

Стиль с обратными вызовами — один из самых распространенных стилей событийного программирования. В этом стиле объекты — источники событий отвечают за то, какие обработчики должны реагировать на эти события. Это подразумевает отношение «один к одному» или (максимум) «многие к одному» между источниками и обработчиками событий.

Этот стиль событийного программирования доминирует во фреймворках и библиотеках GUI. Причина проста: он действительно отражает то, как пользователи и программисты смотрят на пользовательские интерфейсы. Каждое действие — установка флажка, нажатие кнопки и т. д.— обычно выполняется с четкой и однозначной целью.

Мы уже рассмотрели пример событийного программирования, основанного на обратных вызовах, и написали простое графическое приложение с использованием библиотеки `tkinter` (см. раздел «Событийное программирование в графическом интерфейсе пользователя»). Напомним строку из этого приложения:

```
zen_button = Button(root, text="Python Zen", command=show_zen)
```

Созданный экземпляр класса `Button` определяет, что при каждом нажатии кнопки должна вызываться функция `show_zen()`. Здесь событие является неявным. Обратный вызов `show_zen()` (в `tkinter` обратные вызовы называются командами) не принимает никакого объекта, который описывал бы событие, связанное с вызовом. И это логично, потому что ответственность за привязку обработчиков событий лежит ближе к источнику событий — здесь это экземпляр `zen_button`. Обработчик события практически не интересуется его источником.

В некоторых реализациях событийного программирования, основанного на обратных вызовах, источники событий явно связываются с обработчиками на отдельном этапе, который может происходить после того, как источник иници-

ализирован. Этот стиль связывания возможен и в `tkinter`, но только для низкоуровневых событий взаимодействия с пользователем. Ниже приведен обновленный фрагмент приложения `tkinter`, где используется такой стиль:

```
def main_window(root):
    frame = Frame(root)

    zen_button = Button(frame, text="Python Zen")
    zen_button.bind("<ButtonRelease-1>", show_zen)
    zen_button.pack(side=LEFT)

def show_zen(event):
    messagebox.showinfo("Zen of Python", this.s.translate(rot13))
```

В этом примере событие уже не является неявным. Поэтому требуется, чтобы обратный вызов `show_zen()` мог принять объект `event`. Экземпляр `event` содержит основную информацию о взаимодействии с пользователем, в том числе позицию указателя мыши, время события и связанный с ним виджет. Важно помнить, что такая привязка событий все еще остается одноадресной. Это означает, что одному событию от одного объекта может соответствовать только один обратный вызов. Один обработчик можно связать с несколькими событиями и/или несколькими объектами, но одному событию, происходящему из одного источника, можно назначить только один обратный вызов. Если привязать новый обратный вызов методом `bind()`, то прежний вызов заменится новый.

Очевидные ограничения событийного программирования с обратными вызовами связаны с его одноадресной природой, потому что она требует жесткого зацепления компонентов приложения. К отдельному событию нельзя привязать несколько специализированных обработчиков, а это часто означает, что каждый обработчик вынужден обслуживать один-единственный источник и не может связываться с объектами другого типа.

Стиль с субъектами инвертирует связи между источниками и обработчиками событий. Он рассматривается в следующем разделе.

Стиль с субъектами

Стиль событийного программирования с субъектами — естественное расширение одноадресной обработки событий с обратными вызовами. В этом стиле источники, или генераторы событий (субъекты), позволяют другим объектам подписываться/регистрироваться для уведомлений о событиях субъекта. На практике это очень похоже на стиль с обратными вызовами, потому что обработчики событий обычно хранят списки функций или методов, которые должны вызываться, когда происходит новое событие.

В событийном программировании с субъектами фокус смещается от события к субъекту (источнику событий). Типичное воплощение этого стиля — паттерн проектирования Наблюдатель (Observer).

Если говорить кратко, паттерн Наблюдатель состоит из двух категорий объектов: наблюдателей и субъектов (иногда называемых «наблюдаемыми»). Экземпляр субъекта `Subject` поддерживает список экземпляров `Observer`, которые желают узнавать о событиях, происходящих в экземпляре `Subject`. Иначе говоря, `Subject` — источник событий, а экземпляры `Observer` — обработчики событий.

Если бы нам понадобилось определить общие интерфейсы для паттерна Наблюдатель, для этого можно было бы создать такие абстрактные базовые классы:

```
from abc import ABC, abstractmethod

class ObserverABC(ABC):
    @abstractmethod
    def notify(self, event): ...

class SubjectABC(ABC):
    @abstractmethod
    def register(self, observer: ObserverABC): ...
```

Экземпляры подклассов `ObserverABC` будут обработчиками событий. Их можно зарегистрировать в качестве наблюдателей событий субъекта с помощью метода `register()` экземпляров подкласса `SubjectABC`. В этой архитектуре интересно то, что она допускает многоадресные коммуникации между компонентами. Один и тот же наблюдатель может быть зарегистрирован у нескольких субъектов, и у одного субъекта может быть несколько подписчиков.

Более практичный пример поможет лучше понять потенциальные возможности этого механизма. Мы попытаемся построить наивную реализацию утилиты наподобие `grep`. Она будет рекурсивно искать в файловой системе файлы, содержащие заданный текст. Для рекурсивного обхода файловой системы мы используем модуль `glob`, а для поиска совпадений регулярных выражений — модуль `re`.

Основной код нашей программы будет размещаться в классе `Grepper`, который является подклассом `SubjectABC`. Начнем с определения базовой оснастки для регистрации и уведомления наблюдателей:

```
class Grepper(SubjectABC):
    _observers: list[ObserverABC]

    def __init__(self):
        self._observers = []

    def register(self, observer: ObserverABC):
        self._observers.append(observer)
```

```
    def notify_observers(self, event):
        for observer in self._observers:
            observer.notify(event)
```

Реализация относительно проста. Функция `__init__()` инициализирует пустой список наблюдателей. Каждый новый экземпляр `Grepper` создается без наблюдателей. Метод `register()` был определен в классе `SubjectABC` как абстрактный, поэтому мы обязаны предоставить его фактическую реализацию. Это единственный метод, который может добавлять новые наблюдатели в состояние субъекта. Наконец, метод `notify_observers()` передает указанное событие всем зарегистрированным наблюдателям.

Поскольку оснастка уже готова, теперь можно определить метод `Grepper.grep()`, который выполняет настоящую работу:

```
from glob import glob
import os.path
import re

class Grepper(SubjectABC):
    ...

    def grep(self, path: str, pattern: str):
        r = re.compile(pattern)

        for item in glob(path, recursive=True):
            if not os.path.isfile(item):
                continue

            try:
                with open(item) as f:
                    self.notify_observers(("opened", item))
                    if r.findall(f.read()):
                        self.notify_observers(("matched", item))
            finally:
                self.notify_observers(("closed", item))
```

Функция `glob(pattern, recursive=True)` выполняет рекурсивный поиск файловых путей, соответствующих шаблону. С ее помощью мы обходим файлы в каталоге, заданном пользователем. Для поиска по содержимому файлов используются регулярные выражения, поддержку которых предоставляет модуль `re`.

Поскольку сейчас возможные сценарии использования наблюдателей еще неизвестны, мы решили порождать события трех типов:

- `"opened"`: открыт новый файл.
- `"matched"`: `Grepper` нашел совпадение в файле.
- `"closed"`: файл закрыт.

Сохраним этот класс в файле с именем observers.py и завершим его следующим фрагментом кода, который инициализирует экземпляр класса Grepper входными аргументами:

```
import sys

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("использование: программа ПУТЬ ШАБЛОН")
        sys.exit(1)

    grepper = Grepper()
    grepper.grep(sys.argv[1], sys.argv[2])
```

Теперь программа observers.py может выполнять поиск в файлах, но еще не выдает никаких видимых результатов. Чтобы узнать, какая часть содержимого файла совнала с выражением, можно создать нодписчик, который будет реагировать на события "matched". Вот пример класса подписчика Presenter, который просто выводит имя файла, связанного с событием "matched":

```
class Presenter(ObserverABC):
    def notify(self, event):
        event_type, file = event
        if event_type == "matched":
            print(f"Найдено в: {file}")
```

А вот как этот подписчик можно связать с экземпляром класса Grepper:

```
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("использование: программа ПУТЬ ШАБЛОН")
        sys.exit(1)
    grepper = Grepper()
    grepper.register(Presenter())
    grepper.grep(sys.argv[1], sys.argv[2])
```

Чтобы узнать, в каком из примеров кода этой главы есть подстрока grep, можно запустить программу такой командой:

```
$ python observers.py 'Chapter 7/**' grep
Найдено в: Chapter 7/04 - Subject-based style/observers.py
```

Главное преимущество этого паттерна — расширяемость. Добавив новые наблюдатели, можно легко расширить функциональность приложения. Например, если вы хотите отслеживать все открытые файлы, можно создать специальный подписчик Auditor, который выводит информацию обо всех открытых и закрытых файлах. Его реализация может быть очень простой:

```
class Auditor(ObserverABC):
    def notify(self, event):
        event_type, file = event
        print(f"{event_type:8}: {file}")
```

Более того, у наблюдателей нет сильного связывания с субъектом, и они ограничиваются минимумом допущений о природе доставляемых событий. Если вы захотите использовать другой механизм поиска совпадений (например, модуль `fnmatch` с традиционными масками вместо регулярных выражений из модуля `re`), то сможете легко новторно использовать существующие наблюдатели, регистрируя их с совершенно новым классом субъекта.

Событийное программирование с субъектами ослабляет связывание между компонентами, а следовательно, улучшает модульность приложений. К сожалению, смещение фокуса с событий на субъекты может создать нроблемы. В нашем приложении наблюдатели будут оповещаться о каждом событии, которое породил класс `Subject`. У них пет возможпости подписаться на конкретпые типы событий; остается только фильтровать события подобно тому, как в нашем примере класс `Presenter` уже отфильтровывал события, отличные от `"matched"`.

Получается, что либо наблюдатель должен фильтровать все входные события, либо субъект должен позволять наблюдателям подписываться на конкретные события источника. Первый подход будет неэффективным, если подписчику придется отфильтровывать слишком много событий, а второй подход может неоправданно усложнить регистрацию наблюдателей и диспетчеризацию событий.

Несмотря на более детализированные обработчики и многоадресность, стиль событийного программирования на основе субъектов редко ослабляет связанность между компонентами приложения лучше, чем стиль с обратными вызовами. Поэтому этот вариант подходит скорее не для общей архитектуры больших приложений, а для конкретных узких задач.

Ориентация на субъекты ведет к тому, что всем обработчикам приходится поддерживать множество допущений о наблюдаемых субъектах. Кроме того, в реализации этого стиля (то есть паттерна Наблюдатель) как наблюдатели, так и субъекты должны в какой-то момент встретиться в одном контексте. Иными словами, наблюдатели не могут подписываться на события, если нет реального субъекта, который их порождает.

К счастью, существует стиль событийного программирования, который допускает детализированную многоадресную обработку событий таким образом, чтобы ноддерживать слабую связанность в больших приложениях. Это стиль, основанный на топиках, и он стал результатом естественной эволюции событийного программирования на основе субъектов.

Стиль с топиками

В событийном программировании с топиками события передаются между программными компонентами без жесткой привязки как к источнику, так и к обработчику. Это обобщение стилей, описанных ранее. Приложения событийного программирования, написанные в стиле с топиками, позволяют компонентам (например, классам, объектам и функциям) порождать события и/или подписываться на типы событий, полностью игнорируя другую сторону отношения «источник — обработчик».

Иначе говоря, можно регистрировать обработчики для типов событий, даже если не существует источника, который их порождает, а источники могут порождать события, даже если никто на них не подписался. В этом стиле событийного программирования события являются сущностями первого класса, которые могут определяться отдельно от источников и обработчиков. Для таких событий часто выделяется специализированный класс или просто глобальные одиночные экземпляры обобщенного класса `Event`. Именно это позволяет обработчикам поднисываться на события даже при отсутствии порождающего их объекта.

В разных фреймворках и библиотеках абстракции, которые инкапсулируют такие наблюдаемые тины/классы событий, могут называться по-разному. Часто встречаются термины «каналы» (channels), «топики» (topics) и «сигналы» (signals). Термин «сигнал» особенно распространен, из-за чего программирование в этом стиле часто называется **сигиальиым**. Сигналы встречаются в таких понулярных библиотеках и фреймворках, как Django (веб-фреймворк), Flask (веб-микрофреймворк), SQLAlchemy (ORM для баз данных) и Scrapy (фреймворк для поисковых ботов и автоматизированного сбора данных).

Успешные проекты Python обычно не строят свои сигнальные фреймворки с нуля, а иснользуют существующие специализированные библиотеки. Вероятно, самая популярная сигнальная библиотека Python — `blinker`. Она отличается чрезвычайно широкой совместимостью с версиями Python (Python 2.4 и выше, Python 3.0 и выше, Jython 2.5 и выше, PyPy 1.6 и выше), а также имеет очепь простой и лаконичный API, благодаря которому ее можно использовать почти в каждом проекте.

`blinker` онирается на концепцию именованных сигналов. Чтобы определить новый сигнал, вы просто используете копструктор `signal(name)`. Два разпых вызова конструктора `signal()` с одним значением `name` вернут один объект сигнала. Это позволяет легко обращаться к сигналам в любой момент времени. Вот нример класса `SelfWatch`, который с номощью именовапных сигпалов уведомляет свои экземпляры при каждом создании нового экземпляра:

```
import itertools

from blinker import signal

class SelfWatch:
    _new_id = itertools.count(1)

    def __init__(self):
        self._id = next(self._new_id)
        init_signal = signal("SelfWatch.init")
        init_signal.send(self)
        init_signal.connect(self.receiver)

    def receiver(self, sender):
        print(f"{self}: получено событие от {sender}")

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"
```

Сохраните этот код в файле topic_based_events.py. Следующий фрагмент интерактивного сеанса показывает, как новые экземпляры класса `SelfWatch` оповещают существующие экземпляры о своей инициализации:

```
>>> from topic_based_events import SelfWatch
>>> selfwatch1 = SelfWatch()
>>> selfwatch2 = SelfWatch()
<SelfWatch: 1>: получено событие от <SelfWatch: 2>
>>> selfwatch3 = SelfWatch()
<SelfWatch: 2>: получено событие от <SelfWatch: 3>
<SelfWatch: 1>: получено событие от <SelfWatch: 3>
>>> selfwatch4 = SelfWatch()
<SelfWatch: 2>: получено событие от <SelfWatch: 4>
<SelfWatch: 3>: получено событие от <SelfWatch: 4>
<SelfWatch: 1>: получено событие от <SelfWatch: 4>
```

Другие интересные возможности библиотеки `blinker`:

- **Анонимные сигналы**: пустой вызов `signal()` всегда создает абсолютно новый анонимный сигнал. Сохраняя сигнал как переменную модуля или атрибут класса, вы предотвратите опечатки в строковых литералах или случайные конфликты имен сигналов.
- **Подписка с учетом субъекта**: метод `signal.connect()` позволяет выбрать конкретного отправителя; благодаря этому можно использовать диспетчеризацию событий на базе субъектов поверх диспетчеризации на базе тем.
- **Декораторы сигналов**: `signal.connect()` можно использовать как декоратор; в результате объем кода сокращается, а обработка событий более явно выделяется в кодовой базе.

- **Данные в сигналах**: метод `signal.send()` получает произвольные именованные аргументы, которые передаются соответствующему обработчику; это позволяет использовать сигналы как механизм передачи сообщений.

У стиля событийпого программирования с топиками есть иптереспая особенность: он не навязывает субъектные отношения между компонентами. В зависимости от ситуации обе стороны отношения могут быть источниками и обработчиками событий друг для друга. Такой способ обработки событий превращается в обычный коммуникационный механизм. Из-за этого событийное программирование с топиками становится хорошим кандидатом для архитектурного паттерна.

Слабая связанность программных компонентов нозволяет вносить небольшие пошаговые изменения. Кроме того, если нроцесс приложения обладает слабой внутренней связанностью на основе системы событий, его можно легко разбить на несколько служб, которые взаимодействуют через очереди сообщений. Это позволяет нреобразовывать событийные нриложения в раснределенные событийные архитектуры.

Событийпые архитектуры рассматриваются в следующем разделе.

Событийные архитектуры

Всего один шаг отделяет событийные приложения от событийных архитектур (иначе называемых «архитектуры, управляемые событиями»). Событийное программирование позволяет разбить приложение на изолированные компоненты, которые взаимодействуют друг с другом только посредством обмена событиями или сигналами. Если это уже сделано, можно дальше разбить приложение на службы, которые делают все то же самое, передавая друг другу события через некий мехапизм **межпроцессных коммуннкацнй (IPC**, Inter-Process Communications) или по сети.

Событийные архитектуры переводят концепцию событийного программирования на уровепь коммуникаций между службами. Эти архитектуры заслуживают внимапия по многим причинам:

- **Масштабнруемость н эффектнвность нснользования ресурсов.** Если рабочую нагрузку можно разделить на множество событий, порядок которых пе важен, событийные архитектуры позволяют легко раснределить ее но разным вычислительным узлам (хостам). Вычислительную мощность также можно динамически регулировать в зависимости от количества событий, которые система обрабатывает в каждый конкретпый момент времени.
- **Слабая связанность.** В системах, состоящих из множества служб (желательно небольших), которые взаимодействуют друг с другом через очереди,

связанность обычно слабее, чем в монолитных системах. Это упрощает пошаговые изменения и стабильную эволюцию системных архитектур.

- **Отказоустойчивость.** Событийные системы с надлежащими технологиями доставки событий (распределенные очереди сообщений со встроенными средствами долгосрочного хранения сообщений) обычно оказываются более устойчивыми к врéменным проблемам. Современные очереди сообщений, такие как Kafka или RabbitMQ, предоставляют разные механизмы, дающие гарантию, что сообщение всегда будет доставлено хотя бы одному получателю и что оно будет доставлено повторпо в случае ненредвиденных ошибок.

Событийные архитектуры лучше всего подходят для задач, которые можно выполнять асинхронно (таких, как обработка файлов или доставка файлов/электронной почты), либо для систем с регулярными и/или планируемыми событиями (например, заданиями `cron`). В Python событийные архитектуры также нозволяют преодолеть ограничения быстродействия интернретатора CPython (такие, как GIL, рассмотренная в главе 6 «Конкурентное выполнение») за счет разбиения рабочей нагрузки по мпогим независимым процессам.

Наконец, событийные архитектуры по своей природе родственны бессерверным вычислениям. В этой модели облачных вычислений вас не интересует инфраструктура и не нужно приобретать блоки вычислительной мощности. За масштабирование и унравление инфраструктурой отвечает оператор облачной службы, а вы нредоставляете только выполняемый код. Часто расценки таких служб зависят только от ресурсов, которые использует ваш код. Среди бессерверных вычислительных служб особенно выделяется категория **FaaS** (Function as a Service, «функция как услуга»), которая выполняет небольшие блоки кода (функции) в ответ на события.

В следующем разделе более нодробно рассматриваются очереди событий и сообщений, которые лежат в основе большинства событийных архитектур.

Очереди событий и сообщений

В большинстве однопроцессных реализаций событийпого программировапия события обрабатываются сразу же после их возникновения, нричем, как правило, последовательно. Будь то стиль GUI-приложений с обратными вызовами или полноценпая передача сигналов в стиле библиотеки `blinker`, событийное приложепие обычно поддерживает ту или иную форму отображения между событиями и списками выполняемых обработчиков.

Такой стиль передачи информации в раснределенных приложениях обычно реализуется через коммуникационную модель «запрос — ответ». Это двусторонний — и очевидно синхронный — механизм коммуникации между службами. Он является хорошей основой для нростой обработки событий, но из-за своих

многочисленных недостатков неэффективен в крупномасштабных или сложных системах. Самая большая проблема схемы «запрос — ответ» заключается в том, что она создает относительно сильную связанность между компонентами:

- Все взаимодействующие компоненты должны уметь обнаруживать зависимые службы. Иначе говоря, источники событий должны знать сетевые адреса соответствующих обработчиков.
- Подписка происходит непосредственно в той службе, которая порождает событие. То есть чтобы создать совершенно новую событийную связь, обычно приходится изменять сразу несколько служб.
- Обе стороны взаимодействия должны согласовать коммуникационный протокол и формат сообщения. Из-за этого становится сложнее вносить изменения.
- Служба, порождающая события, должна обрабатывать потенциальные ошибки, которые возвращаются в ответах от зависимых служб.
- Коммуникации «запрос — ответ» часто не удается легко обрабатывать в асинхронной форме. Это означает, что событийная архитектура, построенная поверх системы «запрос — ответ», редко выигрывает от конкурентной обработки.

По перечисленным причинам событийные архитектуры обычно реализуются на основе очередей сообщений вместо циклов «запрос — ответ». Очередь сообщений — коммуникационный механизм, воплощенный в виде специализированной службы или библиотеки, который сосредоточен только на сообщениях и предполагаемом механизме их доставки. Очередь сообщений просто работает как коммуникационный центр для разных участников взаимодействия. Наоборот, схема «запрос — ответ» требует, чтобы обе взаимодействующие стороны знали друг о друге и были активны во время каждого сеанса обмена информацией.

Как правило, запись нового сообщения в очередь — это быстрая операция, потому что она не требует, чтобы на стороне подписчика немедленно выполнялось соответствующее действие (обратный вызов). Более того, источникам событий не требуется, чтобы их подписчики функционировали в момент возникновения нового сообщения, а асинхронная передача сообщений может улучшить отказоустойчивость. Наоборот, схема «запрос — ответ» предполагает, что зависимые службы постоянно доступны, а синхронная обработка событий может создавать большие задержки.

Очереди сообщений делают возможной слабую связанность служб, потому что они изолируют источники и обработчики событий друг от друга. Источники публикуют сообщения прямо в очередь, но им не нужно беспокоиться о том, прослушивает ли их другая служба. Аналогичным образом обработчики событий принимают сообщения прямо из очереди, и им не важно, кто породил события.

(Иногда информация об источнике события все-таки важна, но тогда она либо кодируется в содержимом передаваемого сообщения, либо участвует в механизме маршрутизации сообщений.) В такой коммуникационной схеме нет прямого синхронного соединения между источниками и обработчиками событий, и весь обмен информацией осуществляется через очередь.

В некоторых ситуациях слабая связанность доходит до такой крайности, что служба может взаимодействовать сама с собой через механизм внешней очереди. Этому не стоит удивляться, потому что очередь сообщений уже является отличным механизмом межпоточной коммуникации, который позволяет обойтись без блокировок (см. главу 6).

Кроме слабой связанности, очереди сообщений (особенно в форме специализированных служб) предоставляют ряд дополнительных возможностей:

- **Долгосрочное хранение.** Многие очереди обеспечивают долгосрочное хранение сообщений. Это означает, что даже если сама очередь перестанет работать, сообщения не потеряются.
- **Повторные попытки.** Многие очереди поддерживают подтверждение доставки и обработки сообщений и позволяют задать механизм повторных попыток для сообщений, которые не удалось доставить. Эта возможность в сочетании с долгосрочным хранением гарантирует, что если сообщение было успешно отправлено, то оно в конечном итоге будет обработано даже при временных сбоях сети или отказах службы.
- **Естественная конкурентность.** Очереди сообщений конкурентны по своей природе. С разными схемами распределения сообщений (например, веерная рассылка или циклическая диспетчеризация) очереди становятся отличной основой для распределенных архитектур с высокой масштабируемостью.

В фактической реализации очередей сообщений преобладают две основные архитектуры:

- **Очереди сообщений с брокером.** В этой архитектуре существует одна служба (или кластер служб), которая отвечает за прием и распределение событий. Самые известные примеры таких систем, распространяемых с открытым кодом, — RabbitMQ и Apache Kafka. Также популярна облачная служба Amazon SQS. Самые сильные стороны очередей с брокером — долгосрочное хранение сообщений и встроенная логика их доставки.
- **Очереди сообщений без брокера.** Эти системы реализуются исключительно как программные библиотеки. Пример популярной библиотеки сообщений без брокера — ZeroMQ (это название иногда пишется как ØMQ или zmq). Главное преимущество систем сообщений без брокера — эластичность. Они достигают простоты использования (не нужно поддерживать дополнительную централизованную службу или кластер служб), пожертвовав функцио-

нальностью и сложностью (такие аспекты, как долгосрочное хранение и сложные механизмы доставки, приходится реализовывать внутри служб).

У каждой разновидности очередей сообщений есть свои достоинства и недостатки. В очередях с брокером всегда приходится поддерживать дополнительную службу, если очереди с открытым кодом работают на собственной инфраструктуре, или оплачивать дополнительные услуги провайдера облачной службы. Такие системы обмена сообщениями быстро становятся критической частью архитектуры. Если такая центральная очередь сообщений проектируется без учета высокой доступности, она быстро может стать единой точкой отказа для архитектуры всей системы. Впрочем, современные системы очередей обычно богато оснащены готовой функциональностью, а их интеграция в код часто сводится к тому, чтобы правильно настроить конфигурацию или наладить несколько вызовов API. Со стандартом AMQP можно также относительно легко запускать локальные очереди для тестирования.

В очередях сообщений без брокера коммуникации обычно становятся более распределенными. Это означает, что системная архитектура не зависит от какой-либо одной службы или от одного кластера передачи сообщений. Даже если некоторые службы не работают, оставшаяся часть системы все еще может обмениваться данными. Недостаток такого подхода заключается в том, что вам обычно приходится самостоятельно решать такие проблемы, как долгосрочное хранение сообщений, подтверждения доставки и обработки, повторные попытки доставки и обработка сложных сетевых сбоев (например, расщепление сети). Если возникнет такая необходимость, вам придется либо реализовать соответствующую функциональность непосредственно в своих службах, либо с нуля разработать собственный брокер сообщений на базе библиотек передачи сообщений без брокера. В больших распределенных приложениях обычно лучше использовать проверенные и хорошо зарекомендовавшие себя реализации с брокером.

Событийные архитектуры способствуют модульности и декомпозиции больших приложений на меньшие службы. У этого факта есть как достоинства, так и недостатки. Когда множество компонентов взаимодействуют через очереди, становится труднее отлаживать приложения и разбираться в логике их работы.

С другой стороны, хорошие приемы системной архитектуры, такие как разделение обязанностей, изоляция областей и формальные контракты коммуникации, улучшают общую архитектуру и упрощают разработку отдельных компонентов.

Примеры стандартов, посвященных созданию формальных контрактов коммуникации, — OpenAPI и AsyncAPI. Это языки на базе YAML, предназначенные для определения спецификаций коммуникационных протоколов и схем приложений. О них можно больше узнать по адресам https://swagger.io/specification/ и https://www.asyncapi.com.

Итоги

В этой главе рассматривались элементы событийного программирования. Мы начали с типичных примеров и простых практических приложений, которые послужили введением в эту парадигму. Затем были описаны три основных стиля событийного программирования: с обратными вызовами, с субъектами и с топиками. С событийным программированием связано множество паттернов проектирования и приемов написания кода, но все они относятся к какой-либо из этих трех категорий. Последняя часть этой главы была посвящена событийным программным архитектурам.

Эта глава завершает ту часть книги, которую можно было бы назвать «сюжетной линией архитектуры и проектирования». В дальнейшем мы будем меньше говорить об архитектуре, паттернах проектирования и парадигмах, а больше — о внутреннем устройстве и расширенных возможностях синтаксиса Python.

Следующая глава посвящена метапрограммированию на Python, то есть написанию программ, которые могут интерпретировать себя как данные, анализировать себя и изменять себя во время выполнения.

8

Элементы метапрограммирования

Метапрограммироваиием называется совокунность нриемов, которые относятся к способности программ анализировать себя, нонимать собственный код и изменять себя. Такой подход обеснечивает донолнительную мощь и гибкость. Без метанрограммирования у нас, вероятно, не было бы современных фреймворков или, по крайней мере, эти фреймворки были бы гораздо менее выразительными.

Термин «метапрограммирование» часто окружен ореолом таинственности. Многие разработчики ассоциируют его почти исключительно с программами, которые могут анализировать и модифицировать свой код на уровне исходного текста. Программы, которые манипулируют собственным исходным кодом, — это, безусловно, один из самых заметных и сложпых примеров прикладного метапрограммирования, но оно нроявляется также во многих других формах и не всегда отличается сложностью или трудоемкостью. Язык Python особенно богат функциями и модулями, благодаря которым некоторые нриемы метапрограммирования становятся простыми и естественными.

В этой главе мы объясним, что собой представляет метапрограммирование, и рассмотрим ряд практических нодходов к нему в Python. Начнем с простых средств метанрограммирования (таких, как декораторы функций и классов), а также обсудим расширенные механизмы, которые нозволяют переопределять процесс создания экземпляров класса и использовать метаклассы. В конце главы последуют примеры самого мощного, но нри этом и самого опасного нодхода к метапрограммированию — шаблоны генерации кода.

В этой главе рассматриваются следующие темы:

- Что такое метапрограммирование?
- Как использовать декораторы, чтобы изменять поведение функции перед вызовом.
- Вмешательство в процесс создания экземпляра класса.
- Метаклассы.
- Генерация кода.

Но прежде чем рассматривать средства метапрограммирования, доступные для разработчиков Python, начнем с технических требований.

Технические требования

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `inflection`
- `macropy3`
- `falcon`
- `hy`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%208.

Что такое метапрограммирование?

Возможно, нам удалось бы подобрать хорошее академическое определение метапрограммирования, однако эта книга посвящена не теории computer science, а тому, как профессионально разрабатывать ПО. Поэтому мы приведем свое неформальное определение:

> *Метапрограммирование — методология написания программ, которые могут рассматривать себя как данные, чтобы анализировать, генерировать и/или изменять себя во время выполнения.*

Это определение позволяет различать две основные ветви метапрограммирования в Python:

- **Метапрограммирование, ориентированное на интроспекцию**: сосредоточено на естественных возможностях интроспекции языка и динамических определениях функций и типов.
- **Метапрограммирование, ориентированное на код**: сосредоточено на манипулировании участками кода как изменяемыми структурами данных.

Метапрограммирование, ориентированное на интроспекцию, в первую очередь касается возможностей программно анализировать базовые элементы языка (функции, классы и типы), создавать и изменять их на ходу. Python предоставляет великое множество инструментов в этой области. Эту функциональность языка Python часто используют интегрированные среды (IDE), чтобы анализировать код в реальном времени и предлагать имена. Простейшие средства метапрограммирования в Python, которые используют интроспекцию, — это декораторы, с помощью которых можно добавлять новую функциональность к существующим функциям, методам и классам. Затем идут специальные методы классов, которые позволяют вмешаться в процесс создания экземпляров класса. Наконец, самый мощный инструмент — метаклассы, с помощью которых можно хоть полностью перестроить реализацию объектно-ориентированного программирования в Python.

Метапрограммирование, ориентированное на код, позволяет программистам работать непосредственно с кодом — либо в низкоуровневом формате (простой текст), либо в более доступной для программирования форме абстрактного синтаксического дерева (AST). Конечно, со вторым вариантом сложнее управляться, зато он открывает выдающиеся возможности, такие как расширение синтаксиса языка Python и даже создание ваших собственных предметно-ориентированных языков (DSL, Domain-Specific Language).

В следующем разделе будут рассматриваться декораторы в контексте метапрограммирования.

Как использовать декораторы, чтобы изменить поведение функции перед вызовом

Декораторы — один из самых популярных инструментов интроспективно-ориентированного метапрограммирования в Python. Так как функции в языке Python являются объектами первого класса, их можно анализировать и изменять во время выполнения. Декораторы — это специальные функции, которые способны анализировать, изменять и оборачивать другие функции.

Синтаксис декораторов разъяснялся в главе 4 «Python в сравнении с другими языками». Собственно, это «синтаксический сахар», предназначенный для того,

чтобы упростить работу с функциями, которые расширяют существующие кодовые объекты дополнительным поведением.

Код с простым синтаксисом декоратора может выглядеть так:

```
@some_decorator
def decorated_function():
    pass
```

Его также можно записать другим, более развернутым способом:

```
def decorated_function():
    pass
decorated_function = some_decorator(decorated_function)
```

Развернутая форма ясно показывает, что делает декоратор. Он принимает объект функции и изменяет его во время выполнения. Декоратор обычно возвращает новый объект функции, который теперь фигурирует под прежним именем декорируемой функции.

Как было показано в главе 5 «Интерфейсы, паттерны и модульность», декораторы функций незаменимы при реализации многих паттернов проектирования. Благодаря декораторам часто удается перехватывать и предварительно обрабатывать исходные аргументы функций, изменять возвращаемые значения или расширять контекст вызова функции такими дополнительными функциональными аспектами, как журналирование, профилирование или проверка статуса авторизации/аутентификации вызывающей стороны.

Рассмотрим пример использования декоратора `@lru_cache` из модуля `functools`:

```
from functools import lru_cache

@lru_cache(size=100)
def expensive(*args, **kwargs):
    ...
```

Декоратор `@lru_cache` создает кэш возвращаемых значений заданной функции, работающий по принципу LRU (Least Recently Used, «вытеснение давно не использовавшихся элементов»). Он перехватывает входные аргументы функции и сравнивает их со списком недавно использованных наборов аргументов. Если обнаружится совпадение, то вместо вызова декорируемой функции возвращается кэшированное значение. Если совпадений нет, то сначала вызывается исходная функция, а затем возвращенное значение сохраняется в кэше для использования в будущем. В нашем примере кэш может содержать не более 100 элементов.

Здесь интересно то, что использование `@lru_cache` — это уже элемент метапрограммирования. Декоратор получает существующий кодовый объект (в данном

случае функцию `expensive()`) и изменяет его поведение. Он также перехватывает аргументы и проверяет их значения и типы, чтобы решить, нужно ли их кэшировать.

Это хороший знак. Как было показано в главе 4 «Python в сравнении с другими языками», декораторы в Python пишутся и используются относительно просто. Во многих случаях код с декораторами становится короче, проще читается, а его сопровождение обходится дешевле. Таким образом, декораторы функций идеально подходят, чтобы познакомиться с метапрограммированием. Другие инструменты метапрограммирования, доступные в Python, труднее понять и освоить.

Следующий естественный этап после декораторов функций — декораторы классов, которые рассматриваются в ближайшем разделе.

Следующий этап: декораторы классов

Одна из менее известных синтаксических возможностей Python — декораторы классов. По синтаксису и реализации они не отличаются от декораторов функций. Разница только в том, что они возвращают классы вместо объектов функций.

Декораторы классов уже встречались в предыдущих главах. Так, декоратор `@dataclass` из модуля `dataclasses` рассматривался в главе 4 «Python в сравнении с другими языками», а декоратор `@runtime_checkable` из модуля `typing` — в главе 5 «Интерфейсы, паттерны и модульность». Оба декоратора опираются на механизмы интроспекции Python, чтобы расширять существующие классы дополнительным поведением:

- Декоратор `@dataclass` проверяет аннотации атрибутов класса, чтобы создать реализацию по умолчанию метода `__init__()` и протокол сравнения, что избавляет разработчиков от необходимости писать однообразный шаблонный код. Он также позволяет писать «фиксированные» классы с неизменяемыми и хешируемыми экземплярами, которые можно использовать как ключи словаря.
- Декоратор `@runtime_checkable` помечает подклассы `Protocol` как «проверяемые на стадии выполнения». Это означает, что по аннотации аргументов и возвращаемого значения подкласса `Protocol` можно во время выполнения определить, реализует ли другой класс интерфейс, определенный классом протокола.

Лучший способ понять, как работают декораторы классов, — применить их на практике. Декораторы `@dataclass` и `@runtime_checkable` устроены довольно сложно, так что мы не будем рассматривать их код, а попытаемся создать собственный простой пример.

Одна из замечательных особенностей классов данных — возможность предоставить реализацию метода `__repr__()` по умолчанию. Этот метод возвращает строковое представление объекта, которое может отображаться в интерактивном сеансе, журналах или стандартном выводе.

Для пользовательских классов метод `__repr__()` по умолчанию возвращает только имя класса и адрес памяти, но для классов данных он автоматически включает представление каждого отдельного поля класса. Мы попробуем создать декоратор класса данных, который предоставит похожую функциональность для любого класса.

Для начала напишем функцию, которая возвращает человекочитаемое представление любого экземпляра класса, если передать ей список атрибутов для представления:

```
from typing import Any, Iterable

UNSET = object()

def repr_instance(instance: object, attrs: Iterable[str]) -> str:
    attr_values: dict[str, Any] = {
        attr: getattr(instance, attr, UNSET)
        for attr in attrs
    }
    sub_repr = ", ".join(
        f"{attr}={repr(val) if val is not UNSET else 'UNSET'}"
        for attr, val in attr_values.items()
    )
    return f"<{instance.__class__.__qualname__}: {sub_repr}>"
```

Наша функция `repr_instance()` сначала обходит атрибуты экземпляра с помощью функции `getattr()`, которая перебирает имена атрибутов, переданных в аргументе `attrs`. Некоторые атрибуты экземпляров могут быть не заданы в момент создания представления. Если атрибут не задан, функция `getattr()` вернет `None`. Однако `None` — тоже допустимое значение атрибута, так что нам понадобится способ отличать незаданные атрибуты от `None`. Именно поэтому мы используем сигнальную метку `UNSET`.

`UNSET = object()` — распространенный паттерн для создания уникальных сигнальных меток. Минимальный экземпляр типа `object` возвращает `True` при проверке оператором `is`, только если сравнивается сам с собой.

Когда атрибуты и их значения известны, наша функция с помощью f-строк создает фактическое представление экземпляра класса, в которое входят представления каждого отдельного атрибута, определенного в аргументе `attrs`.

Вскоре мы покажем, как автоматически включать такие представления в собственные классы, но сначала разберемся, как они работают с существующими объектами. Например, в следующем примере функция `instance_repr()` запускается в интерактивном сеансе, чтобы получить представление комплексного числа:

```
>>> repr_instance(1+10j, ["real", "imag"])
'<complex: real=1.0, imag=10.0>'
```

Это выглядит неплохо, но приходится явно передавать экземпляр объекта и знать все возможные имена атрибутов, прежде чем их можно будет вывести. Это не очень удобно, потому что нужно обновлять аргументы `repr_instance()` при каждом изменении структуры класса. Мы напишем декоратор класса, который сможет внедрить функцию `repr_instance()` в декорируемый класс. Мы также будем использовать аннотации атрибутов класса, хранящиеся в его атрибуте `__annotations__`, чтобы определять, какие атрибуты войдут в представление. Код декоратора выглядит так:

```
def autorepr(cls):
    attrs = set.union(
        *(
            set(c.__annotations__.keys())
            for c in cls.mro()
            if hasattr(c, "__annotations__")
        )
    )

    def __repr__(self):
        return repr_instance(self, sorted(attrs))

    cls.__repr__ = __repr__
    return cls
```

В этих нескольких строках используется многое из того, о чем вы узнали в главе 4 «Python в сравнении с другими языками». Сначала мы получаем список атрибутов с аннотациями из словаря `cls.__annotations__` каждого класса в порядке MRO. Нам приходится обходить весь MRO, потому что аннотации не наследуются от базовых классов.

Затем мы используем замыкание, чтобы определить внутреннюю функцию `__repr__()`, у которой есть доступ к переменной `attrs` из внешней области видимости. Когда эта функция готова, мы переопределяем существующий метод `cls.__repr__()` новой реализацией. Это можно сделать, потому что объекты функций уже являются дескрипторами без данных. Это означает, что в контексте класса они становятся методами и просто принимают объект экземпляра в первом аргументе.

Теперь можно протестировать декоратор с каким-нибудь экземпляром нашего собственного класса. Сохраним код в файле autorepr.py и определим тривиальный класс с аннотациями атрибутов, который будет снабжен декоратором `@autorepr`:

```
from typing import Any

@autorepr
class MyClass:
    attr_a: Any
    attr_b: Any
    attr_c: Any

    def __init__(self, a, b):
        self.attr_a = a
        self.attr_b = b
```

Вероятно, бдительные читатели заметили, что мы не стали инициализировать атрибут `attr_c`. Это сделано намеренно, чтобы продемонстрировать, как `@autorepr` обращается с незаданными атрибутами. Запустите Python, импортируйте класс и посмотрите на автоматически сгенерированные представления:

```
>>> from autorepr import MyClass
>>> MyClass("Ultimate answer", 42)
<MyClass: attr_a='Ultimate answer', attr_b=42, attr_c=UNSET>
>>> MyClass([1, 2, 3], ["a", "b", "c"])
<MyClass: attr_a=[1, 2, 3], attr_b=['a', 'b', 'c'], attr_c=UNSET>
>>> instance = MyClass(None, None)
>>> instance.attr_c = None
>>> instance
<MyClass: attr_a=None, attr_b=None, attr_c=None>
```

Приведенный пример из интерактивного сеанса Python показывает, как декоратор `@autorepr` может использовать аннотации атрибутов класса, чтобы обнаруживать поля, которые надо включить в представление экземпляра. Также он умеет отличать незаданные атрибуты от тех, которым было явно присвоено значение `None`. Декоратор пригоден для повторного использования, что позволяет применить его к любому классу с аннотациями типов для атрибутов вместо того, чтобы вручную создавать новые методы `__repr__()`.

Более того, декоратор не нужно постоянно сопровождать. Если вы расширите класс дополнительной аннотацией атрибута, этот атрибут будет автоматически включен в представление экземпляра.

Модификация существующих классов на месте (также называемая горячей подменой или monkey patching) — стандартный прием, используемый в декораторах классов. Еще один способ расширения существующих классов декораторами заключается в том, чтобы создавать новые подклассы «на ходу» с по-

мощью замыканий. Если бы нам пришлось переписать пример в форме шаблона подклассирования, это могло бы выглядеть так:

```
def autorepr(cls):
    attrs = cls.__annotations__.keys()

    class Klass(cls):
        def __repr__(self):
            return repr_instance(self, attrs)

    return Klass
```

Главный недостаток подобного использования замыканий в декораторах классов заключается в том, что этот механизм влияет на иерархию классов. Среди прочего переопределяются атрибуты класса `__name__`, `__qualname__` и `__doc__`. В нашем случае это также будет означать, что потеряется часть предполагаемой функциональности. Ниже приведены потенциальные представления `MyClass`, снабженные таким декоратором:

```
<autorepr.<locals>.Klass: attr_a='Ultimate answer', attr_b=42, attr_
c=UNSET>
<autorepr.<locals>.Klass: attr_a=[1, 2, 3], attr_b=['a', 'b', 'c'],
attr_c=UNSET>
```

Легко решить эту проблему не получится. В модуле `functools` есть вспомогательный декоратор `@wraps`, который можно использовать в обычных декораторах функций, чтобы сохранить метаданные аннотированной функции. К сожалению, его нельзя использовать с декораторами классов, так что возможности создания подклассов в декораторах классов ограниченны. Например, подклассы могут испортить результаты работы инструментов, которые автоматически генерируют документацию.

И все же, несмотря на этот недостаток, декораторы классов являются простой альтернативой популярному паттерну примесей. **Примесь** (mixin) в Python — это класс, который не предназначен для создания экземпляров, а предоставляет повторно используемый API или иную функциональность для других существующих классов. Классы-примеси почти всегда добавляются посредством множественного наследования и обычно используются в такой форме:

```
class SomeConcreteClass(MixinClass, SomeBaseClass):
    pass
```

Примеси составляют полезный паттерн проектирования, который используется во многих библиотеках и фреймворках. В частности, они широко применяются во фреймворке Django. Будучи, безусловно, полезными и популярными, при неправильном проектировании примеси могут создать проблемы, потому

что в большинстве случаев они заставляют разработчика полагаться на множественное наследование. Как упоминалось ранее, Python относительно неплохо справляется со множественным наследованием благодаря четкой реализации MRO. Тем не менее лучше избегать множественного наследования, потому что оно усложняет код и с ним трудно работать. Декораторы классов могут стать хорошей заменой примесям.

В общем случае задача декораторов — изменять поведение функций и классов до их фактического использования. Декораторы функций заменяют существующие функции их инкапсулированными альтернативами, а декораторы классов обычно модифицируют определение класса. Однако некоторые приемы метапрограммирования позволяют изменять поведение кода прямо во время его использования. Один из таких приемов основан на вмешательстве в процесс создания экземпляра класса, для чего переопределяется метод `__new__()`. Эта возможность рассматривается в следующем разделе.

Вмешательство в процесс создания экземпляра класса

Существуют два специальных метода, относящихся к созданию экземпляра класса и процессу инициализации: `__init__()` и `__new__()`.

Метод `__init__()` концептуально ближе всего к конструкторам, которые встречаются во многих объектно-ориентированных языках. Он получает свежий экземпляр класса с аргументами инициализации и инициализирует состояние экземпляра.

Снециальный метод `__new__()` — статический метод, который непосредственно отвечает за создание экземпляра класса. Метод `__new__(cls, [,...])` вызывается до метода инициализации `__init__()`. Как нравило, реализация переопределенного метода `__new__()` вызывает свою версию из надкласса `super().__new__()` с нодходящими аргументами и модифицирует экземпляр, прежде чем возвращать его.

Метод `__new__()` — особый статический метод, поэтому его не нужно объявлять статическим с помощью декоратора `staticmethod`.

Рассмотрим пример класса, где реализация метода `__new__()` переопределяется, чтобы подсчитывать количество экземпляров класса:

```
class InstanceCountingClass:
    created = 0
```

```
    number: int

    def __new__(cls, *args, **kwargs):
        instance = super().__new__(cls)
        instance.number = cls.created
        cls.created += 1

        return instance

    def __repr__(self):
        return (
            f"<{self.__class__.__name__}: "
            f"{self.number} of {self.created}>"
        )
```

Следующий фрагмент интерактивного сеанса демонстрирует, как работает наша реализация `InstanceCountingClass`:

```
>>> instances = [InstanceCountingClass() for _ in range(5)]
>>> for i in instances:
...     print(i)
...
<InstanceCountingClass: 0 of 5>
<InstanceCountingClass: 1 of 5>
<InstanceCountingClass: 2 of 5>
<InstanceCountingClass: 3 of 5>
<InstanceCountingClass: 4 of 5>
>>> InstanceCountingClass.created
5
```

Метод `__new__()` обычно возвращает экземнляр того класса, к которому он относится, но может возвращать и экземпляры других классов. Когда `__new__()` возвращает экземпляр другого класса, то вызов метода `__init__()` пропускается. Это может пригодиться, если возникает необходимость вмешаться в процесс создания или инициализации экземпляров неизменяемых классов (например, некоторых встроенных классов Python).

Вот пример создания подкласса типа `int`, в который не входит нулевое значение:

```
class NonZero(int):
    def __new__(cls, value):
        return super().__new__(cls, value) if value != 0 else None

    def __init__(self, skipped_value):
        # Реализацию __init__ в данном случае можно опустить,
        # но мы оставили ее, чтобы продемонстрировать,
        # что она не вызывается
        print("__init__() called")
        super().__init__()
```

В приведенный пример включено несколько команд `print`, чтобы показать, как Python может пропускать вызов `__init__()` в некоторых ситуациях. Следующий интерактивный сеанс демонстрирует этот эффект:

```
>>> type(NonZero(-12))
__init__() called
<class '__main__.NonZero'>
>>> type(NonZero(0))
<class 'NoneType'>
>>> NonZero(-3.123)
__init__() called
-3
```

Когда же следует использовать `__new__()`? Ответ прост: только тогда, когда вызова `__init__()` недостаточно. Один пример уже упоминался ранее — создание подклассов неизменяемых встроенных типов Python (таких, как `int`, `str`, `float`, `frozenset` и т. д.). Здесь дело в том, что после того, как экземпляр неизменяемого объекта создан, его уже невозможно модифицировать в методе `__init__()`.

Некоторые программисты возразят, что вызов `__new__()` также пригодится для инициализации важных объектов, которая будет пропущена, если пользователь забудет включить вызов `super().__init__()` в переопределенный метод инициализации. И хотя это звучит разумно, у такого решения есть серьезный недостаток: в тех случаях, где, наоборот, потребуется явно пропускать предыдущие этапы инициализации, программисту будет труднее это обеспечить. Также здесь нарушается неписаное правило о том, чтобы все инициализации выполнялись в `__init__()`.

Фабричные методы в Python обычно определяются с помощью декоратора `classmethod`, который может перехватывать аргументы перед вызовом конструктора класса. Как правило, это позволяет упаковать сразу несколько вариантов логики инициализации в один класс. Ниже приведен пример подкласса типа `list`, у которого есть два фабричных метода для создания экземпляров списка удвоенного и утроенного размера:

```
from collections import UserList

class XList(UserList):
    @classmethod
    def double(cls, iterable):
        return cls(iterable) * 2

    @classmethod
    def triple(cls, iterable):
        return cls(iterable) * 3
```

Так как `__new__()` не обязательно должен возвращать экземпляр того же класса, этот метод легко становится предметом злоупотреблений. Его безответственное использование может сильно ухудшить удобочитаемость кода, поэтому манипулировать методом `__new__()` следует осторожно, сопровождая код обширной документацией. Обычно для каждой конкретной задачи лучше поискать другие решения, чем модифицировать процедуру создания объекта таким образом, что результат будет противоречить стандартным ожиданиям программистов. Даже переопределенную инициализацию неизменяемых типов можно заменить более предсказуемыми и привычными паттернами проектирования, например фабричными методами.

В программировании на Python есть по крайней мере один аспект, в котором активное применение метода `__new__()` действительно оправданно. Речь идет о метаклассах, описанных в следующем разделе.

Метаклассы

Многие разработчики считают метаклассы Python одним из самых трудных для понимания аспектов языка и поэтому стараются их избегать. На самом деле все не так сложно, как кажется: стоит лишь понять несколько базовых концепций.

Зато если вы умеете пользоваться метаклассами, то можете делать многие вещи, которые без них невозможны.

Метакласс — это тип (класс), который определяет другие типы (классы). Самое важное, что необходимо знать об их устройстве, — что классы (то есть типы, которые определяют структуру и поведение объектов) тоже являются объектами. А поскольку это объекты, они должны быть связаны с некоторым классом. Базовым типом каждого определения класса является встроенный класс `type` (рис. 8.1).

Рис. 8.1. Как устроены типы классов

В Python метакласс объекта класса можно заменить вашим собственным типом. Обычно новый метакласс остается подклассом типа `metaclass` (рис. 8.2), потому что в противном случае полученные классы окажутся несовместимыми с другими классами в отношении наследования.

В следующем разделе рассматривается общий синтаксис метаклассов.

Общий синтаксис

Вызов встроенной функции `type()` можно использовать как динамический эквивалент инструкции `class`. Вот пример определения класса с вызовом `type()`:

```
def method(self):
    return 1

MyClass = type('MyClass', (), {'method': method})
```

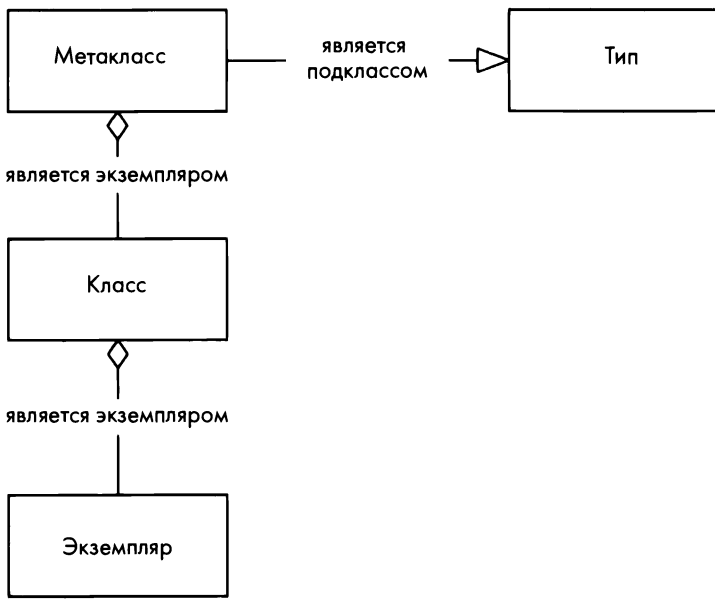

Рис. 8.2. Обычная реализация собственных метаклассов

В первом аргументе передается имя класса, во втором — список базовых классов (в данном случае пустой кортеж), а в третьем — словарь атрибутов классов (обычно методов). Эта конструкция эквивалентна явному определению класса с ключевым словом `class`:

```
class MyClass:
    def method(self):
        return 1
```

Каждый класс, созданный инструкцией `class`, неявно использует `type` как свой метакласс. Чтобы изменить это поведение по умолчанию, передайте инструкции `class` именованный аргумент `metaclass`:

```
class ClassWithAMetaclass(metaclass=type):
    pass
```

В аргументе `metaclass` обычно передается другой объект класса, но это может быть любой вызываемый объект, который принимает те же аргументы, что и класс `type`, и возвращает другой объект класса.

Развернутая сигнатура вызова метакласса имеет вид `type(name, bases, namespace)`. Назначение аргументов таково:

- `name`: имя класса, которое будет храниться в атрибуте `__name__`.
- `bases`: список родительских классов, который станет атрибутом `__bases__` и будет использоваться, чтобы сконструировать MRO созданного класса.
- `namespace`: пространство имен (отображение) с определениями для тела класса, которое станет атрибутом `__dict__`.

Метаклассы можно рассматривать как метод `__new__()`, но на более высоком уровне определения класса.

Несмотря на то что вместо метаклассов можно использовать функции, которые явно вызывают `type()`, обычно для этой цели используется другой класс, наследующий классу `type`. Стандартный шаблон метакласса выглядит так:

```
class Metaclass(type):
    def __new__(mcs, name, bases, namespace):
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        return super().__call__(*args, **kwargs)
```

Аргументы `name`, `bases` и `namespace` имеют такой же смысл, как для рассмотренного ранее вызова `type()`, но четыре метода вызываются на разных стадиях жизненного цикла класса:

- `__new__(mcs, name, bases, namespace)`: отвечает за непосредственное создание объекта класса точно так же, как в случае обычных классов. Первый пози-

ционный аргумент содержит объект метакласса. В предыдущем примере это был бы `Metaclass`. Этому аргументу принято присваивать имя `mcs`.

- `__prepare__(mcs, name, bases, **kwargs)`: создает пустой объект пространства имен. По умолчанию этот метод возвращает пустой экземпляр `dict`, но его можно переопределить, чтобы возвращать экземпляр любого другого подкласса `dict`. Обратите внимание, что метод не принимает аргумент `namespace`, потому что перед вызовом пространство имен еще не существует. Пример использования этого метода будет рассмотрен позднее в разделе «Использование метаклассов».
- `__init__(cls, name, bases, namespace, **kwargs)`: этот метод нечасто используется в реализациях метаклассов, однако имеет тот же смысл, что и в обычных классах. Он может выполнять дополнительную инициализацию объекта класса после того, как этот объект создан вызовом `__new__()`. Первый позиционный аргумент на этот раз принято называть `cls`, чтобы показать, что это уже созданный объект класса (экземпляр метакласса), а не объект метакласса. При вызове `__init__()` класс уже сконструирован, так что методу `__init__()` достается меньше работы, чем методу `__new__()`. Реализация такого метода очень похожа на использование декораторов классов, но главное отличие в том, что `__init__()` будет вызываться для каждого подкласса, тогда как декораторы классов не вызываются для подклассов.
- `__call__(cls, *args, **kwargs)`: вызывается при вызове экземпляра метакласса. Экземпляр метакласса представляет собой объект класса (см. рис. 8.1); он вызывается, когда создаются новые экземпляры класса. Это поведение можно использовать, чтобы переопределять способ, с помощью которого экземпляры класса создаются и инициализируются по умолчанию.

Каждый из перечисленных методов может принимать дополнительные именованные аргументы, представленные как `**kwargs`. Эти аргументы могут передаваться объекту `metaclass` с помощью дополнительных именованных аргументов в определении класса, как в следующем коде:

```
class Klass(metaclass=Metaclass, extra="value"):
    pass
```

Такой объем информации трудно с ходу усвоить без примеров, поэтому будем отслеживать создание метаклассов, классов и экземпляров с помощью вызовов `print()`:

```
class RevealingMeta(type):
    def __new__(mcs, name, bases, namespace, **kwargs):
        print(mcs, "METACLASS __new__ called")
        return super().__new__(mcs, name, bases, namespace)
```

```
    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print(mcs, " METACLASS __prepare__ called")
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        print(cls, " METACLASS __init__ called")
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        print(cls, " METACLASS __call__ called")
        return super().__call__(*args, **kwargs)
```

Если использовать `RevealingMeta` как метакласс при создании определения нового класса, вы получите следующий вывод в интерактивном сеансе Python:

```
>>> class RevealingClass(metaclass=RevealingMeta):
...     def __new__(cls):
...         print(cls, "__new__ called")
...         return super().__new__(cls)
...     def __init__(self):
...         print(self, "__init__ called")
...         super().__init__()
...
<class '__main__.RevealingMeta'> METACLASS __prepare__ called
<class '__main__.RevealingMeta'> METACLASS __new__ called
<class '__main__.RevealingClass'> METACLASS __init__ called
```

Как видите, в процессе определения класса вызываются только методы метакласса. Первый из них — `__prepare__()`, который подготавливает пространство имен нового класса. Сразу за ним следует метод `__new__()`, который непосредственно отвечает за создание класса и припимает нространство имен, созданное методом `__prepare__()`. Последний метод, `__init__()`, принимает объект класса, создапный методом `__new__()` (в данном случае — определение `RevealingClass`).

Методы метакласса взаимодействуют с методами класса во время создания экземпляра класса. Чтобы отследить порядок вызова методов, создадим новый экземпляр `RevealingClass` в интерактивном сеансе Python:

```
>>> instance = RevealingClass()
<class '__main__.RevealingClass'> METACLASS __call__ called
<class '__main__.RevealingClass'> CLASS __new__ called
<__main__.RevealingClass object at 0x10f594748> CLASS __init__ called
```

Первым вызванным методом оказался метод `__call__()` метакласса. В этот момент для него доступен объект класса (в данном случае — определение `RevealingClass`), но экземпляр класса еще не создан. Метод `__call__()` вызывается непосредственно перед созданием экземпляра класса, которое должно

произойти в методе `__new__()` определения класса. Последний шаг создания экземпляра — вызов принадлежащего классу метода `__init__()`, который отвечает за инициализацию экземпляра.

Итак, вы в общих чертах представляете себе, как работают метаклассы в теории. Рассмотрим пример их использования на практике.

Использование метаклассов

Метаклассы отлично подходят для решения нестандартных задач. Гибкость и мощь метаклассов позволяет изменять типичное поведение классов разными способами, поэтому трудно привести характерные примеры их использования. Проще сказать, какие применения метаклассов не характерны.

Например, возьмем метод `__prepare__()`, который есть у каждого типа объекта. Он подготавливает пространство имен атрибутов класса. Типом по умолчанию для пространства имен класса является обычный словарь. В течение многих лет каноническим примером `__prepare__()` считалось предоставление экземпляра `collections.OrderedDict` в качестве пространства имен класса.

Сохранение порядка атрибутов в пространстве имен класса делало возможным такие вещи, как повторяемые представления и сериализация объектов. Но так как, начиная с Python 3.7, словари гарантированно итерируются в порядке вставки ключей, этот пример утратил актуальность. Впрочем, это не означает, что нельзя как-то еще ноэкспериментировать с пространствами имен.

Представьте такую задачу: есть крупная кодовая база Python, которая накапливалась десятилетиями, и бóльшая часть кода образовалась задолго до того, как в команде стали уделять внимание стандартам написания кода. Например, в коде могут попадаться имена классов как в верблюжьем регистре (`camelCase`), так и в змеином (`snake_case`). Если вам захочется навести порядок, то придется приложить титанические усилия, чтобы привести всю кодовую базу к одной из двух схем. Но можно просто добавить поверх существующих классов «умный» метакласс, который позволит вызывать методы с обоими форматами имен. Тогда можно будет писать новый код с использованием новых соглашений об именовании (желательно использовать змеиный регистр), оставить старый код без изменений и ожидать постепенного обновления.

Вот где может пригодиться метод `__prepare__()`! Для начала напишем подкласс `dict`, который автоматически интерполирует имена с верблюжьим регистром в ключи со змеиным регистром:

```
from typing import Any
import inflection
```

```
class CaseInterpolationDict(dict):
    def __setitem__(self, key: str, value: Any):
        super().__setitem__(key, value)
        super().__setitem__(inflection.underscore(key), value)
```

Чтобы немного упростить задачу, мы воспользовались модулем `inflection`, который не входит в стандартную библиотеку. Он умеет преобразовывать строки между различными стилями регистра символов. Модуль можно загрузить из PyPI с помощью pip:

```
$ pip install inflection
```

Наш класс `CaseInterpolationDict` работает почти так же, как обычный тип `dict`, но при каждом сохранении нового значения оно сохраняется с двумя ключами: исходным и преобразованным к змеиному регистру. Обратите внимание, что в качестве родительского класса мы указали тип `dict` вместо рекомендованного `collections.UserDict`. Дело в том, что мы будем использовать этот класс в методе `__prepare__()` метакласса, а Python требует, чтобы пространства имен были экземплярами `dict`.

Пришло время написать реальный метакласс, который переопределяет тип пространства имен класса. Он оказывается на удивление коротким:

```
class CaseInterpolatedMeta(type):
    @classmethod
    def __prepare__(mcs, name, bases):
        return CaseInterpolationDict()
```

Все готово, и можно использовать метакласс `CaseInterpolatedMeta`, чтобы создать фиктивный класс с несколькими методами, которые используют змеиный регистр:

```
class User(metaclass=CaseInterpolatedMeta):
    def __init__(self, firstName: str, lastName: str):
        self.firstName = firstName
        self.lastName = lastName

    def getDisplayName(self):
        return f"{self.firstName} {self.lastName}"

    def greetUser(self):
        return f"Hello {self.getDisplayName()}!"
```

Сохраним весь код в файле case_user.py и запустим интерактивный сеанс, чтобы понаблюдать за поведением класса `User`:

```
>>> from case_user import User
```

Прежде всего обратите внимание на содержимое атрибута `User.__dict__`:

```
>>> User.__dict__
mappingproxy({
    '__module__': 'case_class',
    '__init__': <function case_class.User.__init__(self, firstName:
str, lastName: str)>,
    'getDisplayName': <function case_class.User.getDisplayName(self)>,
    'get_display_name': <function case_class.User.
getDisplayName(self)>,
    'greetUser': <function case_class.User.greetUser(self)>,
    'greet_user': <function case_class.User.greetUser(self)>,
    '__dict__': <attribute '__dict__' of 'User' objects>,
    '__weakref__': <attribute '__weakref__' of 'User' objects>,
    '__doc__': None
})
```

Первое, что бросается в глаза, — дублирование методов (именно этого мы и добивались). Второй важный момент заключается в том, что `User.__dict__` относится к типу `mappingproxy`. Это объясняется тем, что когда Python создает повый объект класса, он всегда копирует содержимое объекта пространства имен в новый объект `dict`. Тип `mappingproxy` также предоставляет опосредованный доступ к суперклассам в MRO класса.

Чтобы проверить, работает ли наше решение, вызовем все его методы:

```
>>> user = User("John", "Doe")
>>> user.getDisplayName()
'John Doe'
>>> user.get_display_name()
'John Doe'
>>> user.greetUser()
'Hello John Doe!'
>>> user.greet_user()
'Hello John Doe!'
```

Работает! Мы можем вызывать методы с именами в змеином регистре, хотя и пе определяли их. Неопытному разработчику это покажется волшебством!

Тем не менее этим волшебством следует пользоваться очень осторожно. На самом деле мы рассмотрели этот игрушечный пример для того, чтобы продемонстрировать возможности метаклассов всего в нескольких строках кода. Делать нечто подобное в большой и сложной кодовой базе очень рискованно. Метаклассы взаимодействуют с моделью данных Python на самом низком уровне, что может привести к различным подводным камням. Некоторые из них описаны в следующем разделе.

Подводные камни метаклассов

Метаклассы обладают мощными возможностями, но всегда усложняют код. Они также плохо комбинируются друг с другом, и если вы попытаетесь объединить несколько метаклассов с помощью наследования, то быстро столкнетесь с проблемами.

Как и другие продвинутые возможности Python, метаклассы чрезвычайно эластичны, и ими быстро начинают злоупотреблять. Хотя сигнатура вызова класса достаточно строго определена, Python не контролирует тип возвращаемого параметра. Он может быть каким угодно — при условии, что возвращаемый объект принимает входные аргументы при вызовах и содержит все требуемые атрибуты.

Одним из таких объектов из разряда «какой угодно» может быть экземпляр класса `Mock` из модуля `unittest.mock`. `Mock` не является метаклассом, не наследует классу `type` и не возвращает объект класса при создании экземпляра. Тем не менее его можно включить как именованный аргумент `metaclass` в определение класса, и это не приведет ни к каким синтаксическим ошибкам.

Конечно, использовать `Mock` как метакласс абсолютно бессмысленно, но рассмотрим следующий пример:

```
>>> from unittest.mock import Mock
>>> class Nonsense(metaclass=Mock):  # pointless, but illustrative
...     pass
...
>>> Nonsense
<Mock spec='str' id='4327214664'>
```

Нетрудно предсказать, что любая попытка создать экземпляр псевдокласса `Nonsense` завершится неудачей. Но что действительно интересно, — так это исключение, которое вы получите при такой понытке, и соответствующая трассировка:

```
>>> Nonsense()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/unittest/mock.py", line 917, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/unittest/mock.py", line 976, in _mock_call
    result = next(effect)
StopIteration
```

Дает ли исключение `StopIteration` какую-то информацию о том, что проблема может быть в определении класса на уровне метакласса? Очевидно, нет. Этот пример показывает, как трудно отлаживать код метаклассов, если неизвестно, где искать ошибки.

Но некоторые задачи непросто решить без метаклассов. Например, трудно представить реализацию ORM в Django, построенную без интенсивного применения метаклассов. Теоретически это возможно, но вряд ли полученное решение будет таким же простым и удобным, как с метаклассами. Фреймворки — область, в которой метаклассы по-настоящему проявляют свой потенциал. Обычно в них много сложного внутреннего кода, который нелегко понимать и анализировать, но который в конечном итоге позволяет другим программистам писать более компактный и удобочитаемый код, работающий на более высоком уровне абстракции.

Для несложных задач, таких как изменение атрибутов чтения и записи или добавление новых атрибутов, вместо метаклассов можно использовать более простые решения: свойства, дескрипторы или декораторы классов. Также существует специальный метод с именем `__init__subclass__()`, который во многих ситуациях может стать альтернативой для метаклассов. Мы подробнее рассмотрим его в следующем разделе.

Метод __init__subclass__() как альтернатива для метаклассов

Декоратор `@autorepr`, представленный в разделе «Следующий этап: декораторы классов», был довольно простым и полезным. К сожалению, у него есть недостаток, о котором мы еще не упоминали: он плохо сочетается с подклассами.

`@autorepr` будет нормально работать с обычными классами, которые созданы под конкретную задачу и не имеют потомков, но стоит вам начать подклассировать исходный декорированный класс, как вы заметите, что он работает не так, как ожидалось. Рассмотрим такую иерархию наследования:

```
from typing import Any
from autorepr import autorepr

@autorepr
class MyClass:
    attr_a: Any
    attr_b: Any
    attr_c: Any

    def __init__(self, a, b):
```

```
        self.attr_a = a
        self.attr_b = b

class MyChildClass(MyClass):
    attr_d: Any

    def __init__(self, a, b):
        super().__init__(a, b)
```

Если вы попытаетесь получить представление экземпляров `MyChildClass` в интерактивном сеансе интерпретатора, то результат будет выглядеть примерно так:

```
<MyChildClass: attr_a='Ultimate answer', attr_b=42, attr_c=UNSET>
<MyChildClass: attr_a=[1, 2, 3], attr_b=['a', 'b', 'c'], attr_c=UNSET>
```

И это понятно. Декоратор `@autorepr` использовался только с базовым классом, поэтому у него нет доступа к аннотациям подкласса. `MyChildClass` наследует неизменепный метод `__repr__()`.

Чтобы решить проблему, можно добавить декоратор `@autorepr` в подкласс:

```
@autorepr
class MyChildClass(MyClass):
    attr_d: Any
    def __init__(self, a, b):
        super().__init__(a, b)
```

Но как сделать, чтобы декоратор класса автоматически применялся к подклассам? Конечно, можпо воспроизвести то же поведение с метаклассами, но мы уже знаем, что это сильно усложнит код. Кроме того, использовать класс будет проблематичнее, потому что практически невозможно паследовать классы, которые используют разные метаклассы.

К счастью, у этой проблемы есть решение. Классы Python предоставляют метод-перехватчик `__init_subclass__()`, который вызывается для каждого подкласса. Это удобная альтерпатива проблематичным метаклассам. Этот перехватчик просто сообщает базовому классу, что у него есть нодклассы. Оп часто позволяет упрощать разные паттерны, связапные с событиями и сигналами (см. главу 7), но с его помощью также можно создавать декораторы «наследуемых» классов.

Рассмотрим такую модификацию декоратора `@autorepr`:

```
def autorepr(cls):
    attrs = set.union(
        *(
```

```
            set(c.__annotations__.keys())
            for c in cls.mro()
            if hasattr(c, "__annotations__")
        )
    )

    def __repr__(self):
        return repr_instance(self, sorted(attrs))
    cls.__repr__ = __repr__

    def __init_subclass__(cls):
        autorepr(cls)
    cls.__init_subclass__ = classmethod(__init_subclass__)

    return cls
```

Здесь появляется метод `__init_subclass__()`, который будет вызываться для нового объекта класса каждый раз, когда декорируемый класс подклассируется. В этом методе мы просто повторно применяем декоратор `@autorepr`. У него будет доступ ко всем новым аннотациям, и он также сможет внедряться в другие нодклассы. При таком нодходе вам не придется вручную добавлять декоратор для каждого нового подкласса, и можно будет гарантировать, что у всех методов `__repr__()` всегда будет достун к новым аннотациям.

До сих пор мы рассматривали встроенные возможности Python, которые унрощают приемы метапрограммирования. Вы видели, что Python обладает богатыми возможностями в этой области благодаря собственным средствам интроспекции, метаклассам и гибкой объектной модели. Однако существует ветвь метанрограммирования, которая достунна практически для любого языка независимо от его возможностей, — это генерация кода. Она рассматривается в следующем разделе.

Генерация кода

Как упоминалось ранее, динамическая генерация кода — самый сложный механизм метапрограммирования. В Python есть инструменты, которые позволяют генерировать и выполнять код и даже вносить некоторые изменения в уже скомпилированные объекты кода.

Различные нроекты, такие как Ну (о котором будет рассказано позднее), ноказывают, что средствами генерации кода в Python можно заново реализовать целые языки. Таким образом, перед разработчиками открываются практически неограниченные возможности. Зная, насколько обширна эта тема и сколько ловушек ноджидает на этом нути, я даже не буду пытаться давать подробные рекомендации о том, как генерировать код этим сноособом, или приводить практические примеры.

Но как бы то ни было, о такой возможности полезно знать, если вы намерены изучать эту область самостоятельно. Таким образом, рассматривайте этот раздел только как краткую сводку возможных отправных точек для дальнейшего изучения.

Рассмотрим примеры использования функций `exec()`, `eval()` и `compile()`.

exec, eval и compile

В Python есть три встроенные функции для того, чтобы вручную выполнять, вычислять и компилировать произвольный код на Python:

- `exec(object, globals, locals)`: динамически выполняет код на Python. Атрибут `object` должен быть строкой или кодовым объектом (см. описание функции `compile()`) и представлять отдельную инструкцию или последовательность из нескольких инструкций. Необязательные аргументы `globals` и `locals` предоставляют глобальное и локальное пространства имен для выполняемого кода. Если они не передаются, код выполняется в текущей области видимости. Если они заданы, аргумент `globals` должен быть словарем, тогда как `locals` может быть произвольным объектом отображения. Функция `exec()` всегда возвращает `None`.
- `eval(expression, globals, locals)`: вычисляет заданное выражение и возвращает полученное значение. Напоминает `exec()`, но ожидает, что аргумент `expression` будет отдельным выражением Python, а не последовательностью инструкций.
- `compile(source, filename, mode)`: компилирует исходный код в кодовый объект или объект абстрактного синтаксического дерева (AST). Исходный код предоставляется в виде строкового значения в аргументе `source`. Аргумент `filename` — имя файла, из которого взят код. При отсутствии файла (например, потому что код был создан динамически) обычно используется значение `"<string>"`. Аргумент `mode` должен содержать либо строку `"exec"` (последовательность инструкций), либо `"eval"` (отдельное выражение), либо `"single"` (одна интерактивная команда, как в интерактивном сеансе Python).

Начать овладевать динамической генерацией кода проще всего с функций `exec()` и `eval()`, потому что они работают со строками. Если вы умеете программировать на Python, то, возможно, уже знаете, как правильно программно генерировать работоспособный исходный код.

Очевидно, в контексте метапрограммирования полезнее всего функция `exec()`, потому что она позволяет выполнить произвольную последовательность инструкций Python. Слово «произвольную» должно вас насторожить. Даже функция `eval()`, которая позволяет вычислять только выражения, и даже в руках

опытного программиста может создать серьезные уязвимости в безопасности, особенно если принимает ввод от пользователя.

Помните, что фатальный сбой интерпретатора Python — неприятность, которой следует бояться меньше всего. Зато если из-за безответственного использования `exec()` и `eval()` у вас появится уязвимость для эксплойтов с удаленным выполнением кода, это может испортить вашу репутацию профессионального разработчика или даже стоить вам рабочего места. Это означает, что ни `exec()`, ни `eval()` никогда нельзя использовать с непроверенным вводом. А любой ввод, поступающий от конечного пользователя, всегда должен считаться небезопасным.

Даже если использовать `exec()` и `eval()` с проверенным вводом, нужно учитывать целый список мелких нюансов (слишком длинный для того, чтобы приводить его здесь), которые могут повлиять на работу вашего приложения так, как вы совершенно не ожидали. Армин Ронакер (Armin Ronacher) написал хорошую статью, где перечислены самые важные из этих нюансов; она называется «Будьте осторожны с exec и eval в Python» («Be careful with exec and eval in Python», http://lucumr.pocoo.org/2011/2/1/exec-in-python/).

Несмотря на все эти пугающие предупреждения, бывают естественные ситуации, в которых использование `exec()` и `eval()` полностью оправданно. Тем не менее когда у вас возникают хотя бы малейшие сомнения, не используйте эти функции и попытайтесь найти другое решение.

Сигнатура функции `eval()` может создать впечатление, что если передать пустые пространства имен `globals` и `locals` и обернуть их в надлежащие инструкции `try...except`, это будет относительно безопасно. Ничего подобного! Нед Батчелдер (Ned Batchelder) написал очень хорошую статью, в которой показал, как с помощью `eval()` вызвать в интерпретаторе ошибку сегментации, даже если все встроенные средства Python недоступны (см. http://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html). Это должно убедить вас в том, что функции `exec()` и `eval()` никогда не следует использовать с непроверенным вводом.

В следующем разделе рассматривается абстрактное синтаксическое дерево.

Абстрактное синтаксическое дерево

Перед тем как код на Python компилируется в байт-код, он преобразуется в формат **абстрактного синтаксического дерева** (**AST**, abstract syntax tree). Это древовидное представление абстрактной синтаксической структуры исходного

кода. Обработка грамматики Python обеспечивается встроенным модулем `ast`. Низкоуровневые деревья AST кода Python можно создавать либо функцией `compile()` с флагом `ast.PyCF_ONLY_AST`, либо вспомогательной функцией `ast.parse()`. Обратное преобразование не так просто, и в стандартной библиотеке для него нет функции. Впрочем, на это способны некоторые проекты, скажем, PyPy.

Модуль `ast` предоставляет вспомогательные функции для работы с AST, например:

```
>>> import ast
>>> tree = ast.parse('def hello_world(): print("hello world!")')
>>> tree
<_ast.Module object at 0x00000000038E9588>
>>> print(ast.dump(tree, indent=4))
Module(
    body=[
        FunctionDef(
            name='hello_world',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[
                            Constant(value='hello world!')],
                        keywords=[]))],
            decorator_list=[])],
    type_ignores=[])
```

Важно знать, что дерево AST можно модифицировать перед тем, как нередавать его функции `compile()`. Это открывает много новых перспектив. Например, можно добавлять новые синтаксические узлы, чтобы внедрять донолнительные средства анализа, скажем, оценку тестового покрытия. Также можно модифицировать имеющееся кодовое дерево, чтобы расширить существующий синтаксис новой семантикой. Этот прием используется в проекте MacroPy (https://github.com/lihaoyi/macropy) и добавляет сиптаксические макросы в Python с использованием уже существующего синтаксиса (рис. 8.3).

К сожалению, MacroPy несовместим с последними версиями Python и тестировался только для работы с Python 3.4. Тем не менее это очень интересный проект, который демонстрирует, чего можно добиться манипуляциями с AST.

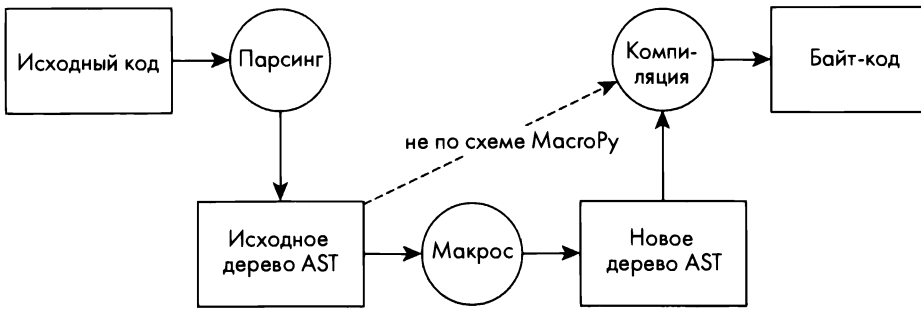

Рис. 8.3. Как MacroPy добавляет синтаксические макросы в модули Python при импортировании

Деревья AST также можно строить полностью искусственно, без парсинга какого-либо исходного кода. Это позволяет программистам на Python создавать байт-код Python для самодельных предметно-ориентированных языков и даже целиком реализовывать другие языки программирования поверх виртуальных машин Python.

Перехватчики импортирования

По сути, использовать способность MacroPy модифицировать исходные AST не сложнее, чем использовать команду `import macropy.activate`, которая неким образом переопределяет поведение импортирования в Python. В этом нет ничего волшебного, потому что Python позволяет вмешиваться в процесс импортирования с помощью двух видов перехватчиков (хуков):

- **Метаперехватчики** (meta hooks): вызываются до всей остальной обработки импортирования. С их помощью можно переопределить способ обработки `sys.path` даже для зафиксированных и встроенных модулей. Чтобы подключить новый метаперехватчик, нужно добавить в список `sys.meta_path` новый объект **искателя метапути** (meta path finder).
- **Перехватчики пути импортирования** (import path hooks): вызываются как часть обработки `sys.path` и используются, если обнаружен элемент пути, связанный с заданным перехватчиком. Чтобы подключить новый перехватчик пути импортирования, нужно добавить в список `sys.path_hooks` новый объект **искателя элемента пути** (path entry finder).

Реализация искателей путей и метапутей подробно описана в официальной документации Python (см. https://docs.python.org/3/reference/import.html). Если вы хотите взаимодействовать с импортированием на этом уровне, используйте

официальную документацию как основной источник информации. Дело в том, что механизмы импортирования в Python весьма сложны, и любые попытки бегло охарактеризовать их в нескольких абзацах обречены на неудачу. Здесь мы хотим просто показать, какие манипуляции с импортированием в принципе возможны.

В нескольких ближайших разделах рассматриваются проекты, в которых используются паттерны генерации кода.

Примечательные примеры генерации кода в Python

Трудно найти сколько-нибудь полезную реализацию библиотеки, которая опиралась бы на механизм генерации кода и при этом не была бы экспериментальной разработкой или просто проверкой концепции. Причины этого вполне очевидны:

- Обоснованный страх перед функциями `exec()` и `eval()`, потому что их безответственное использование может привести к настоящей катастрофе.
- Успешные системы с генерацией кода очень трудно разрабатывать и сопровождать, потому что для этого нужно глубоко понимать язык и в целом обладать выдающейся квалификацией программиста.

Несмотря на эти трудности, некоторые проекты успешно идут по этому пути и достигают либо улучшенного быстродействия, либо результатов, которых невозможно было бы добиться другими способами.

Компилируемый маршрутизатор Falcon

Falcon (http://falconframework.org/) — это минималистский веб-фреймворк WSGI для Python, который предназначен для построения быстрых и легких веб-API. Он настойчиво продвигает архитектурный стиль REST, который сейчас весьма популярен в веб-программировании. Это хорошая альтернатива другим, относительно тяжеловесным фреймворкам, таким как Django или Pyramid. Также это сильный конкурент других микрофреймворков, ориентированных на простоту, таких как Flask, Bottle или web2py.

Одна из лучших особенностей Falcon — очень простой механизм маршрутизации. Он не настолько сложен, как маршрутизация `urlconf` в Django, и не предоставляет таких богатых возможностей, но в большинстве случаев его достаточно для любых API, которые следуют архитектурному паттерну REST. Самое интересное в маршрутизации Falcon — ее внутренний механизм. Маршрутизатор Falcon

реализован с использованием кода, который генерируется на основе списка маршрутов, и код меняется каждый раз, когда регистрируется новый маршрут. Все это нужно для того, чтобы маршрутизация работала быстро.

Рассмотрим очень короткий пример API из веб-документации Falcon:

```
# sample.py
import falcon
import json

class QuoteResource:
    def on_get(self, req, resp):
        """Обрабатывает запросы GET"""
        quote = {
            'quote': 'Меня всегда больше интересовало '
                     'будущее, чем прошлое.',
            'author': 'Грейс Хоппер'
        }

        resp.body = json.dumps(quote)

api = falcon.API()
api.add_route('/quote', QuoteResource())
```

В двух словах, вызов метода `api.add_route()` динамически обновляет все сгенерированное дерево кода для маршрутизатора запросов Falcon. Он также компилирует дерево функцией `compile()` и генерирует новую функцию поиска маршрута с помощью `eval()`. Давайте повнимательнее присмотримся к атрибуту `__code__` функции `api._router._find()`:

```
>>> api._router._find.__code__
<code object find at 0x00000000033C29C0, file "<string>", line 1>
>>> api.add_route('/none', None)
>>> api._router._find.__code__
<code object find at 0x00000000033C2810, file "<string>", line 1>
```

Этот фрагмент показывает, что код функции был сгенерирован из строки, а не из реального файла с исходным кодом (файл `"<string>"`). Он также демонстрирует, что реальный объект кода изменяется с каждым вызовом метода `api.add_route()` (изменяется адрес объекта в памяти).

Hy

Hy (http://docs.hylang.org/) — диалект Lisp, написанный полностью на Python. Многие похожие проекты, которые реализуют другие языки программирования средствами Python, обычно ограничиваются тем, что разбирают «сырую» форму кода, который передается в виде строки или файлоподобного объекта, и интерпретируют его как серию явных вызовов Python. В отличие от них, Hy

можно рассматривать как язык, который работает нолностью в исполнительной среде Python, как и сам Python. Код, нанисанный на Ну, может использовать существующие встроенные модули и внешние пакеты и наоборот, а также может импортироваться обратно в Python.

Чтобы внедрить Lisp в Python, Ну транслирует код Lisp неносредственно в абстрактное синтаксическое дерево Python. Совместимость импортирования достигается благодаря перехватчику, который регистрируется при импортировании модуля Ну в Python. Каждый модуль с расширением .hy интернретируется как модуль Ну и может импортироваться как обычный модуль Python. Ниже нриведена нрограмма "hello world", написанная на этом диалекте Lisp:

```
;; hyllo.hy
(defn hello [] (print "hello world!"))
```

Следующий код на Python импортирует и выполняет эту программу:

```
>>> import hy
>>> import hyllo
>>> hyllo.hello()
    hello world!
```

Если заглянуть поглубже и дизассемблировать hyllo.hello с использованием встроенного модуля dis, можно заметить, что байт-код функции Ну не так сильно отличается от аналогичной функции на чистом Python:

```
>>> import dis
>>> dis.dis(hyllo.hello)
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_CONST               1 ('hello world!')
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 RETURN_VALUE
>>> def hello(): print("hello world!")
...
>>> dis.dis(hello)
  1           0 LOAD_GLOBAL              0 (print)
              3 LOAD_CONST               1 ('hello world!')
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP
             10 LOAD_CONST               0 (None)
             13 RETURN_VALUE
```

Как видите, байт-код фупкции на основе Ну короче, чем байт-код ее аналога на Python. Возможно, аналогичная тенденция нроявляется и в более крунных блоках кода. Она нноказывает, что создать совершенно новый язык на базе виртуальной машины Python — это вполне реально, и пожалуй, с этой возможностью стоит ноэкснериментировать.

Итоги

В этой главе затрагивалась необъятная тема метапрограммирования на Python. Мы описали синтаксические средства для реализации различных паттернов метанрограммирования, в первую очередь декораторы и метаклассы.

Также мы рассмотрели другой важный аспект метапрограммирования — динамическую генерацию кода. Эту тему удалось охватить лишь в общих чертах, потому что она слишком обширна для ограниченного объема книги. Впрочем, приведенный материал может стать неплохой отправной точкой, которая даст первичное представление о возможностях в этой области.

На примере Ну было показано, что метапрограммирование можно использовать даже для того, чтобы реализовывать другие языки поверх исполнительной среды Python. Конечно, разработчики Ну выбрали довольно необычный путь, и в общем случае Python интегрируется с другими языками через специализированные расширения интерпретатора Python или с помощью общих библиотек и интерфейсов внешних функций. Именно эти механизмы станут темой следующей главы.

9

Интеграция Python с C и C++

Python — замечательный язык, но он подходит не для всех задач. Иногда выясняется, что некоторые задачи проще решаются на другом языке. Возможно, этот другой язык оказывается выразительнее для некоторых технических областей (например, разработки систем управления, обработки изображений, системного программирования) или обладает естественными преимуществами быстродействия по сравнению с Python. Дело в том, что у Python (с реализацией CPython по умолчанию) есть ряд особенностей, которые не способствуют быстродействию:

- Польза от многопоточности сильно снижается для задач, требующих интенсивных вычислений, из-за глобальной блокировки (Global Interpreter Lock, GIL) в CPython и зависит от выбранной реализации Python.
- Python — некомпилируемый язык (как C или Go), поэтому в нем нет многих механизмов оптимизации времени компиляции.
- В Python не поддерживается статическая типизация и возможности оптимизации, которые ей присущи.

Но тот факт, что некоторые языки лучше подходят для конкретных задач, не означает, что ради них вам придется отказаться от Python. Используя надлежащие методы программирования, можно писать приложения, которые совмещают сильные стороны нескольких технологий.

Один из таких методов состоит в том, чтобы формировать архитектуру приложения из независимых компонентов, которые взаимодействуют друг с другом по четко определенным коммуникационным каналам. Часто такие решения воплощаются в форме **сервисно-ориентированных** или **микросервисных**

архитектур. Они чаще всего встречаются в распределенных системах, где каждый компонент (служба) системной архитектуры может работать независимо на отдельном хосте. Системы, нанисанные на нескольких языках, обычно называются **миогоязычиыми**.

Недостатком многоязычных сервисно-ориентированных или микросервисных архитектур является то, что для каждого языка, как правило, нриходится воссоздавать большое количество служебного кода нриложения. Сюда относятся такие вещи, как конфигурация приложения, журналирование, мониторинг и уровни коммуникации, а также разные фреймворки, библиотеки, средства сборки, общепринятые соглашения и паттерны нроектирования. Чтобы включить в систему эти инструменты и соглашения, потребуются затраты времени и будущих усилий на сопровождение, которые могут превысить выигрыш от добавления другого языка в архитектуру.

К счастью, эту проблему можно решить иначе. Часто то, что нам действительно нужно от другого языка, можно упаковать в изолированную библиотеку, которая делает что-то одно, и делает это хорошо. Нужно каким-то образом проложить мост между Python и другими языками, чтобы их библиотеки можно было использовать в нриложениях Python. Это можно сделать с помощью либо специализированных расширений CPython, либо так называемых **иитерфейсов внешних функций (FFI**, Foreign Function Interfaces).

В обоих случаях языки программирования C и C++ служат шлюзами к библиотекам и коду, написанному на разных языках. Интерпретатор CPython сам написан на языке C и предоставляет Python/C API, определенный в заголовочном файле Python.h. Этот API позволяет создавать совместно используемые библиотеки C, которые могут загружаться интернретатором. Такие расширения можно писать на C, а также на C++ ввиду его естественной совместимости с языком C. С другой стороны, с помощью FFI можно взаимодействовать с любой совместимой скомпилированной общей библиотекой независимо от того, на каком языке она написана. При этом такие библиотеки все равно должны опираться на соглашепия вызова и базовые тины языка C.

В этой главе обсуждаются основные причины, по которым пишутся собственные расширения на других языках, а также представлены нопулярные инструмепты для их создания. Здесь рассматриваются следующие темы:

- C и C++ как основа расширяемости Python.
- Компиляция и загрузка расширений Python на C.
- Написание расширений.
- Недостатки использования расширений.
- Как взаимодействовать с компилируемыми динамическими библиотеками без расширений.

Чтобы интегрировать Python с разными языками, нам потребуется ряд дополнительных инструментов и библиотек, поэтому начнем с технических требований для этой главы.

Технические требования

Чтобы компилировать расширения Python, упоминаемые в этой главе, вам понадобятся компиляторы C и C++. Ниже перечислены подходящие компиляторы, которые можно загрузить бесплатно для некоторых операционных систем:

- Visual Studio 2019 (Windows): https://visualstudio.microsoft.com
- GCC (Linux и большинство систем POSIX): https://gcc.gnu.org
- Clang (Linux и большинство систем POSIX): https://clang.llvm.org

В Linux компиляторы GCC и Clang обычно доступны в системах управления пакетами для конкретных дистрибутивов. В macOS компилятор входит в интегрированную среду Xcode (доступную в App Store).

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `Cython`
- `Cffi`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%209.

C и C++ как основа расширяемости Python

Эталонная реализация Python — интерпретатор CPython — написана на C. Поэтому взаимодействие Python с другими языками опирается на C и C++, который естественным образом совместим с C. Есть даже полное надмножество языка Python, которое называется Cython и использует транспилятор (source-to-source compliler), чтобы создавать расширения C для CPython с использованием расширенного синтаксиса Python.

Собственно, можно использовать динамические/общие библиотеки, написанные на любом языке, который поддерживает компиляцию в форме таких библиотек.

Таким образом, возможности межъязыковой интеграции значительно выходят за рамки C и C++. Дело в том, что общие библиотеки по своей природе достаточно универсальны: их можно использовать в любом языке, который поддерживает их загрузку. То есть даже если написать такую библиотеку на совершенно другом языке (скажем, Delphi или Prolog), ее можно будет использовать в Python. Правда, эту библиотеку вряд ли можно будет назвать расширением Python, если она не использует Python/C API.

К сожалению, писать собственные расширения только на C и C++ с помощью минимального Python/C API — весьма непростое дело. Для этого нужно не только хорошо разбираться в одном из двух относительно сложных языков, но и писать огромное количество шаблонного кода. Приходится создавать однообразный код, чья единственная задача — обеспечивать интерфейс, который связывает базовый код C и C++ с интерпретатором Python и его типами данных.

Тем не менее полезно знать, как создаются расширения на чистом C, и вот почему:

- Вы будете лучше понимать, как работает Python в целом.
- Возможно, когда-нибудь вам придется отлаживать или сопровождать расширения C/C++.
- Вы сможете представлять себе, как работают высокоуровневые инструменты для создания расширений.

Именно поэтому в этой главе вы сначала узнаете, как создать простое расширение Python на C с нуля. Позже мы повторно реализуем его другими средствами, которые не требуют использования низкоуровневого Python/C API. Но прежде чем углубляться в подробности разработки расширений, посмотрим, как откомпилировать и загрузить одно из них.

Компиляция и загрузка расширений Python на C

Интерпретатор Python может загружать расширения из динамических/общих библиотек (например, модулей Python), если они предоставляют прикладной интерфейс с использованием Python/C API. Определения всех функций, типов и макросов, образующих Python/C API, содержатся в заголовочном файле `Python.h`, который написан на C и распространяется с исходным кодом Python. Во многих дистрибутивах Linux этот заголовочный файл находится в отдельном пакете (например, `python-dev` в Debian/Ubuntu), но в Windows он по умолчанию

распространяется с интерпретатором. В POSIX и POSIX-совместимых системах (например, Linux и macOS) он находится в каталоге include/ установки Python, а в Windows — в каталоге Include/ этой установки.

Традиционно Python/C API изменяется с каждым выпуском Python. В большинстве случаев в API добавляются только такие средства, которые **совместимы на уровне исходного кода**. Однако чаще всего они не сохраняют **двоичную совместимость** из-за изменений в двоичном интерфейсе приложений (Application Binary Interface, ABI). А это означает, что расширения нужно компилировать отдельно для каждой основной версии Python. Кроме того, ABI разных ОС несовместимы друг с другом, из-за чего создать один двоичный дистрибутив для всех возможных сред практически нереально. Поэтому большинство расширений Python распространяется в виде исходного кода.

С выходом Python 3.2 было определено подмножество Python/C API, имеющее стабильный ABI. С этим ограниченным API можно создавать расширения, которые можно скомпилировать один раз для каждой операционной системы, и они будут работать во всех версиях Python от 3.2 и выше без перекомпиляции. Однако это ограничивает функциональность API и не решает проблему более старых версий Python. Кроме того, так не получится создать единый двоичный дистрибутив, который способен работать в разных операционных системах. Словом, у этого решения есть как достоинства, так и недостатки, и стабильность ABI может не оправдать пользы от него.

Важно знать, что Python/C API доступен только для реализаций CPython. Были попытки внедрить поддержку расширений в альтернативных реализациях (таких, как PyPI, Jython и IronPython), но похоже, на данный момент не существует стабильного и полноценного решения. Единственная альтернативная реализация Python, которая должна легко работать с расширениями, — Stackless Python, потому что, по сути, это просто модифицированная версия CPython.

Расширения на C для Python должны быть скомпилированы в общие/динамические библиотеки, прежде чем их можно будет импортировать. Не существует встроенных средств для импортирования кода на C/C++ в Python непосредственно из исходного кода. К счастью, вспомогательные функции из пакета `setuptools` позволяют определять скомпилированные расширения в виде модулей, которые можно компилировать и распространять с помощью сценария setup.py, как если бы они были обычными пакетами Python.

Создание пакетов Python более подробно рассматривается в главе 11 «Упаковка и распространение кода Python».

Вот пример сценария setup.py из официальной документации, который подготавливает для установки простой пакет с расширением, написанным на C:

```
from setuptools import setup, Extension

module1 = Extension(
    'demo',
    sources=['demo.c']
)


setup(
    name='PackageName',
    version='1.0',
    description='This is a demo package',
    ext_modules=[module1]
)
```

Распространение пакетов Python и скрипт setup.py подробнее рассматриваются в главе 11 «Упаковка и распространение кода Python».

После этой подготовки в процедуру распространения нужно включить дополнительный этап:

```
python3 setup.py build
```

Таким образом все расширения, определенные в аргументе `ext_modules`, компилируются с дополнительными настройками компилятора, переданными конструктору `Extension()`. Используется компилятор, который работает по умолчанию в вашей среде. Если пакет будет распространяться в виде исходного кода, этап компиляции необязателен. В этом случае необходимо убедиться, что в целевой среде есть все, что нужно для компиляции: комнилятор, заголовочные файлы и дополнительные библиотеки, которые будут скомпонованы с вашим двоичным файлом (если они требуются для расширения). Упаковка расширений Python подробнее рассматривается нозднее, в разделе «Недостатки расширений».

В следующем разделе мы разберемся, для чего могут понадобиться расширения.

Зачем нужны расширения

Нелегко определенно сказать, в каких случаях действительно стоит писать расширения на C/C++. Распространенное нравило — «не делать этого никогда, если есть другие варианты». Впрочем, это крайпе субъективное утверждение, которое оставляет простор для спекуляций на тему того, что возможно и что невозможно в Python. На самом деле трудно найти что-то такое, чего нельзя сделать с помощью чистого Python.

И все же существуют задачи, в которых расширения обеспечивают дополнительные возможности:

- обойти GIL в модели потоков CPython;
- улучшить быстродействие на критических участках кода;
- интегрировать исходный код, нанисанный на разных языках;
- интегрировать сторонние динамические библиотеки;
- создавать эффективные типы данных.

Конечно, для каждой такой задачи обычно существует жизнеспособное решение на чистом Python. Например, фундаментальные ограничения интерпретатора CPython (такие, как GIL) легко преодолеть, если перейти на другую модель конкурентного выполнения, скажем, корутины или многопроцессное выполнение вместо потоков (эти варианты обсуждались в главе 6 «Конкурентное выполнение»). Чтобы упростить работу со сторонними дипамическими библиотеками и специальпыми типами данных, библиотеки можно интегрировать с модулем `ctypes`, а любой тип данных можно реализовать в Python.

Тем не менее встроенные средства Python не всегда оптимальны. Чисто «питоническая» интеграция с внешней библиотекой может получиться неуклюжей и сложной в сопровождении. Реализация пользовательских типов данных может оказаться недостаточно эффективной без доступа к низкоуровпевому управлению памятью. Таким образом, окончательное решение о том, какой путь выбрать, всегда нужно принимать очень осторожно, учитывая многие факторы. Хороший подход — начать с реализации на чистом Python и задумываться о расширениях только тогда, когда встроенных возможностей оказывается недостаточно.

В следующем разделе вы узнаете, как с помощью расширений улучшить быстродействие на критических участках кода.

Улучшение быстродействия на критических участках кода

Будем откровенны: разработчики выбирают Python не из-за быстродействия. Скорость выполнения — не самая сильная сторона Python, хотя разработка на нем ведется быстро. Тем не менее, какими бы производительными мы ни были как программисты, мы иногда сталкиваемся с задачами, которые нельзя эффективно решить на чистом Python.

В большинстве случаев проблемы быстродействия решаются за счет правильного выбора алгоритмов и структур данных, а не за счет борьбы с постоянным фактором издержек самого языка. Полагаться на расширения, чтобы сэкономить

немного процессорного времени, — не лучшая идея, если код уже плохо написан или если в нем используются неэффективные алгоритмы.

Часто удается улучшить быстродействие до приемлемого уровня, не усложняя проект включением еще одного языка в технологический стек. И если можно ограничиться одним языком программирования, то именно так и стоит действовать.

Тем не менее даже с самыми хитроумными алгоритмами и предельно оптимизированными структурами данных не всегда удается преодолеть критические технологические ограничения, используя только Python.

Пример области, которая устанавливает четкие ограничения на быстродействие приложения, — аукцион в реальном времени (RTB, Real-Time Bidding). RTB — это покупка и продажа мест для показа онлайн-рекламы, аналогично тому, как работают офлайновые аукционы или биржи. Аукцион обычно проводится на площадке рекламной биржи, которая рассылает информацию о доступных рекламных площадях автоматизированным системам покупки (DSP, Demand-Side Platform), которые хотят приобрести места для своей рекламы. И здесь происходит самое интересное. Бóльшая часть рекламных торгов осуществляется по протоколу OpenRTB (основанному на HTTP), который обеспечивает взаимодействие с участниками торгов. Площадка DSP отвечает за выдачу ответов на HTTP-запросы OpenRTB. Рекламные биржи всегда устанавливают очень жесткие временные ограничения на продолжительность всего процесса: например, не более 50 миллисекунд от получения первого пакета TCP до записи последнего байта сервером DSP. Ситуация усугубляется тем, что платформы DSP нередко обрабатывают десятки тысяч запросов в секунду. От возможности сократить время ответа на несколько миллисекунд часто зависит прибыльность службы. Это означает, что в такой ситуации портирование даже тривиального кода на C может быть разумным, но только если этот код относится к узкому месту по быстродействию и его нельзя улучшить на алгоритмическом уровне. Как однажды сказал Гвидо: «Если вам нужно что-то побыстрее, <...> — цикл, написанный на C, вы все равно не обгоните».

В следующем разделе рассматривается совершенно иной сценарий использования для нестандартных расширений — интеграция кода, написанного на других языках.

Интеграция существующего кода на других языках

Хотя сфера IT относительно молода по сравнению с другими областями техники, в ней уже накопился богатый опыт предыдущих поколений. Многие замечательные разработчики написали полезные библиотеки, которые реша-

ют раснространенные задачи на разных языках нрограммирования. С одной стороны, большим упущением было бы забывать об этом наследии каждый раз, когда появляется новый язык; но с другой стороны, под каждый новый язык невозможно нортировать все ПО, которое было написано до его появлепия.

Похоже, С и C++ — самые важные языки, из которых вам может нонадобиться интегрировать в код приложения многие библиотеки и реализации, пе перенося их нолностью на Python. К счастью, интерпретатор CPython уже написан на С, так что самый естественный способ интегрировать код C/C++ — использовать специализированные расширепия.

В следующем разделе рассматривается очень похожий сцепарий иснользования: интеграция сторонних динамических библиотек.

Интеграция сторонних динамических библиотек

Интеграция кода, написанного с использованием других технологий, не сводится только к C/C++. Многие библиотеки, особенно сторонние продукты с закрытым исходным кодом, распространяются в виде скомпилированных двоичных файлов. В С очень легко загружать такие общие/динамические библиотеки и вызывать их функции. А следовательно, можно использовать любую библиотеку С, если обернуть ее в расширение Python с помощью Python/C API.

Конечно, это не единственный вариант. Существуют такие инструменты, как `ctypes` и CFFI, которые позволяют взаимодействовать с динамическими библиотеками напрямую на чистом Python без необходимости писать расширения на С. Часто Python/C API подходит лучше, потому что обеспечивает надлежащее разделение между уровнем интеграции (написанным на С) и остальной частью приложения.

Наконец, расширения можно использовать, чтобы обогащать Python новыми высокопроизводительными структурами данных.

Создание эффективных структур данных

Python предоставляет исключительно гибкий набор встроенных типов данных. Некоторые из них могут похвастаться передовыми внутренними реализациями (по крайней мере в Cpython), которые специально адаптированы для Python. Хотя ассортимент изначально доступных базовых типов и коллекций способен впечатлить новичка, очевидно, что он не покрывает абсолютно все нотребности программистов.

Конечно, в Python можно создать много нестандартных структур данных: либо как подклассы встроенных типов, либо как совершенно новые классы, построенные с нуля. К сожалению, иногда быстродействие таких структур оставляет желать лучшего. Вся мощь сложных коллекций, таких как `dict` и `set` происходит от реализации C, заложенной в их основу. Почему бы не сделать нечто подобное и не реализовать некоторые собственные структуры данных на C?

Теперь, когда вы знаете, для чего могут понадобиться пользовательские расширения Python, давайте посмотрим, как создать такое расширение.

Как писать расширения

Как говорилось ранее, писать расширения не так просто, однако потраченные усилия не пропадут даром. Проще всего создавать расширения, используя такие инструменты, как Cython. Он позволяет писать расширения C на языке, который сильно напоминает Python без замысловатых нюансов Python/C API. С Cython вы сможете работать эффективнее, и ваш код будет легче писать, читать и сопровождать.

Впрочем, если это ваше первое знакомство с темой расширений, лучше начать с того, чтобы написать код на простейшем C и Python/C API. Благодаря этому вы лучше поймете, как работают расширения, и сможете правильнее оценить преимущества альтернативных решений. Ради простоты мы возьмем несложную алгоритмическую задачу и попробуем реализовать ее двумя разными способами:

- расширение на чистом C;
- использование Cython.

Задача — вычислить n-е число Фибоначчи. Последовательность Фибоначчи начинается с 0 и 1, а каждый следующий элемент является суммой двух предыдущих. Первые 10 чисел Фибоначчи:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Как видите, последовательность легко объяснить и легко реализовать. Крайне маловероятно, что вам когда-нибудь придется создавать компилируемое расширение исключительно для этой задачи. Но поскольку она очень простая, на ее примере будет удобно разобраться, как писать любые функции C для Python/C API. Мы стремимся только к простоте и ясности, так что не пытаемся найти самое эффективное решение.

Прежде чем создать первое расширение, определим эталонную реализацию, чтобы иметь возможность сравнивать разные решения. Эталонная реализация функции Фибоначчи, реализованная на чистом Python, выглядит так:

```
"""Модуль Python, реализующий последовательность Фибоначчи"""

def fibonacci(n):
    """Возвращает n-е число Фибоначчи, вычисленное рекурсивно."""
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Заметим, что это одна из самых простых реализаций функции `fibonnaci()`. К ней можно применить массу улучшений. Мы не оптимизируем эту реализацию (например, с помощью паттерна мемоизации), потому что не в этом заключается цель нашего примера. Аналогичным образом мы не будем оптимизировать код позднее при обсуждении реализаций на C или Cython, хотя скомпилированный код и предоставляет для этого много возможностей.

Популярный прием мемоизации заключается в том, что результаты прошлых вызовов функции сохраняются, чтобы к ним можно было обращаться позднее и таким образом оптимизировать быстродействие приложения. Эта тема подробнее рассматривается в главе 13 «Оптимизация кода».

Следующий раздел посвящен расширениям на чистом C.

Расширения на чистом C

Если вы решили, что вам нужно писать расширения на C для Python, я буду считать, что вы уже владеете языком C на таком уровне, который позволит полностью понять приведенные в книге примеры. Книга посвящена языку Python, поэтому здесь не будут разъясняться технические детали, которые не относятся к Python/C API. Хотя этот API разрабатывался весьма тщательно, он определенно не годится в качестве введения в C. Таким образом, если вы вообще не знаете C, вам лучше не пытаться писать на нем расширения для Python, пока не накопите опыт работы с этим языком. До тех пор оставьте эту задачу другим специалистам, а сами выбирайте Cython или Pyrex, которые гораздо безопаснее для начинающих.

Как упоминалось ранее, мы попробуем перенести функцию `fibonacci()` в C и сделать так, чтобы код Python мог обращаться к ней как к расширению. Начнем с простейшей реализации, аналогичной приведенному ранее примеру на Python. Минимальная функция без использования Python/C API будет выглядеть примерно так:

```
long long fibonacci(unsigned int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

А вот пример завершенного, полностью функционального расширения, которое предоставляет доступ к этой функции в скомпилированном модуле:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

long long fibonacci(unsigned int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}

static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
        result = Py_BuildValue("L", fibonacci((unsigned int)n));
    }

    return result;
}

static char fibonacci_docs[] =
    "fibonacci(n): Возвращает n-е число Фибоначчи, "
    "вычисленное рекурсивно\n";


static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
```

```
    "fibonacci",
    "Модуль расширения, предоставляющий функцию вычисления чисел Фибоначчи",
    -1,
    fibonacci_module_methods
};


PyMODINIT_FUNC PyInit_fibonacci(void) {
    Py_Initialize();

    return PyModule_Create(&fibonacci_module_definition);
}
```

Я знаю, о чем вы подумали. На первый взгляд этот пример выглядит устрашающе: пришлось увеличить объем кода вчетверо просто для того, чтобы фупкция C `fibonacci()` стала доступна из Python. Не волпуйтесь, позже мы разберем каждую строку кода подробнее. Но сначала посмотрим, как упаковать этот код и выполнить его в Python.

В мипимальной копфигурации `setuptools` для пашего модуля нужно использовать класс `setuptools.Extension`, чтобы сообщить интерпретатору, как скомпилировано расширепие:

```
from setuptools import setup, Extension

setup(
    name='fibonacci',
    ext_modules=[
        Extension('fibonacci', ['fibonacci.c']),
    ]
)
```

Процесс сборки расширений можно инициализировать командой `setup.py build`, но он также автоматически выполняется при устаповке пакета. В следующем фрагменте показан результат устаповки в режиме редактировапия (комапда `pip` с флагом `-e`):

```
$ python3 -m pip install -e .
Obtaining file:///Users/.../Expert-Python-Programming-Fourth-Edition/
Chapter%209/02%20-%20Pure%20C%20extensions
Installing collected packages: fibonaccib
  Running setup.py develop for fibonacci
Successfully installed fibonacci
```

Режим редактировапия `pip` позволяет просмотреть файлы, созданные на этапе сборки. Ниже приведеп пример файлов, которые могут быть создапы в вашем рабочем каталоге в ходе установки:

```
$ ls -1ap
./
../
build/
fibonacci.c
fibonacci.cpython-39-darwin.so
fibonacci.egg-info/
setup.py
```

Файлы fibonacci.c и setup.py содержат исходный код. fibonacci.egg-info/ — это специальный каталог для метаданных накетов, и нока на него можно не обращать внимания. Зато сейчас для нас по-настоящему важен файл fibonacci.cpython-39-darwin.so. Это двоичная общая библиотека, совместимая с интерпретатором CPython. Интерпретатор Python будет загружать ее, когда мы понытаемся импортировать модуль `fibonacci`. Попробуем сделать это и просмотрим результат в интерактивном сеансе:

```
$ python3
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import fibonacci
>>> help(fibonacci)
Help on module fibonacci:

NAME
    fibonacci - Модуль расширения, предоставляющий функцию вычисления чисел
                Фибоначчи
FUNCTIONS
    fibonacci(...)
        fibonacci(n): Возвращает n-е число Фибоначчи, вычисленное рекурсивно
FILE
    /(...)/fibonacci.cpython-39-darwin.so
>>> [fibonacci.fibonacci(n) for n in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

А теперь поближе познакомимся с тем, как устроено наше расширение.

Детальное ознакомление с Python/C API

Итак, вы уже знаете, как нравильно упаковать, скомпилировать и установить собственное расширение C, и убедились в том, что оно работает как предполагалось. Пришло время рассмотреть код нодробнее.

Модуль расширепия начинается с директивы препроцессора C, которая включает заголовочный файл Python.h:

```
#include <Python.h>
```

Директива подключает весь Python/C API и все, что вам может потребоваться для разработки собственных расширений. В более реалистичной ситуации вашему коду понадобится гораздо больше директив препроцессора, чтобы пользоваться функциями стандартной библиотеки C или интегрировать другие исходные файлы. Наш пример был простым, поэтому других директив в нем не потребовалось.

Затем идет основная часть модуля:

```
long long fibonacci(unsigned int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

Функция `fibonacci()` — единственная часть кода, которая делает что-то полезное. Это реализация на чистом языке C, который Python не понимает сам по себе. Остальной код примера создает уровень интерфейса, который будет предоставлять функцию `fibonacci()` через Python/C API.

Чтобы у Python был доступ к этому коду, в качестве первого шага мы создаем функцию C, совместимую с интерпретатором CPython. В Python нет ничего, кроме объектов. Это означает, что функции C, вызываемые в Python, тоже должны возвращать полноценные объекты Python. Python/C API предоставляет тип `PyObject`, и каждый вызываемый объект должен возвращать указатель на него. Сигнатура нашей функции такова:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args)
```

Обратите внимание: в этой сигнатуре не указан точный список аргументов, зато аргумент `PyObject* args` содержит указатель на структуру с кортежем переданных значений.

Фактическая проверка списка аргументов должна происходить в теле функции; именно это делает `fibonacci_py()`. Она анализирует список аргументов `args`, предполагая, что он содержит единственное значение типа `unsigned int`, и использует это значение как аргумент функции `fibonacci()`, чтобы получить соответствующее число Фибоначчи:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
```

```
        result = Py_BuildValue("L", fibonacci((unsigned int)n));
    }

    return result;
}
```

Функция в этом примере содержит серьезную ошибку, которая моментально бросится в глаза любому опытному разработчику. Попробуйте найти ее в качестве упражнения по работе с расширениями C. Ради краткости мы временно оставим все как есть и исправим ошибку позднее, когда будем разбираться с ошибками и исключениями в разделе «Обработка исключений».

Строка `"l"` (буква L в нижнем регистре) в вызове `PyArg_ParseTuple(args, "l", &n)` означает, что аргумент `args` должен содержать только одно значение `long`. В случае сбоя функция вернет `NULL`, а информация об исключении будет сохранена в состоянии интерпретатора на уровне потока.

Фактическая сигнатура функции синтаксического разбора имеет вид `int PyArg_ParseTuple(PyObject *args, const char *format, ...)`, а после форматной строки идет список аргументов переменной длины, который представляет вывод разобранного значения (в виде указателей). Функция `scanf()` из стандартной библиотеки C работает аналогичным образом. Если допущение окажется ошибочным и пользователь передаст несовместимый список аргументов, то `PyArg_ParseTuple()` выдаст соответствующее исключение. Кодировать сигнатуры функций таким способом очень удобно, если вы к нему привыкли, но у него есть огромный недостаток по сравнению с обычным кодом на Python. Такие сигнатуры, неявно определяемые вызовами `PyArg_ParseTuple()`, нельзя легко проанализировать в интернретаторе Python. Необходимо помнить об этом, когда вы используете код, предоставляемый в виде расширений.

Как говорилось ранее, Python ожидает, что вызываемые объекты будут возвращать объекты. Это означает, что скалярное значение `long`, полученное от функции `fibonacci()`, нельзя вернуть как результат `fibonacci_py()`. Такая попытка даже не скомпилируется, и базовые типы C не будут автоматически преобразовываться в объекты Python.

Поэтому нужно использовать функцию `Py_BuildValue(*format, ...)`. Она подобна функции `PyArg_ParseTuple()` и получает похожий набор форматных строк. Главное отличие заключается в том, что список аргументов — не вывод, а ввод функции, поэтому вместо указателей должны передаваться фактические значения.

После того как функция `fibonacci_py()` определена, бóльшая часть черной работы позади. Остается последний шаг — инициализировать модуль и добавить

метаданные в нашу функцию, чтобы немного упростить пользователям работу с ней. Это шаблонная часть кода расширения. В простых примерах (таких, как наш) она может занимать больше места, чем сами функции, к которым мы хотим предоставить доступ. В большинстве случаев эта часть состоит из нескольких статических структур и одной функции инициализации, которую интерпретатор будет выполнять при импортировании модуля.

Сначала мы создаем статическую строку, которая станет содержимым doc-строки Python для функции `fibonacci_py()`:

```
static char fibonacci_docs[] =
    "fibonacci(n): Возвращает n-е число Фибоначчи, "
    "вычисленное рекурсивно\n";
```

Вообще говоря, эту строку можно было встроить позднее в `fibonacci_module_methods`, но хороший стиль требует отделять doc-строки и хранить их вблизи от фактического определения функции, к которой они относятся.

Следующая часть определения — массив структур `PyMethodDef`, которые определяют методы (функции), доступные в нашем модуле. Структура `PyMethodDef` содержит ровно четыре поля:

- `char* ml_name`: имя метода.
- `PyCFunction ml_meth`: указатель на реализацию функции на языке C.
- `int ml_flags`: флаги, обозначающие соглашение вызова или соглашение связывания (последнее актуально только для определения методов классов).
- `char* ml_doc`: указатель на содержимое doc-строки метода/функции.

Такой массив всегда должен завершаться сигнальной меткой `{NULL, NULL, 0, NULL}`, которая просто обозначает конец структуры. В нашем простом примере создается статический массив `PyMethodDef fibonacci_module_methods[]`, в котором есть только два элемента (включая сигнальную метку):

```
static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
    {NULL, NULL, 0, NULL}
};
```

Первый элемент отображается на структуру `PyMethodDef` следующим образом:

- `ml_name = "fibonacci"`: функция C `fibonacci_py()` будет фигурировать как функция Python с именем `fibonacci`.
- `ml_meth = (PyCFunction)fibonacci_py`: преобразование к `PyCFunction` обязательно в Python/C API и выполняется в соответствии с соглашением вызова, которое определено далее в `ml_flags`.

- `ml_flags = METH_VARARGS`: здесь флаг `METH_VARARGS` означает, что соглашение вызова нашей функции нринимает переменный список аргументов и пе принимает именованных аргументов.
- `ml_doc = fibonacci_docs`: функция Python документируется содержимым строки `fibonacci_docs`.

Когда массив определений функций готов, можно создать другую структуру, которая содержит определение всего модуля. Она описывается типом `PyModuleDef` и содержит группу полей. Некоторые из них нужны только для более сложных сценариев, где требуется детализированный контроль за процессом инициализации модуля. Здесь нас интересуют только нервые пять полей:

- `PyModuleDef_Base m_base`: всегда должно инициализироваться значением `PyModuleDef_HEAD_INIT`.
- `char* m_name`: имя только что созданного модуля (в нашем случае `fibonacci`).
- `char* m_doc`: указатель на содержимое doc-строки модуля. Обычно в одном исходном файле C определяется только один модуль, поэтому ничто не мешает встроить строку документации на уровне всей структуры.
- `Py_ssize_t m_size`: размер памяти, выделенной для хранения состояния модуля. Используется только в ситуациях, где нужна поддержка нескольких субинтерпретаторов или многофазной инициализации. В большинстве случаев это поле не требуется и содержит значение −1.
- `PyMethodDef* m_methods`: указатель на массив с функциями уровня модуля, которые описаны значениями `PyMethodDef`. Может иметь значение `NULL`, если модуль не предоставляет никаких функций. В нашем примере значение — `fibonacci_module_methods`.

Другие поля подробно рассматриваются в официальной документации Python (см. https://docs.python.org/3/c-api/module.html), но в расширении из нашего примера опи не нужны. Неиспользуемые поля должны иметь зпачение `NULL`, и они будут неявно инициализированы этим значением, если его пе присвоить. Поэтому описание модуля, содержащееся в переменной `fibonacci_module_definition`, может иметь простую форму:

```
static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
    "fibonacci",
    "Модуль расширения, предоставляющий функцию вычисления чисел Фибоначчи",
    -1,
    fibonacci_module_methods
};
```

Последний фрагмент кода — функция инициализации модуля. Ее имя должно удовлетворять предельно конкретным правилам, чтобы интерпретатор Python

легко нашел ее, когда загружается общая/динамическая библиотека. Функция должна называться `PyInit_<имя>`, где `<имя>` — имя модуля. Это та же строка, которая использовалась в поле `m_base` определения `PyModuleDef` и в первом аргументе вызова `setuptools.Extension()`. Если модулю не нужен сложный процесс ипициализации, то функция принимает очень простую форму, как в этом примере:

```
PyMODINIT_FUNC PyInit_fibonacci(void) {
    return PyModule_Create(&fibonacci_module_definition);
}
```

`PyMODINIT_FUNC` — это макрос препроцессора, который объявляет, что функция инициализации возвращает объект типа `PyObject*`, и добавляет особые объявления компоновки, если платформа их требует.

Между функциями Python и C есть одно очень важное различие: соглашения вызова и связывания. Эта тема достаточно обширная, поэтому мы обсудим ее в следующем разделе.

Соглашения вызова и связывания

Python — объектно-ориентированный язык с гибкими соглашениями вызова, где используются как нозиционные, так и именованные аргументы. Рассмотрим следующий вызов функции `print()`:

```
print("hello", "world", sep=" ", end="!\n")
```

Первые два выражепия, передаппые при вызове (выражения `"hello"` и `"world"`), являются нозиционными и будут соответствовать позиционному аргументу функции `print()`. Порядок важен, и если изменить его, то вызванная функция вернет другой результат. С другой стороны, следующие выражения, `" "` и `"!\n"`, будут сопоставлены с именоваппыми аргументами. Их порядок не важен при условии, что имепа остаются пеизменными.

C — процедурный язык, и в нем поддерживаются только позиционные аргументы. В расширениях для Python необходимо поддерживать гибкость аргументов и объектно-ориентированную модель дапных Python. Это достигается в основном за счет того, что поддерживаемые соглашения вызова и связывания объявляются явно.

Как объяснялось в разделе «Детальное ознакомление с Python/C API», битовое поле `ml_flags` структуры `PyMethodDef` содержит флаги соглашений вызова и связывания. Флаги соглашений вызова таковы:

- `METH_VARARGS`: типичное соглашение для функций или методов Python, которые получают только позиционные аргументы. В поле `ml_meth` для такой

функции нужно передавать тип PyCFunction. Функция будет принимать два аргумента типа PyObject*. Первый из них — объект self (для методов) или объект module (для функций модулей). Вот типичная сигнатура функции C с этим соглашением вызова: PyObject* function(PyObject* self, PyObject* args).

- METH_KEYWORDS: соглашение для функции Python, которая принимает именованные аргументы при вызове. Соответствующий тип C — PyCFunctionWithKeywords. Функция C должна получать три аргумента типа PyObject*: self, args и словарь именованных аргументов. Если совмещать этот флаг с METH_VARARGS, то первые два аргумента имеют тот же смысл, что и при предыдущем соглашении вызова; в противном случае значением args будет NULL. Типичная сигнатура функции C: PyObject* function(PyObject* self, PyObject* args, PyObject* keywds).
- METH_NOARGS: соглашение для функций Python, которые не принимают никаких других аргументов. Тип функции C должен быть PyCFunction, так что сигнатура будет такой же, как у соглашения METH_VARARGS (с аргументами self и args). Единственное отличие заключается в том, что args всегда будет NULL, так что не требуется вызывать PyArg_ParseTuple(). Этот флаг нельзя совмещать с любыми другими флагами соглашений вызова.
- METH_O: сокращение для функций и методов, которые принимают в аргументах только один объект. Тип функции C — снова PyCFunction, так что она принимает два аргумента PyObject*: self и args. Отличие от METH_VARARGS состоит в том, что вызывать PyArg_ParseTuple() не нужно, потому что передаваемый в аргументах указатель PyObject* уже представляет единственный аргумент, который передается при вызове этой функции из Python. Этот флаг также нельзя совмещать с другими флагами соглашений вызова.

Функция, которая принимает именованные аргументы, описывается флагом METH_KEYWORDS или поразрядной комбинацией флагов METH_VARARGS | METH_KEYWORDS. В последнем случае она должна разбирать свои аргументы вызовом PyArg_ParseTupleAndKeywords(), а не PyArg_ParseTuple() или PyArg_UnpackTuple().

Рассмотрим пример модуля с одной функцией, которая возвращает None и принимает два именованных аргумента, направляемых в стандартный вывод:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

static PyObject* print_args(PyObject *self, PyObject *args,
 PyObject *keywds)
{
    char *first;
    char *second;
```

```
    static char *kwlist[] = {"первый", "второй", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "ss", kwlist,
                                     &first, &second))
        return NULL;

    printf("%s %s\n", first, second);

    Py_INCREF(Py_None);
    return Py_None;
}

static PyMethodDef module_methods[] = {
    {"print_args", (PyCFunction)print_args,
     METH_VARARGS | METH_KEYWORDS,
     "выводит принятые аргументы"},
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef module_definition = {
    PyModuleDef_HEAD_INIT,
    "kwargs",
    "Пример обработки именованных аргументов",
    -1,
    module_methods
};


PyMODINIT_FUNC PyInit_kwargs(void) {
    return PyModule_Create(&module_definition);
}
```

Синтаксический разбор аргументов в Python/C API чрезвычайно гибок. Он подробно описан в официальной документации по адресу https://docs.python.org/3/c-api/arg.html.

Аргумент формата при вызове `PyArg_ParseTuple()` и `PyArg_ParseTupleAndKeywords()` позволяет детально управлять количеством и типом аргументов. С этим API на C можно закодировать любое нетривиальное соглашение вызова, известное в Python, в том числе:

- функции со значениями по умолчанию для аргументов;
- функции только с именованными аргументами;
- функции только с позиционными аргументами;
- функции с переменным числом аргументов;
- функции без аргументов.

Донолнительные флаги соглашений связывания `METH_CLASS`, `METH_STATIC` и `METH_COEXIST` зарезервированы для методов, и их нельзя использовать для описания функций модулей. Первые два флага не требуют объяснений. Они являются аналогами языка C для декораторов `@classmethod` и `@staticmethod` и меняют смысл аргумента `self`, который передается функции C.

Флаг `METH_COEXIST` позволяет загрузить метод вместо существующего определения. Он пригождается очень редко — в основном тогда, когда вы хотите предоставить реализацию метода C, которая генерируется автоматически на основе других характеристик определяемого типа. В документации Python приводится пример метода-обертки `__contains__()`, который генерируется, если в типе определен слот `sq_contains`. К сожалению, определение собственных классов и типов с помощью Python/C API выходит за рамки этой вводной главы.

Пример обработки исключений рассматривается в следующем разделе.

Обработка исключений

В отличие от Python и даже C++, в языке C нет синтаксиса, который позволял бы порождать и обрабатывать исключения. Вся обработка ошибок обычно выполняется на уровне возвращаемых значений функций, а также с помощью необязательного глобального состояния, которое хранит подробную информацию о причинах последнего сбоя.

Обработка исключений в Python C/API строится на этом простом принципе. У каждого потока есть глобальный индикатор последней ошибки. Ему присваивается значение, описывающее причину проблемы. Также существует стандартизированный способ уведомить вызывающую сторону о том, изменилось ли это состояние во время вызова, например:

- Если функция должна вернуть указатель, она возвращает `NULL`.
- Если функция должна вернуть значение типа `int`, она возвращает `-1`.

Единственное исключение из этого правила в Python/C API — функции `PyArg_*()`, которые возвращают 1 в случае успеха и 0 в случае неудачи.

Чтобы понять, как этот механизм работает на практике, вспомните функцию `fibonacci_py()` из предыдущего примера:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
        result = Py_BuildValue("L", fibonacci((unsigned int) n));
```

```
    }

    return result;
}
```

Обработка ошибок начинается в самом начале функции, когда инициализируется переменная `result`. Предполагается, что в этой переменной хранится возвращаемое значение функции. Она инициализируется значением `NULL`, которое, как вы уже знаете, является признаком ошибки. Это обычная практика при разработке расширений — предполагать, что ошибка является состоянием вашего кода по умолчанию.

Далее следует вызов `PyArg_ParseTuple()`, который сохраняет информацию об ошибке в случае исключения и возвращает `0`. Это часть инструкции `if`, так что в случае исключения больше ничего не происходит и функция возвращает `NULL`. Сторона, которая вызвала функцию, получит уведомление об ошибке.

`Py_BuildValue()` тоже может порождать исключение. Эта функция должна возвращать `PyObject*` (указатель), так что в случае неудачи она возвращает `NULL`. Это значение можно просто сохранить в переменной `result` и передать его дальше в инструкции `return`.

Однако наша задача не ограничивается обработкой исключений, которые порождаются вызовами Python/C API. Весьма вероятно, что вам понадобится сообщить пользователю расширения о том, какая именно ошибка (или сбой) произошла. В Python/C API есть несколько функций, которые помогают порождать исключения, но самая распространенная из них — `PyErr_SetString()`. Она устанавливает индикатор ошибки с заданным типом исключения и дополнительной строкой, которая объясняет причину ошибки. Полная сигнатура этой функции выглядит так:

```
void PyErr_SetString(PyObject* type, const char* message)
```

Возможно, вы уже заметили проблему в функции `fibonacci_py()` из раздела «Детальное ознакомление с Python/C API». А если нет, сейчас самое время найти и исправить ее. К счастью, у нас наконец-то появились подходящие средства для этого.

Проблема связана с небезопасным преобразованием типа `long` в `unsigned int` в следующих строках:

```
if (PyArg_ParseTuple(args, "l", &n)) {
    result = Py_BuildValue("L", fibonacci((unsigned int) n));
}
```

Благодаря вызову `PyArg_ParseTuple()` первый и единственный аргумент интерпретируется как тип `long` (спецификатор `"l"`) и сохраняется в локальной пере-

менной n. Затем он приводится к типу `unsigned int`, так что возникнут проблемы, если функцию `fibonacci()` вызвать из Python с отрицательным значением. Например, 32-разрядное целое число со знаком `-1` при преобразовании в беззнаковое 32-разрядное целое число превратится в `4294967295`. Такое значение породит очень глубокую рекурсию, что приведет к переполнению стека и ошибке сегментации. То же самое может произойти, если пользователь передаст очень большое положительное число. Это нельзя исправить без полной переработки функции C `fibonacci()`, но можно по крайней мере проверить, что входной аргумент функции удовлетворяет некоторым предусловиям. Здесь мы проверяем, что значение аргумента n больше либо равно 0, и порождаем исключение `ValueError`, если условие не выполняется:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;
    long long fib;

    if (PyArg_ParseTuple(args, "l", &n)) {
        if (n<0) {
            PyErr_SetString(PyExc_ValueError,
                            "n must not be less than 0");
        } else {
            result = Py_BuildValue("L", fibonacci((unsigned int) n));
        }
    }

    return result;
}
```

И последнее замечание про обработку исключений: глобальное состояние ошибки не сбрасывается само собой. Некоторые ошибки можно корректно обрабатывать в функциях C (как это делается в Python с помощью конструкции `try... except`), и если индикатор ошибки стал недействительным, его нужно сбросить. Для этого используется функция `PyErr_Clear()`.

Одно из огромных преимуществ расширений на C заключается в том, что они позволяют обходить глобальную блокировку GIL, которая может губительно сказаться на быстродействии многопоточного конкурентного выполнения в приложениях Python. Следующий раздел посвящен тому, как избавиться от GIL в расширениях на C.

Как обойти GIL

Ранее уже упоминалось, что расширения дают возможность обойти блокировку GIL в Python. Знаменитое ограничение реализации CPython заключается в том, что в каждый момент времени код Python может выполняться только одним

потоком. Чтобы обойти это ограничение, рекомендуется применять многопроцессное выполнение (см. главу 6), но оно может оказаться не лучшим решением для некоторых алгоритмов с высокой степенью параллелизма, потому что потребует расходовать ресурсы на запуск дополнительных процессов.

Поскольку расширения используются в основном там, где бóльшая часть работы выполняется на чистом C без вызовов Python/C API, возможно (и даже желательно) снимать GIL в некоторых разделах приложения во время обработки данных, которая не связана с Python. Благодаря этому можно извлечь пользу из многоядерных процессоров или многопоточной архитектуры приложения. Единственное, что для этого потребуется, — обернуть блоки кода, которые заведомо не используют вызовы Python/C API или структуры Python, в специальные макросы Python/C API. Следующие два макроса препроцессора упрощают всю процедуру снятия и повторной установки GIL:

- `Py_BEGIN_ALLOW_THREADS`: объявляет скрытую локальную переменную, в которой сохраняется текущее состояние потока, и снимает GIL.
- `Py_END_ALLOW_THREADS`: заново устанавливает GIL и восстанавливает состояние потока из локальной переменной, которую объявил предыдущий макрос.

Если внимательно присмотреться к расширению `fibonacci`, легко увидеть, что функция `fibonacci()` не выполняет никакого кода Python и не притрагивается к структурам Python. А это означает, что функцию `fibonacci_py()`, которая просто инкапсулирует выполнение `fibonacci(n)`, можно модифицировать так, чтобы она снимала GIL на время этого вызова:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;
    long long fib;

    if (PyArg_ParseTuple(args, "l", &n)) {
        if (n<0) {
            PyErr_SetString(PyExc_ValueError,
                            "n must not be less than 0");
        } else {
            Py_BEGIN_ALLOW_THREADS;
            fib = fibonacci(n);
            Py_END_ALLOW_THREADS;

            result = Py_BuildValue("L", fib);
        }
     }

    return result;
}
```

Другая важная тема, относящаяся к Python/C API, — управление памятью и сборка мусора. Самый распространенный механизм сборки мусора в динамических языках программирования — **трассировочиая сборка мусора**, когда сборщик мусора отслеживает, можно ли добраться до объектов из корневой ссылки программы. Если объекты становятся недостижимыми, можно освободить память, которую опи занимают.

В Python есть примитивный трассировочный сборщик мусора, который находит циклические ссылки, однако в качестве основного механизма управлепия памятью используется счетчик ссылок. Это не создает проблем в обычном коде на Python, но существепно увеличивает объем работы при нанисании расширений на C. Эта тема подробнее рассматривается в следующем разделе.

Стратегия трассировочной сборки мусора распространена настолько широко, что ее часто считают синонимом сборки мусора вообще. Из-за этого одни утверждают, будто в Python вовсе нет сборки мусора (потому что основным методом управления памятью в нем является счетчик ссылок), а другие настаивают на обратном (потому что трассировка используется для поиска циклических ссылок, а счетчик ссылок можно рассматривать как альтернативную стратегию сборки мусора).

Счетчик ссылок

Мы наконец подошли к важной теме управления памятью в Python. В Python есть собственный сборщик мусора, но он предназначен только для того, чтобы решать проблему циклических ссылок в алгоритме счетчика ссылок. Счетчик ссылок — основпой метод управления утилизацией объектов, которые больше не используются.

Чтобы объяснить, как в Python/C API устроена утилизация объектов, в документации вводится попятие **прииадлежиости ссылок** (ownership of references). Объекты в Python никогда не принадлежат коду расширепия, и поэтому само расширение не может их создавать или уничтожать. Фактическим созданием объектов управляет диспетчер памяти Python. Вот почему мы говорим, что объекты в Python нринадлежат диспетчеру памяти.

Диспетчер памяти — внутренпий компонент интерпретатора CPython; только он может выделять и освобождать память для объектов, которые хранятся в частной куче (private heap). Однако принадлежать может не сам объект, а ссылка на него.

С каждым объектом в Python, который представлен ссылкой (указателем `PyObject*`), связывается счетчик ссылок. Когда он уменьшается до нуля, это означает, что нигде в программе больше нет действительных ссылок на этот

объект и можно запустить процедуру освобождения памяти, связанную с этим типом. В Python/C API есть несколько макросов, которые увеличивают и уменьшают счетчики ссылок:

- `Py_INCREF()` и `Py_DECREF()`: первый макрос увеличивает счетчик ссылок, второй уменьшает его. Эти макросы принимают указатели на объекты, которые должны быть отличны от `NULL`.
- `Py_XINCREF()` и `Py_XDECREF()`: первый макрос увеличивает счетчик ссылок, второй уменьшает его. Эти макросы могут припимать `NULL`, поэтому их следует использовать тогда, когда вы не уверены в том, что указатели отличны от `NULL`.

Но прежде чем углубляться в подробности, нужно усвоить следующие термины, относящиеся к принадлежности ссылок:

- **Передача прииадлежиости** (passing of ownership): когда мы говорим, что функция передает припадлежность ссылки, это означает, что она уже увеличила счетчик ссылок и теперь вызывающая сторона должна уменьшить счетчик, когда ссылка па объект будет больше не нужна. Так делают многие функции, которые возвращают только что созданные объекты, например `Py_BuildValue`. Если такой объект возвращается из нашей функции другой вызывающей стороне, то принадлежность снова передается. В этом случае не нужно уменьшать счетчик ссылок, потому что это уже не наша обязанпость. Вот почему функция `fibonacci_py()` не вызывает `Py_DECREF()` для переменной `result`.
- **Заимствовапиые ссылки** (borrowed references): заимствование ссылок происходит, когда функция принимает в качестве аргумента ссылку на объект Python. В этой фупкции счетчик ссылок для такой ссылки никогда не должен умепьшаться, если только он не был явно увеличен в ее области видимости. В нашей функции `fibonacci_py()` аргументы `self` и `args` являются такими заимствованными ссылками, поэтому для них не вызывается `PyDECREF()`. Некоторые функции Python/C API также могут возвращать заимствованные ссылки: характерпые примеры такого рода — `PyTuple_GetItem()` и `PyList_GetItem()`. Часто такие ссылки называют незащищенными. Избавляться от их принадлежности не нужно, если только они не возвращаются как значение функции. В большинстве случаев нужна особая осторожность, если такие заимствованные ссылки используются в качестве аргументов других вызовов Python/C API. В некоторых ситуациях приходится дополнительно защищать эти ссылки отдельным вызовом `Py_INCREF()`, прежде чем использовать их как аргументы других функций, а затем вызвать `Py_DECREF()`, когда они больше не нужны. Соответствующий пример приведен в конце раздела.
- **Крадепые ссылки** (stolen references): также функции Python/C API могут красть ссылки, вместо того чтобы заимствовать их, когда они передаются

в аргументе вызова. Это относится ровно к двум функциям — `PyTuple_SetItem()` и `PyList_SetItem()`. Эти функции принимают полную ответственность за ссылки, которые им передаются. Они не увеличивают счетчик ссылок сами по себе, но вызывают `Py_DECREF()`, когда ссылка больше не нужна.

Следить за счетчиком ссылок — одпа из самых трудных задач при написании сложных расширений. Некоторые неочевидные проблемы могут остаться незамеченными, пока код не начнет выполняться в многопоточной конфигурации.

Другая тиничная проблема обусловлена самой природой объектной модели Python и тем фактом, что некоторые функции возвращают заимствованные ссылки. Когда счетчик ссылок уменьшается до нуля, выполняется функция освобождения памяти. Для пользовательских классов можно онределить метод `__del__()`, который будет вызываться в этот момент.

Метод может содержать произвольный код Python, так что он может влиять на другие объекты и их счетчики ссылок. В официальной документации Python нриводится следующий пример кода, где проявляется эта нроблема:

```
void bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* ОШИБКА! */
}
```

Код выглядит абсолютно безобидно, но проблема в том, что мы не знаем, какие элемепты содержит объект `list`. Когда `PyList_SetItem()` нрисваивает новое значение элементу `list[1]`, теряется нринадлежность объекта, который рапьше хранился по этому индексу. Если это была единственная существующая ссылка, то счетчик ссылок уменьшается до 0 и объект может утилизироваться. Может оказаться, что это был некий пользовательский класс с собственной реализацией метода `__del__()`. Возникнут серьезные проблемы, если в результате выполнения этого `__del__()` из списка будет удален элемент `item[0]`.

Обратите внимание: функция `PyList_GetItem()` возвращает заимствованную ссылку! Она не вызывает `Py_INCREF()` перед тем, как возвращать ссылку. Таким образом, в этом коде нельзя исключать, что `PyObject_Print()` будет вызвана со ссылкой на объект, которого больше нет. Это приведет к ошибке сегментации и фатальпому сбою интерпретатора Python.

Поэтому следует защищать заимствованные ссылки на все время, когда они нужны, так как есть вероятность того, что какой-нибудь промежуточный вызов

приведет к утилизации объекта. Это может произойти, даже если на первый взгляд ссылки пе связаны с этим вызовом, как в следующем примере:

```
void no_bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

Как видите, писать расширения Python на C с помощью Python/C API может оказаться непростым делом, особенпо если у вас нет опыта программирования на C. Здесь нужно хорошо разбираться во внутреннем устройстве CPython и тонкостях управления памятью. Но, к счастью, существует более простой способ. Речь идет о Cython — специальном диалекте Python, который рассматривается в следующем разделе.

Разработка расширений на Cython

Именем Cython обозначается как оптимизирующий статический компилятор, так и язык программирования, являющийся надмножеством Python. Оп помогает ускорять приложения на Python, компилируя их в машипный код, а также служит «языком-оберткой» для кода, нанисанного на C и C++.

Как компилятор, Cython транспилирует «родпой» код па Python и Cython в расширения на C для Python, которые используют Python/C API. Это позволяет объединить мощь Python и C без необходимости вручную работать с Python/C API.

Как надмножество Python, Cython позволяет использовать статическую типизацию и статическую компоновку библиотек C (в отличие от динамической компоновки общих библиотек), взаимодействовать с заголовочными файлами C и папрямую управлять блокировкой GIL в CPython.

Для начала обсудим Cython как транспилятор.

Cython как транспилятор

Главное преимущество расширений, созданных с помощью Cython, — это использование соответствующего надмножества языка. Впрочем, можно создавать расширепия и из обычного кода Python, применяя транспиляцию (компиляцию «исходный код — исходный код»). Это простейший вариант использования

Cython, потому что он почти не требует изменений в коде и может заметно улучшить быстродействие с минимальными затратами.

Чтобы создавать расширения на Cython, вам прежде всего понадобится пакет Cython. Его можно установить из PyPI с помощью `pip`:

```
$ python3 -m pip install Cython
```

В Cython есть простая вспомогательная функция `cythonize`, которая позволяет легко интегрировать процесс компиляции с пакетом `setuptools`. Допустим, вы хотите откомпилировать реализацию функции `fibonacci()`, написанную на чистом Python, в расширение на C. Если она находится в модуле fibonacci.py, то минимальный сценарий setup.py может выглядеть так:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name='fibonacci',
    ext_modules=cythonize(['fibonacci.py'])
)
```

Такой модуль можно установить через `pip` так же, как обычное расширение на C:

```
$ python3 -m pip install -e .
Installing collected packages: fibonacci
  Running setup.py develop for fibonacci
Successfully installed fibonacci
```

Эта команда устанавливает пакет в режиме редактирования, благодаря чему можно просмотреть файлы, сгенерированные в процессе установки. Выполнив ее в своей командной оболочке, вы увидите, что она также создает дополнительные артефакты сборки:

```
$ ls -1ap
./
../
build/
fibonacci.c
fibonacci.cpython-39-darwin.so
fibonacci.egg-info/
fibonacci.py
setup.py
```

Файл fibonacci.c в этом выводе — это автоматически сгенерированный код расширения на C. Cython преобразовывает (транспилирует) обычный код на Python в код на C. В процессе установки этот код на C будет использоваться, чтобы

сформировать библиотеку модуля расширения; в нашем случае это файл fibonacci.cpython-39-darwin.so.

Чтобы понять, какой объем работы Cython выполняет «под капотом», просмотрите файл fibonacci.c. Он получается довольно длинным. Даже для нашего простого модуля fibonacci.py его длина может превышать 4000 строк.

Если использовать Cython для того, чтобы компилировать исходный код на Python, это дает еще одно преимущество. В процессе установки пакета не всегда обязательно транспилировать его в расширение. Если в среде, где его надо установить, нет Cython или других предусловий сборки, возможна установка в виде обычного пакета Python. При такой модели распространения пользователь, как правило, не заметит никаких функциональных различий в поведении кода. Стандартная практика распространения расширений на Cython состоит в том, чтобы включать в дистрибутив как исходные файлы на Python/Cython, так и код C, сгенерированный из этих исходных файлов.

При таком подходе пакет можно установить тремя разными способами в зависимости от предусловий сборки:

- Если в среде установки доступен Cython, код расширения на C генерируется на основе предоставленного исходного кода на Python/Cython.
- Если Cython недоступен, но доступны предусловия сборки (компилятор C, заголовки Python/C API), расширение собирается на основе предварительно сгенерированных файлов C, которые входят в дистрибутив.
- Если ни один из этих вариантов не доступен, а расширение состоит из исходного кода на чистом Python, то модули устанавливаются как обычный код Python, а этап компиляции пропускается.

В документации Cython сказано, что при распространении расширений Cython рекомендуется включать как сгенерированные файлы на C, так и исходный код на Cython. Согласно той же документации, компиляцию Cython нужно отключать по умолчанию, потому что в окружении пользователя может не оказаться необходимой версии Cython, что приведет к непредвиденным проблемам при компиляции.

С официальными рекомендациями по распространению кода на Cython можно ознакомиться по адресу https://cython.readthedocs.io/src/userguide/source_files_and_compilation.html.

Как бы то ни было, с появлением изолированных сред эта проблема, похоже, стала намного менее актуальной. Кроме того, Cython доступен в PyPI как

нолноценный пакет Python, поэтому можно легко определить его конкретную версию как требование проекта. Конечно, решение о таком предусловии влечет серьезные последствия и должно быть тщательно продумано. Другой, более безопасный вариант — использовать функциональность `extras_require` из пакета `setuptools`, чтобы пользователь мог сам решить, хочет ли он использовать Cython с конкретной переменной среды:

```
import os

from setuptools import setup, Extension

try:
    # транспиляция Cython возможна,
    # только если Cython доступен
    # и специальная переменная среды явно сообщает,
    # что нужно использовать Cython,
    # чтобы генерировать исходный код на C
    USE_CYTHON = bool(os.environ.get("USE_CYTHON"))
    import Cython

except ImportError:
    USE_CYTHON = False

ext = '.pyx' if USE_CYTHON else '.c'

extensions = [Extension("fibonacci", ["fibonacci"+ext])]

if USE_CYTHON:
    from Cython.Build import cythonize
    extensions = cythonize(extensions)

setup(
    name='fibonacci',
    ext_modules=extensions,
    extras_require={
        # Эта конкретная версия Cython будет указана
        # в числе требований, если устанавливать пакет
        # с дополнительным ключом '[with-cython]'
        'with-cython': ['cython==0.29.22']
    }
)
```

`pip` поддерживает установку пакетов с дополнениями (`extras`), для чего к имени пакета добавляется суффикс `[имя_дополнения]`. Для приведенного примера можно запустить следующую команду, чтобы активировать необязательные требования к версии Cython и компиляцию из локальных источников во время установки:

```
$ USE_CYTHON=1 pip install .[with-cython]
```

Переменная окружения `USE_CYTHON` гарантирует, что `pip` будет использовать Cython, чтобы компилировать исходные коды .pyx в C, а ключ `[with-cython]` — что компилятор Cython будет загружен перед установкой.

Хотя с помощью Cython можно компилировать код на обычном Python, гораздо эффективнее будет использовать диалект Cython. Он поддерживает ряд дополнительных возможностей, которые недоступны в обычном коде на Python. Специфические средства Cython рассматриваются в следующем разделе.

Cython как язык

Cython — не только компилятор, но и надмножество языка Python. Термин «надмножество» означает, что Cython не только поддерживает любой действительный код Python, но и способен расширять его новыми возможностями, такими как вызов функций C или объявление типов C для переменных и атрибутов классов. Таким образом, любой код на Python также является кодом на Cython, но обратное не всегда верно. Это объясняет, почему обычные модули Python так легко компилируются в C с помощью компилятора Cython.

Но мы не остановимся на этом простом факте. Вместо того чтобы просто сказать, что наша эталонная функция `fibonacci()` также является кодом на Cython, мы попытаемся ее немного усовершенствовать. Это не приведет к реальной оптимизации, потому что мы по-прежнему реализуем вычисление чисел Фибоначчи в рекурсивном виде. Но мы внесем небольшие изменения, чтобы извлечь больше пользы от того, что код написан на Cython.

Для исходного кода Cython используется другое расширение файла — .pyx вместо .py. Содержимое файла fibonacci.pyx может выглядеть так:

```
"""Модуль Cython, реализующий числа Фибоначчи."""

def fibonacci(unsigned int n):
    """Возвращает n-е число Фибоначчи, вычисленное рекурсивно."""
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

По сути, изменилась только сигнатура функции `fibonacci()`. Благодаря необязательной статической типизации в Cython можно объявить аргумент `n` с типом `unsigned int`, и от этого функция будет работать немного эффективнее. Есть и дополнительный эффект: если аргумент функции Cython объявлен со статическим тином, то расширение будет автоматически обрабатывать ошибки преобразования и переполнения, порождая соответствующие исключения. Следу-

ющий фрагмент интерактивного сеанса показывает, как это происходит с функцией `fibonacci()`, написанной на Cython:

```
>>> from fibonacci import fibonacci
>>> fibonacci(5)
5
>>> fibonacci(0)
0
>>> fibonacci(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 4, in fibonacci.fibonacci
    def fibonacci(unsigned int n):
OverflowError: can't convert negative value to unsigned int
>>> fibonacci(10 ** 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 4, in fibonacci.fibonacci
    def fibonacci(unsigned int n):
OverflowError: value too large to convert to unsigned int
```

Мы уже знаем, что Cython выполняет только транспиляцию (преобразование исходного кода в исходный код), а генерируемый код использует тот же Python/C API, с которым мы бы работали, если бы вручную писали расширения на C. Учтите, что функция `fibonacci()` является рекурсивной и поэтому очень часто вызывает саму себя. А значит, несмотря на то что мы объявили статический тип для входного аргумента, во время рекурсивного вызова функция будет рассматривать себя как любую другую функцию Python. Таким образом, `n-1` и `n-2` будут упаковываться обратно в объект Python, а затем передаваться скрытому уровню-обертке внутренней реализации `fibonacci()`, которая вернет их к типу `unsigned int`. Это будет повторяться снова и снова, пока программа не достигнет максимальной глубины рекурсии. Такая схема не обязательно является проблемой, но требует значительно большей обработки аргументов, чем действительно необходимо.

Чтобы сократить лишние затраты на вызовы функций Python и обработку аргументов, можно делегировать больше работы функции на чистом C, которая ничего не знает о структурах Python. Мы занимались этим раньше, когда создавали расширения на чистом C, и то же самое можно сделать в Cython. С помощью ключевого слова `cdef` объявляются функции в стиле C, которые принимают и возвращают только типы C:

```
cdef long long fibonacci_cc(unsigned int n):
    if n == 0:
        return 0
    if n == 1:
```

```
        return 1
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)

def fibonacci(unsigned int n):
    """ Возвращает n-е число Фибоначчи, вычисленное рекурсивно.
    """
    return fibonacci_cc(n)
```

Функция `fibonacci_cc()` не будет доступна для импортирования в итоговом скомпилированном модуле `fibonacci`. Вместо этого функция `fibonacci()` формирует фасад к низкоуровневой реализации `fibonacci_cc()`.

Можно пойти еще дальше. В примере с обычным C мы показали, как снимать GIL на время вызова фупкции на чистом C, чтобы расширение лучше вело себя в многопоточпых нриложениях. В предыдущих примерах использовались макросы препроцессора `Py_BEGIN_ALLOW_THREADS` и `Py_END _ALLOW_THREADS` из заголовков Python/C API, чтобы пометить участки кода, в которых пет вызовов Python. Соответствующий синтаксис Cython немного компактнее и проще запоминается. Чтобы снять блокировку GIL вокруг участка кода, достаточно выполнить команду `nogil`:

```
def fibonacci(unsigned int n):
    """ Возвращает n-е число Фибоначчи, вычисленное рекурсивно.
    """
    with nogil:
        return fibonacci_cc(n)
```

Также можпо пометить всю функцию в стиле C как безопасную для вызова без GIL:

```
cdef long long fibonacci_cc(unsigned int n) nogil:
    if n < 2:
        return n
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)
```

Важно знать, что объекты Python не могут быть ни аргументами, ни возвращаемыми типами таких функций. Каждый раз, когда функции с пометкой `nogil` нужно выполнить какой-пибудь вызов Python/C API, она должна установить GIL инструкцией `gil`.

Вы уже умеете создавать расширения Python двумя способами: в виде кода на обычном C с использованием Python/C API и с помощью Cython. Первый вариант предоставляет больше мощи и гибкости ценой более сложного и подробного кода, а во втором случае писать расширения проще, но Cython выполняет бо́льшую часть «черной магии» незаметно для вас. Мы также рассмотрели не-

которые потенциальные достоинства расширений, ноэтому пришло время поближе нознакомиться с их потенциальными недостатками.

Недостатки расширений

Откровенно говоря, я перешел на Python только нотому, что мне надоели сложности написания программ на C и C++. На самом деле многие программисты берутся за изучение Python, когда понимают, что другие языки не позволяют осуществить то, что нужно их пользователям.

По сравнению с C, C++ или Java программирование на Python выглядит легкой прогулкой. Вроде бы все здесь устроено нросто и хорошо спроектировано. Начинает казаться, что тут вообще негде споткнуться, а другие языки программирования больше не нужны.

И конечно, это абсолютно не соответствует действительности. Безусловно, Python — замечательный язык со множеством интересных возможностей, и он востребован во многих областях. Но это не означает, что он идеален и у него нет недостатков. Код на Python легко писать и понимать, но за эту легкость приходится платить. Он выполняется не настолько медленно, как многие считают, однако никогда не догонит C. Python отличается высокой переносимостью, но его интерпретаторы доступны для меньшего количества архитектур, чем компиляторы некоторых других языков. И этот список можно было бы продолжить.

Чтобы преодолеть многие недостатки, можно разрабатывать расширения, которые позволяют задействовать в Python некоторые сильные стороны старого доброго C. В большинстве случаев этот подход неплохо работает. Вопрос в другом: разве мы пишем код на Python именно для того, чтобы расширять его с помощью C? Нет конечно. Это всего лишь неизбежное зло в ситуациях, где не удается придумать ничего лучше.

За расширения всегда приходится расплачиваться, и один из самых больших недостатков их использования — рост сложности.

Дополнительная сложность

Не секрет, что разрабатывать приложения на нескольких языках — непростая задача. Python и C — совершенно разные технологии, и между ними трудно найти что-то общее. Правда и в том, что не существует приложений, в которых вообще нет ошибок. Если активно использовать расширения в кодовой базе, отладка может стать весьма неприятной — не только из-за того, что отладка

кода на C требует совершенно иных рабочих процессов и инструментов, но и потому, что вам придется слишком часто переключать мышление между двумя языками.

Все мы люди, и наши когнитивные способности ограниченны. Конечно, некоторые представители человеческого рода способны эффективно работать с несколькими уровнями абстракции одновременно, но похоже, такие люди встречаются очень редко. Каким бы опытным разработчиком вы ни были, за сопровождение таких гибридных решений всегда приходится платить повышенную цену. Это могут быть дополнительные усилия и время, необходимые для переключения между C и Python, или же лишний стресс, от которого ваша работа со временем может стать менее эффективной.

По статистике TIOBE, C до сих пор остается одним из самых популярных языков программирования. Однако, несмотря на это, программисты на Python очень часто знают его на минимальном уровне или не знают вообще. Лично я считаю, что C должен быть универсальным языком общения в мире программирования, но мое мнение вряд ли что-нибудь изменит.

Python также настолько соблазнителен и прост в изучении, что многие программисты забывают весь предыдущий опыт и полностью переключаются на новую технологию. Но не стоит думать, будто программирование — это что-то вроде умения ездить на велосипеде. Этот навык очень легко теряется, если не пользоваться им и не поддерживать квалификацию. Даже программисты с серьезной подготовкой в области C рискуют постепенно утратить свое мастерство, если погрузятся в Python слишком надолго.

Все сказанное приводит к простому выводу: вам будет труднее найти людей, которые способны понять и расширить ваш код. Для пакетов с открытым кодом это означает меньшее количество добровольных участников. Для закрытого кода это означает, что не все ваши коллеги смогут разрабатывать и сопровождать расширения, ничего не сломав при этом. А отлаживать сломанные расширения определенно сложнее, чем код на обычном Python.

Более сложная отладка

Дефектные расширения способны создать кучу проблем. Казалось бы, у статической типизации есть множество преимуществ перед обычным кодом на Python, и она позволяет выявить на стадии компиляции многие проблемы, которые было бы трудно обнаружить в Python, причем это даже не потребует скрупулезного тестирования и полного тестового покрытия. Однако это лишь одна сторона монеты.

Другая сторона заключается в том, что памятью приходится управлять вручную. А некорректное управление памятью — главная причина большинства ошибок в программах на C. В лучшем случае такие ошибки приведут к утечкам намяти, которые постепенно исчерпают ресурсы вашей среды. Впрочем, хотя это и лучший случай, это не означает, что с ним легко снравиться. Утечки памяти очень трудно обнаружить без таких внешних инструментов, как Valgrind. Как правило, проблемы с управлением намятью в коде расширения приводят к ошибкам сегментации, с которыми невозможно бороться па уровне Python, а интерпретатор аварийно завершается, не норождая исключение, которое объясняло бы причину сбоя. Следовательно, в конце концов вам придется вооружиться дополнительными инструментами, которыми большинство программистов Python обычно не пользуются. От этого и среда разработки, и рабочие процессы становятся сложнее.

Недостатки расширений позволяют сделать вывод, что они не всегда оказываются онтимальным средством для интеграции Python с другими языками. Если вам нужно только взаимодействовать с уже готовыми общими библиотеками, иногда лучше помогает совершенно иной подход. В следующем разделе идет речь о том, как взаимодействовать с динамическими библиотеками, не используя расширения.

Взаимодействие с динамическими библиотеками без расширений

Благодаря `ctypes` (это модуль стандартной библиотеки) или `cffi` (это внешний пакет, доступный в PyPI) в Python можно интегрировать любую скомпилированную динамическую/общую библиотеку, на каком бы языке она ни была написана. И это можно сделать в коде на чистом Python без промежуточной комниляции. Эти два пакета, известные как **библнотеки внешиих функцнй**, — перспективная альтернатива написанию собственных расширений на C.

Хотя работа с библиотеками внешних функций не требует писать код на C, это не означает, что их можно эффективпо использовать, ничего не зная о C. Как `ctypes`, так и `cffi` требуют на некотором уровне разбираться в C и принципах работы динамических библиотек в целом. С другой стороны, они избавляют вас от бремени подсчета ссылок в Python и значительно снижают риск неприятных ошибок. Кроме того, взаимодействие с кодом C через `ctypes` или `cffi` — более переносимое решение, чем писать и компилировать модули расширений на C.

Начнем с модуля `ctypes`, который входит в стандартную библиотеку Python.

Модуль ctypes

ctypes — самый популярный модуль для вызова функций из динамических или общих библиотек без необходимости писать специальные расширения на C. Причина очевидна: этот модуль входит в стандартную библиотеку, поэтому он всегда под рукой и не требует никаких внешних зависимостей.

Чтобы использовать код общей библиотеки, ее сначала надо загрузить. Посмотрим, как это делается средствами ctypes.

Загрузка библиотек

В ctypes есть четыре типа загрузчиков динамических библиотек и два соглашения по их использованию. Динамические и общие библиотеки представляются классами ctypes.CDLL, ctypes.PyDLL, ctypes.OleDLL и ctypes.WinDLL. Различия между ними таковы:

- ctypes.CDLL: загружаемые общие библиотеки. Функции в этих библиотеках используют стандартное соглашение вызова и должны возвращать тип int. Блокировка GIL снимается на время вызова.
- ctypes.PyDLL: класс работает как ctypes.CDLL, но блокировка GIL не снимается на время вызова. После выполнения проверяется флаг ошибки Python, и если он был установлен во время выполнения, то порождается исключение. Это приносит пользу, только если загруженная библиотека напрямую вызывает функции из Python/C API или использует функции обратного вызова, которые могут содержать код на Python.
- ctypes.OleDLL: класс доступен только в Windows. Функции в этих библиотеках используют соглашение вызова Windows stdcall и возвращают специфический для Windows код HRESULT, который информирует о том, был ли вызов успешным. Если код указывает на ошибку, Python автоматически порождает исключение OSError.
- ctypes.WinDLL: класс доступен только в Windows. Функции в этих библиотеках используют соглашение Windows stdcall и по умолчанию возвращают значения типа int. Python не проверяет автоматически, указывают ли эти значения на ошибку.

Чтобы загрузить библиотеку, можно либо создать экземпляр одного из перечисленных классов с подходящими аргументами, либо вызвать функцию LoadLibrary() из подмодуля, связанного с конкретным классом:

- ctypes.cdll.LoadLibrary() для ctypes.CDLL
- ctypes.pydll.LoadLibrary() для ctypes.PyDLL

- `ctypes.windll.LoadLibrary()` для `ctypes.WinDLL`
- `ctypes.oledll.LoadLibrary()` для `ctypes.OleDLL`

Основная проблема при загрузке общих библиотек — как найти их переносимым способом. В разных системах для общих библиотек используются разные суффиксы (.dll в Windows, .dylib в macOS, .so в Linux), и находятся они в разных местах. Главный нарушитель в этой области — Windows, где не существует заранее определенной схемы именования библиотек. Поэтому мы не будем обсуждать подробности загрузки библиотек с помощью модуля `ctypes` в этой системе, а сосредоточимся в основном на Linux и macOS, где эта проблема решается последовательно и в целом единообразно.

Если вас интересует Windows, обратитесь к официальной документации `ctypes`, в которой приведено достаточно информации о поддержке этой системы. Документация доступна по адресу https://docs.python.org/3/library/ctypes.html.

Обе схемы загрузки библиотек (функции `LoadLibrary()` и классы конкретных разновидностей библиотек) требуют указывать полные имена библиотек. Это означает, что имя должно содержать все предопределенные префиксы и суффиксы. Например, чтобы загрузить стандартную библиотеку C в Linux, нужно выполнить следующие инструкции:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.so.6')
<CDLL 'libc.so.6', handle 7f0603e5f000 at 7f0603d4cbd0>
```

Для macOS код будет выглядеть так:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.dylib')
```

К счастью, в подмодуле `ctypes.util` есть функция `find_library()`, которая позволяет загрузить библиотеку по имени без префиксов или суффиксов и будет работать в любой системе с предопределенной схемой именования общих библиотек:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> ctypes.cdll.LoadLibrary(find_library('c'))
<CDLL 'libc.so.6', handle 7f2e82f12000 at 0x7f2e8288e220>
>>> ctypes.cdll.LoadLibrary(find_library('bz2'))
<CDLL 'libbz2.so.1.0', handle 55fb3c2d1660 at 0x7f2e827e8af0>
```

Таким образом, если вы пишете пакет `ctypes`, который должен работать в macOS и Linux, всегда используйте `ctypes.util.find_library()`.

Когда общая библиотека загружена, можно использовать ее функции. Вызов функций C средствами `ctypes` рассматривается в следующем разделе.

Вызов функций C с помощью ctypes

Когда динамическая/общая библиотека успешно загружена в объект Python, ее обычно сохраняют в переменной уровня модуля, которая называется так же, как библиотека. К функциям можно обращаться как к атрибутам объекта, так что они вызываются так же, как функции Python, из любого другого импортированного модуля, например:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> libc = ctypes.cdll.LoadLibrary(find_library('c'))
>>> libc.printf(b"Hello world!\n")
Hello world!
13
```

К сожалению, все встроенные типы Python, кроме целых чисел, строк и байтов, несовместимы с типами данных C, и поэтому их нужно оборачивать в соответствующие классы, предоставляемые модулем `ctypes`. Вот полный список совместимых типов данных из документации `ctypes`:

Тип ctypes	Тип C	Тип Python
`c_bool`	`_Bool`	`bool`
`c_char`	`char`	1-символьный `bytes`
`c_wchar`	`wchar_t`	1-символьный `string`
`c_byte`	`char`	`int`
`c_ubyte`	`unsigned char`	`int`
`c_short`	`short`	`int`
`c_ushort`	`unsigned short`	`int`
`c_int`	`int`	`int`
`c_uint`	`unsigned int`	`int`
`c_long`	`long`	`int`
`c_ulong`	`unsigned long`	`int`

(*окончание*)

Тип ctypes	Тип C	Тип Python
c_longlong	__int64 или long long	int
c_ulonglong	unsigned __int64 или long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t или Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (с завершающим NULL)	bytes или None
c_wchar_p	wchar_t* (с завершающим NULL)	string или None
c_void_p	void*	int или None

Как видите, в приведенной таблице нет специализированных типов, которые представляли бы коллекции Python в виде массивов C. Чтобы создать тип для массива C, рекомендуется просто использовать оператор умножения с нужным базовым типом `ctypes`:

```
>>> import ctypes
>>> IntArray5 = ctypes.c_int * 5
>>> c_int_array = IntArray5(1, 2, 3, 4, 5)
>>> FloatArray2 = ctypes.c_float * 2
>>> c_float_array = FloatArray2(0, 3.14)
>>> c_float_array[1]
3.140000104904175
```

Этот синтаксис работает для всех базовых типов `ctypes`.

В следующем разделе мы поговорим о том, как передавать функции Python в виде обратных вызовов C.

Передача функций Python в виде обратных вызовов C

Очень популярный паттерн проектирования — делегировать часть работы реализации функции пользовательским обратным вызовам. Самая известная функция стандартной библиотеки C, получающая такие обратные вызовы, — `qsort()`, которая предоставляет обобщенную реализацию алгоритма быстрой

сортировки. Маловероятно, что вам понадобится этот алгоритм вместо стандартной сортировки `TimSort`, которая реализована в интерпретаторе CPython и лучше подходит для сортировки коллекций Python. Однако `qsort()` считается каноническим примером эффективного алгоритма сортировки, а во многих книгах по нрограммированию рассматривается C API, который использует механизм обратного вызова. Поэтому мы рассмотрим `qsort()` как пример нередачи фупкции Python в качестве обратного вызова C.

Обычный тип функции Python несовместим с типом функции обратного вызова, которого требует снецификация `qsort()`. Вот сигнатура `qsort()` из справочной страницы BSD, где также указап тип принимаемого обратного вызова (аргумент `compar`):

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void*, const void *));
```

Таким образом, чтобы выполпить `qsort()` из `libc`, нужно передать следующие аргументы:

- `base`: сортируемый массив в виде указателя `void*`.
- `nel`: количество элементов в виде значепия `size_t`.
- `width`: размер одного элемента массива в виде значения `size_t`.
- `compar`: указатель на функцию, которая должна возвращать `int` и принимает два указателя `void*`. Функция сравнивает два сортируемых элемента.

Из раздела «Вызов функций C с помощью ctypes» вы уже знаете, как сконструировать массив C из других типов `ctypes` с помощью оператора умножения. Значение `nel` должно иметь тин `size_t`, что соответствует типу `int` в Python, поэтому оно не требует дополнительпой обертки и может передаваться в виде `len(iterable)`. Зпачение `width` можно получить с помощью функции `ctypes.sizeof()`, если известен тип базового массива. Осталось узнать, как создать указатель на функцию Python, совместимую с аргументом `compar`.

В модуле `ctypes` есть фабричная функция `CFUNCTYPE()`, которая позволяет оборачивать функции Python и представлять их в виде указателей на вызываемые функции C. Первый аргумент содержит тип C, который должна возвращать оберпутая функция C.

Далее следует неременный список типов C, которые функция получает в аргументах. Тип функции, совместимый с аргумептом `compar` функции `qsort()`, выглядит так:

```
CMPFUNC = ctypes.CFUNCTYPE(
    # возвращаемый тип
    ctypes.c_int,
    # тип первого аргумента
```

```
    ctypes.POINTER(ctypes.c_int),
    # тип второго аргумента
    ctypes.POINTER(ctypes.c_int),
)
```

Функция CFUNCTYPE() использует соглашение вызова cdecl, поэтому она совместима только с общими библиотеками CDLL и PyDLL. Динамические библиотеки Windows, которые загружаются с помощью WinDLL или OleDLL, используют соглашение вызова stdcall. Это означает, что для того, чтобы оборачивать функции Python как указатели на вызываемые функции C, надо использовать другую фабрику. В ctypes это функция WINFUNCTYPE().

Чтобы подвести черту, допустим, что мы хотим отсортировать случайно перемешанный список целых чисел функцией qsort() из стандартной библиотеки C. Следующий пример показывает, как это сделать с использованием всего, что вы узнали ранее о ctypes:

```
from random import shuffle

import ctypes
from ctypes.util import find_library

libc = ctypes.cdll.LoadLibrary(find_library('c'))

CMPFUNC = ctypes.CFUNCTYPE(
    # возвращаемый тип
    ctypes.c_int,
    # тип первого аргумента
    ctypes.POINTER(ctypes.c_int),
    # тип второго аргумента
    ctypes.POINTER(ctypes.c_int),
)

def ctypes_int_compare(a, b):
    # аргументы являются указателями, поэтому используется индекс [0]
    print(" %s cmp %s" % (a[0], b[0]))

    # согласно спецификации qsort, функция должна возвращать:
    # * отрицательное значение, если a < b
    # * ноль, если a == b
    # * положительное значение, если a > b
    return a[0] - b[0]

def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("перемешанный массив: ", numbers)
```

```
    # создать новый тип, представляющий массив
    # такой же длины, как список numbers
    NumbersArray = ctypes.c_int * len(numbers)
    # создать новый массив C на основе нового типа
    c_array = NumbersArray(*numbers)

    libc.qsort(
        # указатель на отсортированный список
        c_array,
        # длина массива
        len(c_array),
        # размер одного элемента массива
        ctypes.sizeof(ctypes.c_int),
        # обратный вызов (указатель на функцию сравнения C)
        CMPFUNC(ctypes_int_compare)
    )
    print("отсортированный массив:   ", list(c_array))

if __name__ == "__main__":
    main()
```

Функция сравнения, переданная в виде функции обратного вызова, содержит дополнительную инструкцию `print`, чтобы можно было посмотреть, как выполняется сравнение в процессе сортировки:

```
$ python3 ctypes_qsort.py
перемешанный массив:  [4, 3, 0, 1, 2]
 4 cmp 3
 4 cmp 0
 3 cmp 0
 4 cmp 1
 3 cmp 1
 0 cmp 1
 4 cmp 2
 3 cmp 2
 1 cmp 2
отсортированный массив:    [0, 1, 2, 3, 4]
```

Конечно, использовать `qsort` в Python не имеет особого смысла, потому что в Python есть собственный снециализированный алгоритм сортировки. Тем не менее передавать функции Python в виде обратных вызовов C — чрезвычайно полезный прием для интеграции многих сторонних библиотек.

Модуль `ctypes` очень популярен среди программистов на Python, потому что он входит в стандартную библиотеку. Его недостаток заключается в том, что приходится много работать с типами на низком уровне и писать шаблонный код для взаимодействия с загруженными библиотеками. Поэтому некоторые раз-

работчики предпочитают сторонний пакет CFFI, который упрощает вызовы внешних функций. Этот пакет рассматривается в следующем разделе.

CFFI

CFFI — интерфейс внешних функций для Python, который считается интересной альтернативой библиотеке `ctypes`. Он не входит в стандартную библиотеку, но доступен из PyPI в виде пакета `cffi`. Он отличается от `ctypes` тем, что ориентирован больше на повторное использование объявлений на обычном языке C, чем на то, чтобы предоставлять обширные Python API в одном модуле. Он устроен гораздо сложнее, а также дает возможность автоматически компилировать отдельные части уровня интеграции в расширения с помощью компилятора C. Это значит, что CFFI можно использовать как гибридное решение, которое заполняет нробел между расширениями на обычном C и `ctypes`.

Это очень большой нроект, так что его певозможно кратко описать в нескольких абзацах. С другой стороны, было бы ненравильно о нем не рассказывать. Мы уже рассмотрели пример с интеграцией функции `qsort()` из стандартной библиотеки с номощью `ctypes`. А значит, чтобы продемонстрировать основные различия между двумя решениями, лучше всего заново реализовать тот же нример средствами `cffi`.

Надеюсь, следующий блок кода скажет вам больше, чем несколько абзацев текста:

```
from random import shuffle

from cffi import FFI

ffi = FFI()

ffi.cdef("""
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
""")
C = ffi.dlopen(None)

@ffi.callback("int(void*, void*)")
def cffi_int_compare(a, b):
    # Сигнатура обратного вызова требует точного совпадения типов.
    # Здесь меньше "черной магии", чем в ctypes,
    # но вам приходится точнее выражать свои намерения
    # и использовать явное приведение типов
    int_a = ffi.cast('int*', a)[0]
```

```
    int_b = ffi.cast('int*', b)[0]
    print(" %s cmp %s" % (int_a, int_b))

    # согласно спецификации qsort, функция должна возвращать:
    # * отрицательное значение, если a < b
    # * ноль, если a == b
    # * положительное значение, если a > b
    return int_a - int_b

def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("перемешанный массив: ", numbers)

    c_array = ffi.new("int[]", numbers)

    C.qsort(
        # указатель на отсортированный массив
        c_array,
        # длина массива
        len(c_array),
        # размер одного элемента массива
        ffi.sizeof('int'),
        # обратный вызов (указатель на функцию сравнения C)
        cffi_int_compare,
    )
    print("отсортированный массив:   ", list(c_array))

if __name__ == "__main__":
    main()
```

Результат практически не отличается от того, который рассматривался ранее в примере с обратными вызовами C в `ctypes`. Обратите внимание, что использовать CFFI, чтобы интегрировать `qsort` в Python, в целом так же бессмысленно, как применять для этого `ctypes`. Однако приведенный пример демонстрирует основные различия между `ctypes` и `cffi` в том, что касается работы с типами данных и обратных вызовов.

Итоги

В этой главе рассматривалась одна из самых сложных тем этой книги. Мы обсудили, зачем может понадобиться интегрировать Python с другими языками и с помощью каких инструментов создаются расширения Python для этой интеграции. Сначала мы писали расширения на чистом C, которые опирались только на Python/C API, а затем заново реализовали их на Cython, чтобы показать, как легко это делается, если применять подходящий инструментарий.

Бывают ситуации, когда приходится выбирать сложный путь и не использовать ничего, кроме «голого» компилятора C и заголовков Python.h. Однако в общем случае рекомендуется применять такие инструменты, как Cython, потому что в результате код в вашей кодовой базе будет удобнее читать и сопровождать. Это также избавит вас от многих проблем из-за невнимательного подсчета ссылок и некорректного управления памятью.

Тема расширений завершилась разговором о `ctypes` и CFFI — альтернативных способах решения проблем с интеграцией общих библиотек. Поскольку эти инструменты не требуют написания специальных расширений для вызова функций из скомпилированных двоичных файлов, они годятся на роль основных инструментов интеграции динамических/общих библиотек с закрытым кодом, особенно если вам не нужно использовать свой собственный код на C.

В нескольких последних главах мы рассмотрели немало сложных тем: от продвинутых паттернов проектирования и конкурентного выполнения до событийного программирования и сопряжения Python с разными языками. А теперь перейдем к теме сопровождения приложений на Python: от тестирования и контроля качества до упаковки, мониторинга и оптимизации приложений любого размера.

Одна из самых важных проблем при сопровождении программного продукта — проверить, что код написан правильно. Поскольку со временем код неизбежно усложняется, без налаженного режима тестирования становится труднее убедиться в том, что он работает корректно. Также с увеличением объема становится невозможно эффективно тестировать код без той или иной разновидности автоматизации. В следующей главе вы узнаете, какие приемы и инструменты Python позволяют автоматизировать процессы тестирования и контроля качества.

10

Автоматизация тестирования и контроля качества

Программирование — сложное дело. Какой бы язык вы ни использовали, с какими бы фреймворками ни разрабатывали программы, каким бы элегантным ни был ваш стиль программирования, вам не удастся проверить правильность кода, просто читая его. Дело даже не в том, что в нетривиальных приложениях обычно много кода. Любое полноценное ПО, как правило, состоит из нескольких уровней и зависит от многих внешних или заменяемых компонентов: операционных систем, библиотек, баз данных, кэшей, веб-API или клиентов, которые взаимодействуют с вашим кодом (например, браузеров).

Сложность современных программных продуктов означает, что при проверке правильности часто нужно выходить за границы кода. Необходимо учитывать, в какой среде выполняется код, а также какие возможны варианты заменяемых компонентов и способы взаимодействия с кодом. Поэтому разработчики качественных продуктов часто применяют специальные приемы тестирования, которые позволяют быстро и надежно убедиться, что код удовлетворяет заданным критериям.

Другая проблема сложных программных продуктов — **сопровождаемость** (maintainability). Речь идет о том, насколько легко будет поддерживать продолжающуюся разработку продукта. В ходе этой разработки потребуется не только реализовывать новые возможности и улучшения, но также диагностировать и исправлять ошибки, которые неизбежно будут обнаруживаться. В программах с хорошей сопровождаемостью изменения можно внедрять с небольшими усилиями, а риск внесения новых дефектов минимален.

Как вы, вероятно, догадываетесь, сопровождаемость зависит от многих качеств программного продукта. Конечно, автоматизированное тестирование помогает снизить риски от внесения изменений; оно позволяет убедиться в том, что известные сценарии использования правильно отрабатываются существующим и будущим кодом. Но этого недостаточно, чтобы гарантировать, что будущие изменения окажется просто реализовать. Вот почему в современных методологиях тестирования также используются автоматизированные средства для вычисления метрик контроля качества и тестирования. Эти средства обеспечивают соблюдение стандартов стиля и других аспектов кода, а также выявляют потенциально ошибочные фрагменты кода и уязвимости безопасности.

Современная область тестирования необъятна: легко потеряться в изобилии методологий, инструментов, фреймворков, библиотек и утилит. Поэтому в этой главе приводится обзор самых популярных методов автоматизации тестирования и контроля качества, которые часто применяют профессиональные разработчики на Python. Вы получите общее представление о доступных возможностях и о том, как организовать процедуру тестирования. Здесь рассматриваются следующие темы:

- Принципы разработки через тестирование.
- Написание тестов с помощью `pytest`.
- Автоматизация контроля качества.
- Тестирование изменений.
- Полезные инструменты тестирования.

В этой главе будет использоваться много пакетов из PyPI, поэтому начнем с технических требований.

Технические требования

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `pytest`
- `redis`
- `coverage`
- `mypy`
- `mutmut`
- `faker`
- `freezegun`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2010.

Принципы разработки через тестирование

Тестирование — один из самых важных элементов процесса разработки ПО. Оно настолько важно, что даже существует методология разработки, которая называется **разработкой через тестирование**, или **TDD** (Test-Driven Development). Она предписывает составлять требования к продукту в виде тестов на самом первом шаге разработки кода.

Принцип прост: вы в первую очередь сосредоточиваетесь на тестах. С их помощью вы описываете поведение продукта, проверяете его правильность и ищете потенциальные ошибки. Только когда тесты будут готовы, можно приступать к фактической реализации, которая должна проходить тесты.

В простейшем варианте TDD — это итеративный процесс, который состоит из следующих шагов:

1. **Написание тестов**. Тесты должны отражать спецификацию некоторой функциональности или улучшений, которые еще не реализованы.
2. **Запуск тестов**. На этой стадии все новые тесты должны проваливаться, потому что функциональность или улучшение еще не реализованы.
3. **Написание минимально допустимой реализации**. Код должен быть предельно простым, но правильным. Ничего страшного, если он выглядит недостаточно элегантно или если у него проблемы с быстродействием. Главная цель на этом шаге — чтобы код удовлетворял всем тестам, которые написаны на шаге 1. Кроме того, проблемы проще диагностировать в простом коде, чем в оптимизированном для быстродействия.
4. **Запуск тестов**. На этой стадии все тесты должны проходить. Это относится как к новым, так и к ранее написанным тестам. Если какие-то тесты проваливаются, код надо переработать, пока он не будет удовлетворять всем требованиям.
5. **Доработка и улучшение**. Когда все тесты проходят, код последовательно перерабатывается, пока он не будет удовлетворять требуемым стандартам качества. На этом шаге применяется упрощение, рефакторинг, а иногда и очевидная оптимизация (если раньше задача решалась методом грубой силы). После каждого изменения все тесты нужно запускать заново, чтобы убедиться, что никакая функциональность не нарушена.

Этот простой процесс позволяет итеративно расширять приложение, не беспокоясь о том, что новое изменение нарушит ранее существовавшую и протестированную функциональность. Он также помогает избежать преждевременной оптимизации и выстроить процесс разработки как серию простых и доступных шагов.

TDD не принесет ожидаемых результатов без должной рабочей гигиены, поэтому важно соблюдать ряд базовых принципов:

- **Небольшие тестируемые модули**. В TDD часто упоминаются «юниты» (единицы, модули) кода и юнит-тесты (модульные тесты). Модуль — это простой автономный участок кода, который должен (по возможности) выполнять только одну операцию, а каждый модульный тест должен проверять одну функцию или метод с одним набором аргументов. Такой подход не только упрощает написание тестов, но и стимулирует применение хороших практик разработки и таких паттернов, как принцип единственной обязанности и инверсия управления (см. главу 5).
- **Небольшие и целенаправленные тесты**. Почти всегда лучше создать много маленьких и простых тестов, чем один длинный и сложный. Каждый тест должен проверять только один аспект или одно требование нужной функциональности. Узконаправленные тесты упрощают диагностику потенциальных проблем и сопровождение набора тестов. Небольшие тесты помогают точнее обнаруживать проблемы и просто лучше читаются.
- **Изолированные и независимые тесты**. Успех каждого отдельного теста не должен зависеть от конкретного порядка выполнения всех тестов в наборе. Если тест опирается на конкретное состояние среды выполнения, то он сам должен позаботиться, чтобы все предусловия соблюдались. Аналогичным образом все побочные эффекты теста должны быть устранены после того, как он выполнен. Методы, соответствующие этим стадиям подготовки и зачистки, часто называются `setup` и `teardown` соответственно.

На основе этих принципов можно писать тесты, которые будут понятными и простыми в сопровождении. И это важно, потому что написание тестов, как и всякая производственная деятельность, занимает время и увеличивает затраты на разработку. Тем не менее при правильном исполнении эти вложения окупаются очень быстро. Систематическая и автоматизированная процедура тестирования снижает количество дефектов, которые могут добраться до конечных пользователей. Также она служит основой для проверки известных программных ошибок.

Тщательно тестировать код и следить за соблюдением некоторых базовых принципов обычно помогают специализированные библиотеки или фрейм-

ворки. Программистам Python повезло, потому что в стандартной библиотеке Python есть два встроенных модуля, созданных именно для автоматизации тестирования:

- `doctest`: модуль для тестирования интерактивных примеров кода, содержащихся в doc-строках. Этот механизм удобен для того, чтобы объединять документацию с тестами. Теоретически `doctest` может проводить модульные тесты, но чаще с помощью этого модуля разработчики проверяют, что фрагменты кода, содержащиеся в doc-строках, отражают правильные примеры использования.

За дополнительной информацией о `doctest` обращайтесь к официальной документации по адресу https://docs.python.org/3/library/doctest.html.

- `unittest`: полнофункциональный фреймворк для тестирования, создателей которого вдохновлял JUnit (популярный фреймворк Java). Позволяет объединять тесты в тестовые сценарии и наборы тестов и предоставляет стандартные средства для управления примитивами подготовительной и завершающей фазы тестов. В `unittest` есть встроенное средство исполнения, которое может искать тестовые модули по всей кодовой базе и выполнять конкретные группы тестов.

За дополнительной информацией о `unittest` обращайтесь к официальной документации по адресу https://docs.python.org/3/library/unittest.html.

Эти два модуля вместе покрывают большинство потребностей в тестировании, которые возникают даже у самых требовательных разработчиков. К сожалению, модуль `doctest` рассчитан на предельно конкретный сценарий использования тестов (тестирование примеров кода), а `unittest` требует довольно большого объема шаблонного кода из-за того, что организация тестов ориентирована на классы. Средство исполнения также получилось менее гибким, чем могло бы быть. Вот почему многие профессиональные программисты предпочитают использовать сторонние фреймворки, доступные в PyPI.

Один из таких фреймворков — `pytest`. Вероятно, это один из лучших и наиболее зрелых фреймворков для тестирования Python. Он обеспечивает более удобные средства организации тестов в виде неструктурированных модулей с тестовыми функциями (вместо классов), но при этом он совместим с иерархией тестов на базе классов, которая принята в `unittest`. В `pytest` реализован более совершен-

ный исполнитель тестов, а также есть широкий выбор необязательных расширений.

Из-за перечисленных преимуществ `pytest` мы не будем подробно рассматривать `unittest` и `doctest`. Эти отличные и полезные модули, но `pytest` почти всегда оказывается эффективнее и практичнее. Поэтому мы рассмотрим примеры написания тестов с номощью этого фреймворка.

Написание тестов с помощью pytest

Пришло время применить теорию на практике. Преимущества TDD вам уже известны, так что мы попытаемся написать простую программу и протестировать ее. Мы обсудим, как устроен тиничный тест, а затем перейдем к стандартным приемам тестирования и инструментам, которые часто применяют профессиональные программисты на Python. И все это будет сделано с помощью фреймворка для тестирования `pytest`.

Для этого понадобится задача, которую мы будем решать. В конце концов, тестирование начинается на самом первом этапе жизненного цикла разработки — когда определяются требования к продукту. Во многих методологиях тестирования тесты представляют собой не что иное, как способ описания требований в исполняемой форме на уровне программного кода.

Трудно найти единственную реалистичную задачу, которая позволила бы продемонстрировать разные методологии тестирования и в то же время уложилась бы в формат книги. Поэтому мы рассмотрим несколько небольших задач, не связанных друг с другом. Также мы вернемся к некоторым примерам из предыдущих глав этой книги.

Конечно, с точки зрения TDD писать тесты для существующего кода — нетипичный подход, потому что в идеале нанисание тестов должно предшествовать реализации, а не наоборот. Тем не менее такая практика тоже встречается. Профессиональным нрограммистам нередко достается в наследство код, который был плохо нротестирован или вообще не тестировался. Если в такой ситуации вы хотите надежно нротестировать нрограммный продукт, то рано или поздно придется выполнить всю пропущенную работу. В нашем случае написание тестов для существующего кода — это также удобный повод поговорить о проблемах, связанных с написанием тестов после кода.

Первый пример будет довольно простым. Он номожет понять базовую структуру тестов, а также находить и выполнять тесты с помощью исполнителя `pytest`. Требуется создать функцию со следующими особенностями:

- она принимает итерируемый объект, содержащий элементы, и размер накета (batch);

- возвращает итерируемый объект подсписков, в котором каждый подсписок представляет собой группу (пакет) последовательных элементов из исходного списка. Порядок элементов должен оставаться неизменным;
- все пакеты имеют одинаковый размер;
- если в исходпом списке не хватает элемептов, чтобы заполнить последний пакет, этот пакет может быть короче других, но оп не может быть пустым.

Это будет отпосительно небольшая, но полезная функция. Например, с ее помощью можно обрабатывать большие потоки данных без необходимости полпостью загружать их в память процесса. Также ее можно использовать, чтобы распределять задачи по отдельным рабочим потокам или процессам (см. главу 6).

Сначала напишем заглушку нашей функции, чтобы понять, с чем мы имеем дело. Функция будет пазываться `batches()` и храниться в файле с именем batch.py. Ее сигпатура может выглядеть так:

```
from typing import Any, Iterable, List

def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    pass
```

Мы пока не написали пикакой реализации, потому что ею следует заниматься после того, как будут готовы все тесты. Обратите внимание на аппотации типов, которые являются частью контракта между фупкцией и вызывающей стороной.

После этого можно импортировать функцию в тестовый модуль. По стандартным соглашениям тестовым модулям присваиваются имена вида `test_<имя_модуля>.py`, где `<имя_модуля>` — имя модуля, содержимое которого вы собираетесь тестировать. Создадим файл с именем test_batch.py.

Первый тест будет выполнять довольпо стандартную процедуру: передавать в фупкцию входные данпые и сравнивать результаты с ожидаемыми. Будем использовать в качестве входных данных обычный литерал списка. Вот пример тестового кода:

```
from batch import batches

def test_batch_on_lists():
    assert list(batches([1, 2, 3, 4, 5, 6], 1)) == [
        [1], [2], [3], [4], [5], [6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 2)) == [
        [1, 2], [3, 4], [5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 3)) == [
```

```
        [1, 2, 3], [4, 5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 4)) == [
        [1, 2, 3, 4], [5, 6],
    ]
```

В pytest инструкция assert считается предпочтительным способом проверять пред- и постусловия тестируемых единиц кода. pytest может проверять такие утверждения, распознавать их исключения и благодаря этому выдавать подробные отчеты о непрошедших тестах в удобочитаемой форме.

Такая структура тестов популярна для небольших программ. Часто этого достаточно, чтобы убедиться в том, что они работают как предполагалось. Тем не менее имеющиеся тесты недостаточно четко отражают наши требования, так что их структуру стоит немного изменить.

Два следующих дополнительных теста более явно соответствуют исходным требованиям:

```
from itertools import chain

def test_batch_order():
    iterable = range(100)
    batch_size = 2

    output = batches(iterable, batch_size)

    assert list(chain.from_iterable(output)) == list(iterable)

def test_batch_sizes():
    iterable = range(100)
    batch_size = 2

    output = list(batches(iterable, batch_size))

    for batch in output[:-1]:
        assert len(batch) == batch_size
    assert len(output[-1]) <= batch_size
```

Тест test_batch_order() проверяет, что порядок элементов в группах совпадает с их порядком в исходном итерируемом объекте. Тест test_batch_sizes() проверяет, что все пакеты имеют одинаковый размер (кроме последнего пакета, который может быть короче).

В обоих тестах можно заметить определенную закономерность. Собственно, многие тесты имеют очень нохожую структуру:

1. **Подготовка**. Подготавливаются тестовые данные и остальные предусловия. В нашем случае нодготовка заключается в инициализации аргументов iterable и batch_size.

2. **Выполнение**. Тестируемая единица кода используется на практике, а результаты сохраняются для последующего анализа. В нашем случае это вызов функции `batches()`.
3. **Проверка**. Анализируя результаты выполнения, мы проверяем, удовлетворено ли то или иное требование. В данном случае мы проверяем сохраненный вывод с помощью ипструкций `assert`.
4. **Завершение**. Все ресурсы, которые могут повлиять на другие тесты, освобождаются или возвращаются в состояние, в котором опи находились до этапа нодготовки. В нашем примере никакие ресурсы не захватываются, так что этот шаг можно пропустить.

В соответствии с процессом тестирования, описанным в разделе «Принципы разработки через тестирование», прямо сейчас тест не должен проходить, потому что у нас еще нет никакой реализации функции. Запустим исполнитель `pytest` и посмотрим, что из этого выйдет:

```
$ pytest -v
```

Полученный результат выглядит примерно так:

```
======================== test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/01 -
Writing tests with pytest
collected 3 items

test_batch.py::test_batch_on_lists FAILED                      [ 33%]
test_batch.py::test_batch_order FAILED                         [ 66%]
test_batch.py::test_batch_sizes FAILED                         [100%]

============================ FAILURES ============================
_____________________ test_batch_on_lists _____________________

    def test_batch_on_lists():
>       assert list(batches([1, 2, 3, 4, 5, 6], 1)) == [
            [1], [2], [3], [4], [5], [6]
        ]
E       TypeError: 'NoneType' object is not iterable

test_batch.py:7: TypeError
_____________________ test_batch_order _____________________

    def test_batch_order():
        iterable = range(100)
        batch_size = 2
```

```
        output = batches(iterable, batch_size)

>       assert list(chain.from_iterable(output)) == list(iterable)
E       TypeError: 'NoneType' object is not iterable

test_batch.py:27: TypeError
_____________________ test_batch_sizes _____________________

    def test_batch_sizes():
        iterable = range(100)
        batch_size = 2

>       output = list(batches(iterable, batch_size))
E       TypeError: 'NoneType' object is not iterable

test_batch.py:34: TypeError
====================== short test summary info ======================
FAILED test_batch.py::test_batch_on_lists - TypeError: 'NoneType' ...
FAILED test_batch.py::test_batch_order - TypeError: 'NoneType' obj...
FAILED test_batch.py::test_batch_sizes - TypeError: 'NoneType' obj...
```

Как видите, при запуске три теста не прошли, и мы получили подробный отчет о том, что же пошло не так. Во всех тестах обнаруживается одна и та же ошибка: в описании `TypeError` сказано, что объект `NoneType` не является итерируемым, поэтому его нельзя преобразовать в список. Это означает, что ни одно из трех требований еще не выполняется, — вполне логично, потому что функция `batches()` пока не делает ничего осмысленного.

Теперь нужно сделать так, чтобы тесты проходили. Наша цель — предоставить минимальную рабочую реализацию. Поэтому мы не будем делать ничего хитроумного, а напишем простую наивную реализацию на базе списков. Взгляните на первую итерацию:

```
from typing import Any, Iterable, List

def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    results = []
    batch = []

    for item in iterable:
        batch.append(item)
        if len(batch) == batch_size:
            results.append(batch)
            batch = []

    if batch:
        results.append(batch)

    return results
```

Идея проста: мы создаем список результатов и перебираем входной объект `iterable`, активно создавая новые группы по мере перебора. Когда группа заполняется, мы добавляем ее в список результатов и начинаем новую группу. Закончив перебор, мы проверяем, осталась ли неполная группа, и если да, то добавляем ее в тот же список, после чего возвращаем результат.

Это очень наивная реализация, которая может плохо работать с результатами произвольного размера, но ее должно быть достаточно для прохождения тестов. Запустим команду `pytest`, чтобы проверить, все ли в порядке:

```
$ pytest -v
```

Теперь результат выполнения тестов должен выглядеть так:

```
======================== test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/01 -
Writing tests with pytest
collected 3 items

test_batch.py::test_batch_on_lists PASSED                    [ 33%]
test_batch.py::test_batch_order PASSED                       [ 66%]
test_batch.py::test_batch_sizes PASSED                       [100%]
======================== 3 passed in 0.01s ========================
```

Как видите, все тесты прошли успешно, а следовательно, функция `batches()` удовлетворяет требованиям, которые описаны в наших тестах. Это не означает, что в коде нет никаких ошибок, однако теперь мы больше уверены в том, что он правильно работает в области условий, которые охвачены тестами. Чем больше тестов и чем они точнее, тем меньше остается сомнений в правильности кода.

Работа еще не закончена. Мы создали простую реализацию и убедились, что она функционирует. Теперь можно заняться улучшением кода. Одна из причин такой схемы работы заключается в том, что при самой простой реализации кода легче обнаруживать ошибки в тестах. Не забывайте, что тесты тоже являются кодом, поэтому в них тоже бывают ошибки.

Если реализация тестируемой единицы кода проста и понятна, вам будет проще разобраться, в чем проблема — в тестируемом коде или в самом тесте.

Очевидный недостаток первой итерации функции `batches()` заключается в том, что все промежуточные результаты должны храниться в списковой переменной `results`. Если аргумент `iterable` достаточно велик (или даже бесконечен), это

создаст большую нагрузку на приложение, потому что все данные придется держать в памяти. Правильнее было бы преобразовать эту функцию в генератор, который порождает последовательные результаты. Для этого достаточно совсем незначительных изменений:

```
def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    batch = []
    for item in iterable:
        batch.append(item)

        if len(batch) == batch_size:
            yield batch
            batch = []

    if batch:
        yield batch
```

Можно предложить и другое решение, которое использует итераторы и модуль `itertools`:

```
from itertools import islice

def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    iterator = iter(iterable)

    while True:
        batch = list(islice(iterator, batch_size))

        if not batch:
            return

        yield batch
```

Это одна из самых замечательных особенностей методологии TDD: можно легко экспериментировать с кодом и модифицировать существующую реализацию с минимальным риском что-то сломать. Чтобы убедиться в этом, замените функцию `batches()` одной из реализаций, приведенных выше, запустите тесты и посмотрите, удовлетворяет ли обновленная функция исходным требованиям.

Задача из нашего примера невелика и легко укладывается в голове, так что написать тесты было совсем несложно. Тем не менее не весь наш код будет таким. При тестировании более крупных и сложных участков кода часто требуются дополнительные инструменты и приемы, которые позволяют писать чистые

и удобочитаемые тесты. В нескольких ближайших разделах мы рассмотрим популярные приемы тестирования, которые часто применяют программисты на Python, и вы узнаете, как реализовать эти приемы с номощью `pytest`. Первый из них — параметризация тестов.

Параметризация тестов

Прямое сравнение полученпого и ожидаемого вывода фупкции — стандартный прием написания коротких модульных тестов. Он позволяет создавать понятные и лаконичпые тесты. Именно поэтому мы использовали этот прием в первом тесте `test_batch_on_lists()` из предыдущего раздела.

Один из недостатков этого приема заключается в том, что он нарушает классический паттерн «нодготовка — вынолнение — проверка — завершение». Нет четкого нонимания, какие инструкции подготавливают коптекст теста, при каком вызове функции выполпяется единица кода и какие инструкции проверяют результат.

Другая проблема связана с тем, что с ростом объема сравниваемых данных ввода и вывода тесты становятся слишком большими. Их трудно читать, а потенциальные независимые ошибки пе будут нормально изолированы. Чтобы лучше понять суть проблемы, вспомните код теста `test_batch_on_lists()`:

```
def test_batch_on_lists():
    assert list(batches([1, 2, 3, 4, 5, 6], 1)) == [
        [1], [2], [3], [4], [5], [6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 2)) == [
        [1, 2], [3, 4], [5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 3)) == [
        [1, 2, 3], [4, 5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 4)) == [
        [1, 2, 3, 4], [5, 6],
    ]
```

Каждая инструкция `assert` отвечает за проверку одной пары данных ввода и вывода. Но в зависимости от того, как сконструировапы пары, они могут проверять разные условия исходпых требований. В пашем случае первые три инструкции проверяют, что размеры всех выходпых групп совпадают. Однако последняя инструкция проверяет, что неполная группа тоже будет возвращена, если длина аргумента `iterable` не кратпа `batch_size`. Цель теста оказывается несколько расплывчатой, потому что он нарушает тот принцип, что тесты должны быть небольшими и целенаправленными.

Структуру теста можно немного улучшить, если выделить подготовку всех наборов данных в отдельную фазу теста, а затем перебрать эти наборы на фазе выполнения. В нашем случае это можно сделать с помощью простого литерала словаря:

```
def test_batch_with_loop():
    iterable = [1, 2, 3, 4, 5, 6]
    samples = {
        # одинаковые группы
        1: [[1], [2], [3], [4], [5], [6]],
        2: [[1, 2], [3, 4], [5, 6]],
        3: [[1, 2, 3], [4, 5, 6]],
        # группы с остатком
        4: [[1, 2, 3, 4], [5, 6]],
    }

    for batch_size, expected in samples.items():
        assert list(batches(iterable, batch_size)) == expected
```

Посмотрите, как небольшое изменение в структуре теста позволило обозначить, какие наборы данных должны проверять то или иное требование к функции. Не для каждого требования нужно писать отдельный тест. Помните: практичность важнее безупречности.

Благодаря этому изменению этапы подготовки и выполнения теста гораздо четче отделяются друг от друга. Можно сказать, что выполнение функции `batch()` параметризуется содержимым словаря `samples`. По сути, в одном запуске теста выполняются сразу несколько меньших тестов.

Другая проблема тестирования нескольких наборов данных в одной тестовой функции состоит в том, что тест может завершиться преждевременно. Если первая инструкция `assert` не проходит, тест немедленно остановится. Мы не знаем, завершатся ли последующие инструкции `assert` успехом или неудачей, пока не исправим первую ошибку, чтобы продолжить выполнение теста. Между тем, когда все ошибки видны одновременно, это часто помогает лучше понять, что же пошло не так в тестируемом коде.

Эту проблему не удастся легко решить в тесте с циклом. К счастью, в `pytest` встроена поддержка параметризации тестов в форме декоратора `@pytest.mark.parametrize`. Это позволяет вынести параметризацию фазы выполнения за пределы тела теста. Модуль `pytest` достаточно интеллектуален, чтобы интерпретировать каждый набор входных параметров как отдельный «виртуальный» тест, который будет запускаться независимо от других тестов.

`@pytest.mark.parametrize` должен принимать как минимум два позиционных аргумента:

- `argnames`: перечень имен аргументов, на основании которого `pytest` будет передавать тестируемой функции тестовые параметры в качестве аргументов. Этот перечень может быть строкой значений, разделенных запятыми, или списком/кортежем строк.
- `argvalues`: итерируемый объект с наборами параметров для каждого отдельного запуска теста. Обычно это список списков или кортеж кортежей.

Последний пример можно переписать с использованием декоратора `@pytest.mark.parametrize`:

```
import pytest

@pytest.mark.parametrize(
    "batch_size, expected", [
        # одинаковые группы
        [1, [[1], [2], [3], [4], [5], [6]]],
        [2, [[1, 2], [3, 4], [5, 6]]],
        [3, [[1, 2, 3], [4, 5, 6]]],
        # группы с остатком
        [4, [[1, 2, 3, 4], [5, 6]]]
    ]
)
def test_batch_parameterized(batch_size, expected):
    iterable = [1, 2, 3, 4, 5, 6]
    assert list(batches(iterable, batch_size)) == expected
```

Если теперь запустить все ранее написанные тесты командой `pytest -v`, мы получим такой результат:

```
======================== test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/01 -
Writing tests with pytest
collected 8 items

test_batch.py::test_batch_on_lists PASSED                    [ 12%]
test_batch.py::test_batch_with_loop PASSED                   [ 25%]
test_batch.py::test_batch_parameterized[1-expected0] PASSED  [ 37%]
test_batch.py::test_batch_parameterized[2-expected1] PASSED  [ 50%]
test_batch.py::test_batch_parameterized[3-expected2] PASSED  [ 62%]
test_batch.py::test_batch_parameterized[4-expected3] PASSED  [ 75%]
test_batch.py::test_batch_order PASSED                       [ 87%]
test_batch.py::test_batch_sizes PASSED                       [100%]

========================= 8 passed in 0.01s =========================
```

Как видите, в отчете упоминаются четыре разных экземпляра `test_batch_parameterized()`. Если в одном из них произойдет ошибка, это никак не повлияет на другие.

Параметризация тестов фактически выводит часть классических обязанностей тестов — подготовку контекста тестирования — за пределы функции тестирования. Это улучшает перспективы повторного использования тестового кода, а на первый план выходит то, что действительно важно: выполнение тестов и проверка результата.

Другой способ вывести функциональность подготовки из тела теста опирается на повторно используемые фикстуры. В `pytest` уже есть отличная встроенная поддержка фикстур с фантастическими возможностями.

Фикстуры pytest

Фикстура (fixture) моделирует фиксированную конфигурацию среды, которая симулирует реальное использование тестируемого программного компонента. Фикстурой может быть все что угодно: от конкретных объектов, которые используются в качестве входных аргументов, и конфигураций переменных окружения до наборов данных, которые хранятся в удаленной базе данных и будут использоваться во время тестирования.

В `pytest` фикстура — это повторно используемый фрагмент кода подготовки и/или завершения, который может предоставляться тестовым функциям в качестве зависимости. В `pytest` есть встроенный механизм внедрения зависимостей, который позволяет писать модульные и масштабируемые наборы тестов.

Тема внедрения зависимостей в Python рассматривалась в главе 5 «Интерфейсы, паттерны и модульность».

Чтобы создать фикстуру `pytest`, нужно определить именованную функцию и снабдить ее декоратором `@pytest.fixture`:

```
import pytest

@pytest.fixture
def dependency():
    return "значение фикстуры"
```

`pytest` запускает функции фикстуры перед выполнением тестов. Возвращаемое значение функции фикстуры (в данном случае строка `"значение фикстуры"`) будет передаваться функции тестирования как входной аргумент. Можно также

включить в одну функцию фикстуры как код подготовки, так и код завершения. Для этого служит следующий синтаксис генератора:

```
@pytest.fixture
def dependency_as_generator():
    # Код подготовки
    yield "значение фикстуры"
    # Код завершения
```

Когда используется синтаксис генератора, `pytest` принимает значение, которое предоставила функция фикстуры, и приостанавливает ее выполнение до завершения теста. Когда тест завершится, `pytest` возобновляет выполнение всех используемых функций фикстур сразу же после инструкции `yield` независимо от результата теста (успех или неудача). Это позволяет удобно и надежно выполнить завершающий сброс тестовой среды.

Чтобы использовать фикстуру в тесте, нужно указать ее имя во входном аргументе тестовой функции:

```
def test_fixture(dependency):
    pass
```

При запуске исполнителя `pytest` собирает все используемые фикстуры, проверяя сигнатуры тестовых функций и сопоставляя имена с доступными функциями фикстур. По умолчанию `pytest` обнаруживает фикстуры и разрешает их имена несколькими способами:

- **Локальные фнкстуры.** Тесты могут использовать все фикстуры, доступные из того же модуля, в котором определены сами тесты, в том числе фикстуры, импортированные в тот же модуль. Локальные фикстуры всегда имеют более высокий приоритет, чем общие.
- **Общие фикстуры.** Тесты могут использовать фикстуры, доступные в модуле `conftest`, который хранится в одном каталоге с тестовым модулем или в любом из его каталогов-нредков. В тестовом наборе может быть несколько модулей `conftest`. Чем ближе фикстура из `conftest` находится в иерархии каталогов, тем выше ее приоритет, а общие фикстуры всегда имеют более высокий приоритет, чем фикстуры плагипов.
- **Фикстуры плагииов.** Плагины `pytest` могут предоставлять собственные фикстуры, имена которых будут соноставляться в последнюю очередь.

Наконец, фикстуры могут быть связаны с конкретными областями видимости, которые определяют срок существования значений фикстур. Эти области видимости чрезвычайно важны для фикстур, которые реализованы в виде генераторов, нотому что они определяют, когда будет выполняться код завершения. Доступны пять областей видимости:

- Область видимости `"function"`: используется по умолчанию. Функция фикстуры с областью видимости `"function"` выполняется один раз для каждого отдельного запуска теста, после чего уничтожается.
- Область видимости `"class"`: может использоваться для тестовых методов, написанных в стиле xUnit (на основе модуля `unittest`). Фикстуры с этой областью видимости уничтожаются после последнего теста в тестовом классе.
- Область видимости `"module"`: фикстуры с этой областью видимости уничтожаются после последнего теста в тестовом модуле.
- Область видимости `"package"`: фикстуры с этой областью видимости уничтожаются после последнего теста в заданном тестовом пакете (наборе тестовых модулей).
- Область видимости `"session"`: нечто вроде глобальной области видимости. Фикстуры с этой областью видимости существуют на протяжении всей работы исполнителя и уничтожаются после последнего теста.

Разные области видимости фикстур можно использовать, чтобы оптимизировать выполнение тестов, потому что фаза подготовки конкретной среды иногда может занимать довольно значительное время. Если разные тесты могут безопасно использовать одну и ту же фазу подготовки, может иметь смысл расширить используемую по умолчанию область видимости `"function"` до `"module"`, `"package"` или даже `"session"`.

Более того, фикстуры `"session"` можно использовать как для глобальной подготовки целых групп тестов, так и для глобального завершения. Поэтому они часто используются с флагом `autouse=True`, который помечает фикстуру как автоматическую зависимость для заданной группы тестов. Область видимости для таких фикстур определяется следующим образом:

- **Уровень модуля для фикстуры тестового модуля.** Если фикстура с флагом `autouse` включена в тестовый модуль (модуль с префиксом `test`), она будет автоматически помечена как зависимость для каждого теста в этом модуле.
- **Уровень пакета для фикстуры модуля conftest.** Если фикстура с флагом `autouse` включена в модуль `conftest` в заданном тестовом каталоге, она будет автоматически помечена как зависимость для каждого теста в каждом тестовом модуле в этом же каталоге (включая его подкаталоги).

Чтобы научиться пользоваться фикстурами в разных формах, лучше всего рассмотреть конкретный пример. Тесты для функции `batch()` из предыдущего раздела были относительно простыми и поэтому не подразумевали широкого применения фикстур. Фикстуры особенно полезны, если вам нужно обеспечить сложную инициализацию объектов или подготовку состояния внешних про-

граммных компонентов, таких как удаленные службы или базы данных. В главе 5 «Интерфейсы, паттерны и модульность» мы рассматривали примеры кода, который отслеживал количество просмотров страниц с подключаемыми подсистемами хранения данных, и в одном из этих примеров использовалось хранилище Redis. Тестирование этих подсистем может послужить отличным примером для фикстур `pytest`, поэтому вспомним общий интерфейс абстрактного базового класса `ViewsStorageBackend`:

```
from abc import ABC, abstractmethod
from typing import Dict

class ViewsStorageBackend(ABC):
    @abstractmethod
    def increment(self, key: str): ...

    @abstractmethod
    def most_common(self, n: int) -> Dict[str, int]: ...
```

Абстрактные базовые классы или другие разновидности реализаций интерфейсов (такие, как подклассы `Protocol`) отлично подходят для тестирования. Они позволяют сосредоточиться на поведении класса вместо реализации.

Чтобы протестировать поведение любой реализации `ViewsStorageBackend`, можно проверить такие утверждения:

- Если подсистема хранения пуста, метод `most_common()` возвращает пустой словарь.
- Если увеличить счетчики страниц для разных ключей и запросить количество самых частых ключей, большее или равное количеству увеличенных, мы получим все отслеживаемые счетчики.
- Если увеличить счетчики страниц для разных ключей и запросить количество самых частых ключей, меньшее или равное количеству увеличенных, мы получим сокращенный набор самых частых элементов.

Начнем с тестов, а затем перейдем к фактической реализации фикстуры. Первая тестовая функция для пустого хранилища получается очень простой:

```
import pytest
import random
from interfaces import ViewsStorageBackend

@pytest.mark.parametrize(
    "n", [0] + random.sample(range(1, 101), 5)
)
def test_empty_backend(backend: ViewsStorageBackend, n: int):
    assert backend.most_common(n) == {}
```

Этот тест не требует основательной подготовки. Можно было использовать статический набор из n аргументов, однако дополнительная параметризация случайными значениями служит приятным дополнением. Аргумент `backend` — объявление об использовании фикстуры, которое `pytest` будет разрешать при запуске теста.

Второй тест для получения полного набора счетчиков требует более объемной настройки и выполнения:

```
def test_increments_all(backend: ViewsStorageBackend):
    increments = {
        "key_a": random.randint(1, 10),
        "key_b": random.randint(1, 10),
        "key_c": random.randint(1, 10),
    }

    for key, count in increments.items():
        for _ in range(count):
            backend.increment(key)

    assert backend.most_common(len(increments)) == increments
    assert backend.most_common(len(increments) + 1) == increments
```

Тест начинается с объявления литерала словаря с нужными приращениями. Эта простая подготовка служит двум целям: переменная `increments` регулирует последующую фазу выполнения, а также предоставляет проверочные данные для двух проверяемых утверждений. Как и в предыдущем тесте, предполагается, что аргумент `backend` будет предоставляться фикстурой `pytest`.

Последний тест весьма похож на предыдущий:

```
def test_increments_top(backend: ViewsStorageBackend):
    increments = {
        "key_a": random.randint(1, 10),
        "key_b": random.randint(1, 10),
        "key_c": random.randint(1, 10),
        "key_d": random.randint(1, 10),
    }

    for key, count in increments.items():
        for _ in range(count):
            backend.increment(key)

    assert len(backend.most_common(1)) == 1
    assert len(backend.most_common(2)) == 2
    assert len(backend.most_common(3)) == 3

    top2_values = backend.most_common(2).values()
    assert list(top2_values) == (
        sorted(increments.values(), reverse=True)[:2]
    )
```

Этапы подготовки и выполнения подобны тем, что использовались в тестовой функции `test_increments_all()`. Если бы мы писали не тесты, то, вероятно, стоило бы выделить эти шаги в отдельные функции, пригодные для повторного использования. Но в данном случае это, пожалуй, повредило бы удобочитаемости кода. Тесты должны быть независимыми, так что небольшая избыточность не помешает, если благодаря ей тесты получаются ясными и четко выраженными. Тем не менее это правило — не догма, и его стоит применять с умом.

Все тесты написаны, теперь нужно обеспечить фикстуру. В главе 5 «Интерфейсы, паттерны и модульность» рассматривались две реализации подсистем хранения данных: `CounterBackend` и `RedisBackend`. В конечном итоге нам хотелось бы использовать один набор тестов для обеих подсистем. Рано или поздно мы к этому придем, но пока будем считать, будто подсистема хранения только одна. Это немного упростит ситуацию.

Предположим, что мы тестируем только `RedisBackend`. Эта подсистема определенно сложнее, чем `CounterBackend`, так что писать тесты будет интереснее. Можно было бы написать всего одну фикстуру, но `pytest` позволяет создавать модульные фикстуры, поэтому посмотрим, как работает эта возможность. Начнем со следующего кода:

```
from redis import Redis
from backends import RedisBackend

@pytest.fixture
def backend(redis_client: Redis):
    set_name = "test-page-counts"
    redis_client.delete(set_name)

    return RedisBackend(
        redis_client=redis_client,
        set_name=set_name
    )
```

Redis не входит в дистрибутивы большинства систем, так что вам, вероятно, придется устанавливать его самостоятельно. В дистрибутивах Linux соответствующий пакет доступен под именем `redis-server` в системном репозитории пакетов. Также можно воспользоваться Docker или Docker Compose. Ниже приведен короткий файл docker-compose.yml, который позволит быстро запустить Redis локально:

```
version: "3.7"
services:
  redis:
    image: redis
    ports:
      - 6379:6379
```

`redis_client.delete(set_name)` удаляет ключ из хранилища данных Redis, если он существует. Мы будем использовать тот же ключ при инициализации `RedisBackend`. Ключ Redis, в котором хранятся все приращения, будет создан при первой модификации хранилища, так что нам не нужно беспокоиться о несуществующих ключах. Таким образом гарантируется, что при каждой ипициализации фикстуры хранилище данных полностью пусто. По умолчанию для фикстуры используется область видимости уровня `"function"`, а значит, каждый тест, который использует эту фикстуру, будет получать пустое хранилище.

За дополнительной информацией об использовании Docker и Docker Compose обращайтесь к главе 2 «Современные среды разработки для Python».

Возможно, вы заметили, что экземпляр клиента Redis не создается в фикстуре `backend()`, а передается как входной аргумент функциям фикстур.

Механизм впедрения зависимостей в `pytest` также распространяется на функции фикстур. Это означает, что внутри фикстуры можно запрашивать другие фикстуры.

Вот пример фикстуры `redis_client()`:

```
from redis import Redis

@pytest.fixture(scope="session")
def redis_client():
    return Redis(host='localhost', port=6379)
```

Чтобы не усложнять, мы использовали фиксированные значения аргументов Redis `host` и `port`. Благодаря модульности будет проще заменить эти значения глобально, если вы когда-нибудь решите использовать удаленный адрес вместо локального.

Сохраните все тесты в модуле test_backends.py, занустите сервер Redis локально и вызовите исполнителя `pytest` командой `pytest -v`. Результат должен выглядеть примерно так:

```
======================== test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/03 -
Pytest's fixtures
collected 8 items

test_backends.py::test_empty_backend[0] PASSED                  [ 12%]
test_backends.py::test_empty_backend[610] PASSED                [ 25%]
test_backends.py::test_empty_backend[611] PASSED                [ 37%]
```

```
test_backends.py::test_empty_backend[7] PASSED                    [ 50%]
test_backends.py::test_empty_backend[13] PASSED                   [ 62%]
test_backends.py::test_empty_backend[60] PASSED                   [ 75%]
test_backends.py::test_increments_all PASSED                      [ 87%]
test_backends.py::test_increments_top PASSED                      [100%]
======================== 8 passed in 0.08s =========================
```

Все тесты проходят успешно, а это значит, что реализация `RedisBackend` прошла проверку. Было бы замечательно, если бы то же самое получилось с `CounterBackend`. Самое наивное, что можно сделать, — скопировать тесты и переписать тестовые фикстуры, чтобы предоставить новую реализацию подсистемы хранения. Однако хотелось бы избежать таких повторений.

Мы знаем, что тесты должны быть независимыми. Тем не менее в трех наших тестах упоминается только абстрактный базовый класс `ViewsStorageBackend`. Значит, они всегда будут одинаковыми независимо от фактической реализации тестируемых подсистем хранения. А следовательно, нужно каким-то образом определить параметризированную фикстуру, которая позволит повторить один тест для разных реализаций подсистемы хранения.

Параметризация функций фикстур несколько отличается от параметризации тестовых функций. Декоратор `@pytest.fixture` принимает необязательный именованный параметр `params` с итерируемым объектом, который содержит параметры фикстуры. Фикстура с параметром `params` также должна объявить об использовании специальной встроенной фикстуры `request`, которая среди прочего предоставляет доступ к текущему параметру фикстуры:

```
import pytest

@pytest.fixture(params=[param1, param2, ...])
def parmetrized_fixture(request: pytest.FixtureRequest):
    return request.param
```

Можно воспользоваться параметризированной фикстурой и методом `request.getfixturevalue()`, чтобы динамически загружать фикстуру в зависимости от ее параметра. Обновленный полный набор фикстур для наших тестовых функций теперь выглядит так:

```
import pytest
from redis import Redis
from backends import RedisBackend, CounterBackend

@pytest.fixture
def counter_backend():
    return CounterBackend()

@pytest.fixture(scope="session")
def redis_client():
```

```
    return Redis(host='localhost', port=6379)

@pytest.fixture
def redis_backend(redis_client: Redis):
    set_name = "test-page-counts"
    redis_client.delete(set_name)

    return RedisBackend(
        redis_client=redis_client,
        set_name=set_name
    )

@pytest.fixture(params=["redis_backend", "counter_backend"])
def backend(request):
    return request.getfixturevalue(request.param)
```

Если теперь выполнить тот же набор тестов с новым набором фикстур, вы увидите, что количество выполненных тестов удвоилось. Вот пример вывода команды `pytest -v`:

```
======================== test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/03 -
Pytest's fixtures
collected 16 items

test_backends.py::test_empty_backend[redis_backend-0] PASSED     [  6%]
test_backends.py::test_empty_backend[redis_backend-72] PASSED    [ 12%]
test_backends.py::test_empty_backend[redis_backend-23] PASSED    [ 18%]
test_backends.py::test_empty_backend[redis_backend-48] PASSED    [ 25%]
test_backends.py::test_empty_backend[redis_backend-780] PASSED   [ 31%]
test_backends.py::test_empty_backend[redis_backend-781] PASSED   [ 37%]
test_backends.py::test_empty_backend[counter_backend-0] PASSED   [ 43%]
test_backends.py::test_empty_backend[counter_backend-72] PASSED  [ 50%]
test_backends.py::test_empty_backend[counter_backend-23] PASSED  [ 56%]
test_backends.py::test_empty_backend[counter_backend-48] PASSED  [ 62%]
test_backends.py::test_empty_backend[counter_backend-780] PASSED [ 68%]
test_backends.py::test_empty_backend[counter_backend-781] PASSED [ 75%]
test_backends.py::test_increments_all[redis_backend] PASSED      [ 81%]
test_backends.py::test_increments_all[counter_backend] PASSED    [ 87%]
test_backends.py::test_increments_top[redis_backend] PASSED      [ 93%]
test_backends.py::test_increments_top[counter_backend] PASSED    [100%]
======================== 16 passed in 0.08s =========================
```

Благодаря разумному использованию фикстур удалось сократить объем тестового кода без ущерба для удобочитаемости теста. Также можно было повторно использовать те же тестовые функции для проверки классов, которые должны обладать тем же поведением, но имеют другие реализации.

Проектировать фикстуры следует осторожно, потому что если злоупотреблять внедрением зависимостей, то весь набор тестов будет труднее воспринимать. Функции фикстур должны быть простыми и хорошо документированными.

Фикстуры удобно использовать, чтобы обеспечивать связь с такими внешними службами, как Redis, потому что Redis легко устанавливается, а применять его для целей тестирования можно без какой-либо специальной конфигурации. Но иногда в вашем коде будет использоваться удаленная служба или ресурс, который невозможно легко предоставить в тестовой среде или протестировать без деструктивных изменений. Например, такие ситуации часто встречаются при работе со сторонними веб-API, оборудованием или закрытыми библиотеками/двоичными файлами. В подобных случаях часто помогают **суррогатные** (fake) объекты или **макеты** (mocks), которые служат заменой для реальных объектов. Они подробнее рассматриваются в следующем разделе.

Суррогаты

Написание модульных тестов предполагает, что тестируемую единицу можно изолировать от остального кода. Тесты обычно передают функции или методу некоторые данные и проверяют возвращаемое значение и/или побочные эффекты их выполнения. В основном это помогает убедиться, что:

- тесты охватывают атомарную часть приложения, например функцию, метод, класс или интерфейс;
- тесты производят детерминированные, воспроизводимые результаты.

Иногда неочевидно, как правильно изолировать программный компонент, или такую изоляцию трудно обеспечить. В предыдущем разделе мы рассмотрели пример тестового набора, который среди прочего проверял фрагмент кода, взаимодействующий с хранилищем данных Redis. Подключение к Redis обеспечивалось фикстурой `pytest`, и вы убедились, что это было не так уж сложно. Но что именно при этом тестировалось — только наш код или также поведение Redis?

В этом конкретном случае такое подключение к Redis было прагматичным решением. Наш код выполнял мало работы, а бóльшая часть нагрузки пришлась на внешнее хранилище данных. Оно не будет работать правильно, если Redis работает неправильно. Чтобы протестировать все решение, нам пришлось бы протестировать интеграцию кода с хранилищем данных Redis. Такие тесты, обычно называемые **интеграционными**, часто используются в тестовом ПО, которое сильно зависит от внешних компонентов.

Но безопасные интеграционные тесты не всегда возможны. Не каждая служба, которую вы используете, запускается локально так же легко, как Redis. Иногда приходится иметь дело с особыми компонентами, которые нельзя воспроизвести в отрыве от повседневного реального использования. В таких случаях приходится заменять зависимость **суррогатным объектом** (fake-объектом), который симулирует реальный компонент.

Чтобы лучше понять типичные сценарии использования суррогатов в тестах, рассмотрим такую воображаемую историю: вы строите масштабируемое приложение, которое дает клиентам возможность отслеживать счетчики обращений к их сайтам в реальном времени. В отличие от конкурентов вы предлагаете решение с высокой доступностью и масштабируемостью при очень низкой задержке, причем ваша система может работать во многих вычислительных центрах по всему миру, не искажая результаты. В основу продукта ляжет небольшой класс счетчика из модуля `backends.py`.

Реализовать распределенную хеш-таблицу (тип данных, который мы использовали с Redis) с высокой доступностью, которая обеспечивала бы низкую задержку в межрегиональной конфигурации, нетривиальная задача. Разумеется, один экземпляр Redis не обеспечит всего, что вы рекламируете своим клиентам. К счастью, недавно к вам обратился поставщик облачных вычислений ACME Corp и предложил бета-версию одного из своих новейших продуктов. Он называется ACME Global HashMap и делает именно то, что вам нужно. Но есть одна проблема: продукт пока существует в бета-версии, поэтому ACME Corp в соответствии со своей политикой не предоставляет изолированной среды, которую можно было бы применять для целей тестирования. И по каким-то малопонятным юридическим причинам вам нельзя использовать для автоматизированных процедур тестирования эндпоинт в среде реальной эксплуатации.

Что же делать? Ваш код разрастается с каждым днем. В запланированном классе `AcmeStorageBackend`, скорее всего, появится дополнительный код для ведения журнала, телеметрии, управления доступом и прочих заумных штук. Вы совершенно точно хотите, чтобы класс можно было тщательно протестировать. И тогда вы решаете использовать суррогат вместо ACME Corp SDK, который собираетесь интегрировать в свой продукт.

ACME Corp Python SDK существует в виде пакета `acme_sdk`. Среди прочего в нем есть два интерфейса:

```
from typing import Dict

class AcmeSession:
    def __init__(self, tenant: str, token: str): ...

class AcmeHashMap:
    def __init__(self, acme_session: AcmeSession): ...
```

```
    def incr(self, key: str, amount):
        """Увеличивает любой ключ на заданное значение amount"""
        ...

    def atomic_incr(self, key: str, amount):
        """Увеличивает любой ключ на заданное значение amount в атомарном
режиме"""
        ...

    def top_keys(self, count: int) -> Dict[str, int]:
        """Возвращает ключи с наибольшими значениями"""
        ...
```

Объект сеанса `AcmeSession` инкапсулирует подключение к службе ACME Corp, а `AcmeHashMap` — это клиент службы, который мы хотим использовать. Чтобы увеличивать счетчики просмотра, скорее всего, мы будем применять метод `atomic_incr()`. Метод `top_keys()` позволит нам получать страницы с наибольшей частотой обращений.

Чтобы создать суррогат, достаточно определить новый класс с интерфейсом, который совместим с нашим использованием `AcmeHashMap`. Прагматичный подход заключается в том, чтобы реализовать только те классы и методы, которые вы собираетесь использовать. Минимальная реализация `AcmeHashMapFake` может выглядеть так:

```
from collections import Counter
from typing import Dict

class AcmeHashMapFake:
    def __init__(self):
        self._counter = Counter()

    def atomic_incr(self, key: str, amount):
        self._counter[key] += amount

    def top_keys(self, count: int) -> Dict[str, int]:
        return dict(self._counter.most_common(count))
```

Можно воспользоваться классом `AcmeHashMapFake`, чтобы создать новую фикстуру в существующем наборе тестов для подсистем хранения данных. Допустим, в модуле `backends` есть класс `AcmeBackend`, который принимает единственный входной аргумент — экземпляр `AcmeHashMapFake`. Тогда можно предоставить такие две функции фикстуры `pytest`:

```
from backends import AcmeBackend
from acme_fakes import AcmeHashMapFake

@pytest.fixture
def acme_client():
```

```
    return AcmeHashMapFake()

@pytest.fixture
def acme_backend(acme_client):
    return AcmeBackend(acme_client)
```

Фаза подготовки разбивается на две фикстуры с расчетом на то, что может произойти в ближайшем будущем. Когда вы наконец получите доступ к изолированной среде ACME Corp, останется изменить только одну фикстуру:

```
from acme_sdk import AcmeHashMap, AcmeSession

@pytest.fixture
def acme_client():
    return AcmeHashMap(AcmeSession(..., ...))
```

Подведем итог: суррогаты предоставляют эквивалентное поведение для объекта, к которому мы не можем или просто не хотим обращаться во время тестирования. Это особенно полезно в ситуациях, когда вам приходится взаимодействовать с внешними службами или запрашивать удаленные ресурсы. Если неревести эти ресурсы на внутренний уровень, можно будет эффективнее контролировать среду тестирования, а значит, лучше изолировать тестируемую единицу кода.

Городить огород из многочисленных суррогатов может стать утомительным и однообразным делом. К счастью, в библиотеке Python есть модуль `unittest.mock`, который позволяет автоматизировать создание суррогатных объектов.

Mock-объекты и модуль unittest.mock

Mock-объекты — это обобщенные суррогатные объекты, с помощью которых можно изолировать тестируемый код. Они автоматизируют построение ввода и вывода суррогатного объекта. Mock-объекты больше распространены в языках со статической типизацией, где сложнее устроить «горячую подмену», однако в Python они тоже могут пригодиться, чтобы уменьшить объем кода, который моделирует внешние API.

В Python доступно множество библиотек mock-объектов, самой популярной из которых является `unittest.mock`, входящая в стандартную библиотеку.

Библиотека `unittest.mock` изначально была создана в виде стороннего пакета, доступного в PyPI. Через некоторое время она была включена в стандартную библиотеку как временный пакет. Чтобы больше узнать о временных пакетах, обращайтесь по адресу https://docs.python.org/dev/glossary.html#term-provisional-api.

Mock-объекты почти всегда можно использовать вместо специализированных суррогатных объектов. Они особенно полезны, чтобы имитировать внешние компоненты и ресурсы, которые не находятся под вашим полным контролем во время тестирования. Также они становятся бесценным инструментом, когда приходится нарушать главный принцип TDD, — то есть писать тесты после того, как реализация уже готова.

В предыдущем разделе уже рассматривался пример, где мы имитировали уровень подключения к внешнему ресурсу.

Теперь поближе рассмотрим ситуацию, когда приходится писать тест к уже существующему коду, для которого до сих пор не было тестов.

Допустим, у нас есть функция `send()`, которая должна отправлять сообщения электронной почты по протоколу SMTP:

```
import smtplib
import email.message

def send(
    sender, to,
    subject='None',
    body='None',
    server='localhost'
):
    """отправляет сообщение"""
    message = email.message.Message()
    message['To'] = to
    message['From'] = sender
    message['Subject'] = subject
    message.set_payload(body)

    client = smtplib.SMTP(server)
    try:
        return client.sendmail(sender, to, message.as_string())
    finally:
        client.quit()
```

Ситуация усугубляется тем, что функция создает собственный экземпляр `smtplib.SMTP`, который очевидно представляет клиентское подключение SMTP. Если бы мы сначала писали тесты, то, вероятно, подумали бы об этом заранее и использовали небольшую инверсию управления, чтобы передавать клиент SMTP в аргументе функции. Но что сделано, то сделано. Функция `send()` уже используется во всей кодовой базе, а заниматься рефакторингом мы еще не готовы. Сначала нужно протестировать код.

Функция `send()` находится в модуле `mailer`. Начнем с подхода по принципу черного ящика и будем считать, что никакая фаза подготовки не нужна. Созда-

дим тест, который наивно пытается вызвать функцию и надеется на успех. Первая итерация будет выглядеть так:

```
from mailer import send

def test_send():
    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

Если только у вас не запущен локальный сервер SMTP, при выполнении `pytest` вы получите такой результат:

```
$ py.test -v --tb line
======================= test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
pytest-mutagen-1.3 : Mutations disabled
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/05 -
Mocks and unittest.mock module
plugins: mutagen-1.3
collected 1 item

test_mailer.py::test_send FAILED                                  [100%]

============================ FAILURES ==============================
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/socket.
py:831: ConnectionRefusedError: [Errno 61] Connection refused
======================= short test summary info ====================
FAILED test_mailer.py::test_send - ConnectionRefusedError: [Errno 61...
======================== 1 failed in 0.05s =========================
```

С помощью параметра `--tb` команды `py.test` можно управлять длиной трассировочного вывода для непрошедших тестов. В данном случае используется значение `--tb line`, которое позволяет получать однострочные трассировки. Другие возможные значения — `auto`, `long`, `short`, `native` и `no`.

Наши падежды рассеиваются. Функция `send` аварийно завершается с исключением `ConnectionRefusedError`. Если вы не хотите запускать сервер SMTP локально или отправлять реальные сообщения через настоящий сервер SMTP, нридется найти способ заменить реализацию `smtplib.SMTP` суррогатным объектом.

Чтобы достичь цели, мы используем два приема:

- **«Горячая подмена»**. Во время запуска теста изменим модуль `smtplib` на ходу, чтобы заставить функцию `send()` использовать суррогатный объект вместо класса `smtplib.SMTP`.
- **Mock-объект**. Создадим универсальный mock-объект, который умеет имитировать абсолютно любой объект. Это будет сделано исключительно для упрощения работы.

Прежде чем изучать оба приема подробнее, рассмотрим пример тестовой функции:

```
from unittest.mock import patch
from mailer import send

def test_send():
    sender = "john.doe@example.com"
    to = "jane.doe@example.com"
    body = "Привет, Джейн!"
    subject = "Как дела?"

    with patch('smtplib.SMTP') as mock:
        client = mock.return_value
        client.sendmail.return_value = {}

        res = send(sender, to, subject, body)

        assert client.sendmail.called
        assert client.sendmail.call_args[0][0] == sender
        assert client.sendmail.call_args[0][1] == to
        assert subject in client.sendmail.call_args[0][2]
        assert body in client.sendmail.call_args[0][2]
        assert res == {}
```

Диспетчер контекстов `unittest.mock.path` создает новый экземпляр класса `unittest.mock.Mock` и подставляет его по заданному пути импортирования. Когда функция `send()` пытается обратиться к атрибуту `smtplib.SMTP`, она получает экземпляр mock-объект вместо объекта класса `SMTP`.

Mock-объекты способны на «черную магию». Если обратиться к любому атрибуту такого объекта за пределами набора имен, зарезервированного модулем `unittest.mock`, он вернет новый экземпляр mock-объекта. Mock-объекты также можно использовать как функции, и при вызове они тоже возвращают новый экземпляр объекта.

Фупкция `send()` ожидает, что `stmptlib.SMTP` будет объектом типа, поэтому она вызывает `SMTP()`, чтобы получить экземпляр объекта клиента SMTP. Мы используем значение `mock.return_value` (`return_value` — одно из зарезервирован-

ных имен), чтобы получить имитацию этого клиентского объекта и управлять возвращаемым значением метода `client.sendmail()`.

После выполнения функции `send()` мы используем еще два зарезервированных имени (`called` и `call_args`), чтобы проверить, был ли вызван метод `client.sendmail()`, и проанализировать аргументы вызова.

То, что мы здесь проделали, вряд ли можно назвать удачным решением, — ведь по сути мы просто повторили то, что делает реализация функции `send()`. Избегайте подобных повторений в собственных тестах, потому что тест, который просто пересказывает реализацию тестируемой функции, не имеет смысла. Однако наша цель состояла скорее в том, чтобы продемонстрировать возможности модуля `unittest.mock`, а не в том, чтобы показать, как надо писать тесты.

Диспетчер контекстов `patch()` из модуля `unittest.mock` — один из инструментов, позволяющих динамически подменять пути импортирования во время запуска тестов. Его также можно использовать как декоратор. Это довольно замысловатый механизм, так что не всегда удается легко подменить то, что нужно. Кроме того, если понадобится подменить сразу несколько объектов, потребуется конструировать вложенные объекты, что может оказаться довольно неудобно.

`pytest` предлагает альтернативный способ «горячей подмены» — встроенную фикстуру `monkeypatch`, которая работает как посредник при подмене. Если вы захотите переписать предыдущий пример с помощью фикстуры `monkeypatch`, это можно сделать так:

```
import smtplib
from unittest.mock import Mock
from mailer import send

def test_send(monkeypatch):
    sender = "john.doe@example.com"
    to = „jane.doe@example.com"
    body = "Привет, Джейн!"
    subject = "Как дела?"

    smtp = Mock()
    monkeypatch.setattr(smtplib, "SMTP", smtp)
    client = smtp.return_value
    client.sendmail.return_value = {}
    res = send(sender, to, subject, body)

    assert client.sendmail.called
    assert client.sendmail.call_args[0][0] == sender
    assert client.sendmail.call_args[0][1] == to
```

```
    assert subject in client.sendmail.call_args[0][2]
    assert body in client.sendmail.call_args[0][2]
    assert res == {}
```

«Горячей подменой» и mock-объектами часто злоупотребляют, особенно если тесты пишутся после реализации. Поэтому лучше обходиться без этих средств, когда есть другие способы надежного тестирования кода. В противном случае у вас может оказаться проект со множеством тестов, которые представляют собой пустые оболочки и на самом деле не проверяют правильность кода. Нет ничего опаснее ложного чувства безопасности. Также есть риск того, что mock-объекты будут предполагать поведение, отличное от реального.

В следующем разделе рассматривается тема автоматизации контроля качества, близкая по смыслу к TDD.

Автоматизация контроля качества

Не существует четкой системы критериев, по которым можно было бы уверенно судить, плох или хорош тот или иной код. К сожалению, абстрактную концепцию качества кода нельзя измерить и выразить в числовой форме. Вместо этого приходится измерять различные метрики, которые соотносятся с качеством кода. Вот несколько примеров таких метрик:

- Степень покрытия кода тестами.
- Количество нарушений стиля программирования.
- Объем документации.
- Метрики сложности, например цикломатическая сложность.
- Количество предупреждений при статическом анализе кода.

Во многих проектах проверка качества кода включена в рабочие процессы непрерывной интеграции. Популярная практика состоит в том, чтобы тестировать как минимум базовые метрики (тестовое покрытие, статический анализ кода и нарушения стиля программирования) и не объединять с основной ветвью любой код, который показывает недостаточно хорошие результаты по этим метрикам.

В следующих разделах мы рассмотрим некоторые интересные инструменты и методы, которые позволяют автоматизировать вычисление отдельных метрик качества кода.

Тестовое покрытие

Тестовое покрытие (или покрытие кода тестами) — чрезвычайно полезная метрика, которая предоставляет объективную информацию о том, насколько тща-

тельно протестирован тот или иной объем исходного кода. Фактически эта метрика просто показывает, сколько строк кода выполняется в ходе тестов и что это за строки. Она часто выражается в процентах; покрытие 100 % означает, что в процессе тестирования была выполнена каждая строка кода.

Самое популярное средство оценки покрытия для кода Python — пакет `coverage`, свободно доступный в PyPI. Пользоваться им очень просто, процесс состоит всего из двух шагов:

1. Запустить набор тестов с помощью `coverage`.
2. Получить отчет `coverage` в нужном формате.

На первом шаге надо запустить в оболочке команду `coverage run` с путем к сценарию или программе, которая выполняет все тесты. Для `pytest` это может выглядеть примерно так:

```
$ coverage run $(which pytest)
```

`which` — полезная утилита командной строки, которая возвращает в стандартный вывод путь к исполняемому файлу другой команды. Во многих оболочках выражение `$()` можно использовать, чтобы подставить вывод команды как значения в текст другой команды.

Команду `coverage run` также можно запустить с флагом `-m`, который задает исполняемый модуль. Это аналогично тому, как исполняемые модули запускаются командой `python -m`. Как пакет `pytest`, так и модуль `unittest` предоставляют свои исполнители тестов в виде исполняемых модулей:

```
$ python -m pytest
$ python -m unittest
```

Таким образом, чтобы запустить тестовые наборы под контролем `coverage`, можно воспользоваться следующими командами:

```
$ python -m pytest
$ python -m unittest
```

По умолчанию `coverage` измеряет тестовое покрытие каждого модуля, который импортируется во время прохождения теста. Следовательно, в число таких модулей могут попасть внешние пакеты, установленные в виртуальной среде вашего проекта. Обычно имеет смысл измерять тестовое покрытие только для исходного кода проекта, без внешних источников. Команда `coverage` принима-

ет параметр `--source`, который позволяет ограничить анализ конкретными путями, как в следующем примере:

```
$ coverage run --source . -m pytest
```

Библиотека coverage позволяет задать флаги конфигурации в файле setup.cfg. Его содержимое для приведенного выше запуска coverage будет выглядеть так:

```
[coverage:run]
source =
    .
```

Во время выполнения теста `coverage` создает файл `.coverage` с промежуточными результатами измерений. После выполнения результаты можно просмотреть с помощью команды `coverage report`.

Чтобы увидеть, как измерение тестового покрытия работает на практике, допустим, что вы решили создать специализированное расширение для одного из классов, упомянутых в разделе «Фикстуры pytest», но не смогли правильно протестировать его. Добавьте метод `count_keys()` в класс `CounterClass`:

```
class CounterBackend(ViewsStorageBackend):
    def __init__(self):
        self._counter = Counter()

    def increment(self, key: str):
        self._counter[key] += 1

    def most_common(self, n: int) -> Dict[str, int]:
        return dict(self._counter.most_common(n))

    def count_keys(self):
        return len(self._counter)
```

Метод `count_keys()` не был включен в объявление интерфейса (абстрактный базовый класс `ViewsStorageBackend`), поэтому мы не предусмотрели его, когда писали тестовый набор.

Запустим тесты с помощью `coverage` и посмотрим на результаты. Следующий фрагмент сеанса оболочки показывает, как они могут выглядеть:

```
$ coverage run -source . -m pytest -q
................

[100%]
16 passed in 0.12s
$ coverage report -m
```

```
Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
backends.py            21      1    95%   19
interfaces.py           7      0   100%
test_backends.py       39      0   100%
-------------------------------------------------
TOTAL                  67      1    99%
```

Все параметры и флаги после параметра `-m <модуль>` в команде `coverage run` будут переданы непосредственно запуску исполняемого модуля. Флаг исполнителя `pytest -q` сообщает, что мы хотим получить сокращенный отчет о прохождении тестов.

Как видите, все тесты прошли успешно, но отчет показывает, что модуль backends.py покрыт тестами на 95 %. Это означает, что 5 % строк не были выполнены во время прохождения тестов. Таким образом, в нашем тестовом наборе возник пробел.

В столбце `Missing` (за который отвечает флаг `-m` команды `coverage report`) показаны номера строк, пропущенных при прохождении теста. Для небольших модулей с высоким покрытием этого бывает достаточно, чтобы найти непокрытые участки. При очень низком покрытии, вероятно, понадобится более подробный отчет.

Пакет `coverage` также поддерживает комапду `coverage html`, которая генерирует интерактивный отчет о покрытии в формате HTML.

Тестовое покрытие — очень хорошая метрика, тесно связанная с общим качеством кода. По статистике у проектов с низким тестовым покрытием бывает больше дефектов и проблем с качеством. У проектов с высоким покрытием таких недостатков обычно меньше — при условии, что тесты написаны в соответствии с базовыми принципами, которые перечислены в разделе «Принципы разработки через тестирование».

Даже проекты со 100-процентным покрытием могут вести себя непредсказуемо и пестрить ошибками. В таких ситуациях иногда приходится применять методы, которые проверяют полезность существующих тестовых наборов и выявляют пропущенные условия тестирования. Один из таких методов описан в разделе «Мутационное тестирование».

Как бы то ни было, очень просто написать бессмысленные тесты, которые при этом заметно повышают тестовое покрытие. Всегда внимательно анализируйте результаты тестового покрытия новых проектов и не рассматривайте эти результаты как безусловную характеристику качества кода проекта.

Рис. 10.1. Пример отчета о покрытии в формате HTML с выделенными пропусками

Кроме того, качество кода определяется не только тем, насколько тщательно он протестирован, но и тем, насколько его удобно читать, сопровождать и расширять. Значит, контроль качества также должен учитывать стиль программирования, соблюдение общепринятых соглашений, удобство повторного использования кода и безопасность. К счастью, измерения и проверки в этих областях программирования можно в определенной степени автоматизировать.

В этом разделе продемонстрировано «классическое» использование пакета `coverage`. Если вы работаете с `pytest`, измерение тестового покрытия можно оптимизировать с помощью плагина `pytest-cov`, который умеет автоматически добавлять `coverage run` к каждому запуску теста. За дополнительной информацией о `pytest-cov` обращайтесь по адресу https://github.com/pytest-dev/pytest-cov.

Начнем с того, как автоматизировать проверку стиля и применять средства статического анализа. Это самый типичный инструментарий контроля качества кода, который используют профессиональные программисты на Python.

Средства проверки стиля программирования и статического анализа кода

Код сложнее читать, чем писать, — это утверждение справедливо для всех языков программирования. Программный код вряд ли будет качественным, если он написан непоследовательно, с причудливым форматированием или экзотическими соглашениями стиля. Дело не только в том, что такой код сложнее прочитать и попять, по и в том, что его будет трудно расширять или сопровождать в разумном темпе, ведь качество кода связано не только с его текущим состоянием, но и с потенциальным будущим.

Чтобы улучшить целостность кодовой базы, программисты используют так называемые **статическне анализаторы (линтеры)**, которые проверяют стиль программирования и соблюдение соглашений по написанию кода. Эти инструменты также могут выявлять на нервый взгляд безобидные, но потенциально проблемные конструкции, в том числе:

- неиспользуемые переменные или команды `import`;
- доступ к защищенным атрибутам за пределами их класса;
- переопределение существующих функций;
- небезопасное использование глобальных переменных;
- неправильный порядок секций `except`;
- норождение некорректных типов исключений.

Хотя обобщающим термином «статические анализаторы» («липтеры») называются любые инструменты, которые обнаруживают стилевые ошибки и непоследовательность, а также подозрительные и потенциально опасные участки кода, в последние годы в этой области появилась некоторая специализация. В сообществе Python обычно используются две основные разновидности линтеров:

- **Корректоры стнля**: линтеры, ориентированные на проверку соглашений и конкретных стилевых рекомендаций. Для Python это могут быть рекомендации PEP 8 или любые другие соглашения. Корректоры способны находить стилевые ошибки и автоматически править их. Примеры популярных корректоров для Python — `black`, `autopep8` и `yapf`. Также бывают узкоспециализированные корректоры, которые концептрируются только на одном аспекте стиля. Яркий пример такого рода — система `isort`, предназначенная для сортировки команд `import`.
- **Классические линтеры**: это анализаторы, в большей степени ориентированные на подозрительные и опасные конструкции, которые могут привести к ошибкам и/или нештатному поведению программы. Впрочем, эти инстру-

менты могут также включать наборы правил для конкретных стилевых соглашений. Классические линтеры обычно работают в режиме выявления дефектов: они помечают возможные проблемы, но не исправляют их автоматически. Примеры популярных классических линтеров Python — `pylint` и `pyflakes`. Также широко распространен чисто стилевой линтер `pycodestyle`.

Бывают и экспериментальные гибридные линтеры, которые умеют автоматически исправлять подозрительные программные конструкции — например, `autoflake`. К сожалению, из-за сложной природы этих подозрительных конструкций (таких, как неиспользуемые инструкции `import` или переменные) не всегда можно безопасно исправить их, не создавая побочных эффектов. Такими анализаторами следует пользоваться крайне осторожно.

Как корректоры стиля, так и классические линтеры чрезвычайно важны при написании высококачественных программ, особенно в профессиональной разработке. Популярные классические линтеры, такие как `pyflakes` и `pylint`, содержат многочисленные правила для ошибок, предупреждений и автоматических рекомендаций, и список этих правил постоянно расширяется.

Большая подборка правил означает, что введение одного из этих линтеров в уже существующий крупный проект обычно требует некоторой настройки, чтобы привести код к стандартным соглашениям. Возможно, какие-то правила покажутся вам довольно произвольными (например, длина строки по умолчанию, конкретные паттерны импортирования или максимальное количество аргументов функций) и вы решите отключить некоторые из проверок по умолчанию. Настройка линтеров требует определенных усилий, но в долгосрочной перспективе они, безусловно, оправдаются.

Как бы то ни было, настраивать линтеры — утомительная задача, поэтому мы не будем подробнее погружаться в нее. И у `pylint`, и у `pyflakes` есть отличная документация, где доступно описано их использование. Корректоры определенно интереснее, чем классические линтеры. Обычно они требуют минимальной настройки, а иногда не требуют вообще.

Корректоры позволяют в значительной мере унифицировать кодовую базу, запустив всего одну команду. Чтобы продемонстрировать их возможности, мы воспользуемся архивом примеров кода для этой книги.

Эти примеры написаны в основном в соответствии с рекомендациями по стилю PEP 8. При этом нам пришлось внести ряд изменений просто для того, чтобы код был понятным, компактным и хорошо читался на книжной странице:

- **Одна пустая строка вместо двух**. В некоторых местах PEP 8 рекомендует вставлять две пустые строки, в основном для отделения функций, методов

и классов. Мы решили ограничиться одной пустой строкой, чтобы сэкономить место и чтобы длинные примеры не приходилось переносить на следующую страницу.

- **Меньшая длиа строки**. Согласно PEP 8, предельная длина строки равна 79 символам. Это не так много, но как выясняется, для книги и это перебор. Книжная страница обычно ориентирована вертикально, и 79–80 символов в стандартном моноширинном шрифте не умещаются в нолосу набора. Кроме того, некоторые читатели открывают книгу на электронных устройствах, где автор не контролирует, как выводятся примеры кода. Короткие строки повышают вероятность того, что примеры на бумаге и на экране будут выглядеть одинаково.
- **Отказ от группировки ииструкций импортпроваиия**: PEP 8 рекомендует разбивать инструкции имнортировапия на группы — модули стандартной библиотеки, сторонние модули и локальные модули — и разделять их одной пустой строкой. Эта рекомендация оправданна для программных модулей, однако в книге, где в одном примере редко используется более двух инструкций импортирования, это только создало бы лишний шум и драгоценное место на страницах расходовалось бы неэффективно.

Эти пебольшие отклонения от рекомендаций PEP 8 безусловпо оправданны для книжного формата. Однако те же фрагменты кода также доступны в репозитории Git, который сопровождает эту книгу. Если вы откроете примеры в своей любимой IDE, только чтобы увидеть, как неестественно выглядит код, оптимизированный для книги, вы испытаете определенный дискомфорт. Таким образом, для Git надо было нодготовить примеры кода, оптимизированные для вывода на экран компьютера.

В исходном коде книги свыше 100 файлов на Python, и если бы мы поддерживали их независимо в двух стилях, это повышало бы риск ошибок и занимало бы много времени. Мы ноступили по-другому, работая над примерами кода в репозитории Git в неформальном книжпом формате. Каждую главу просматривали несколько редакторов, так что отдельные примеры приходилось неоднократно обновлять. Когда мы убедились, что весь код правильный и работает как надо, мы просто воспользовались программой `black`, которая обпаружила все нарушения стиля и автоматически исправила их.

Программа `black` используется достаточно тривиально. Она занускается командой `black <источник>`, где `<источник>` — нуть к исходному файлу или каталогу с исходными файлами, которые вы хотите переформатировать. Чтобы обработать все исходные файлы в текущем рабочем каталоге, используйте команду:

```
$ black .
```

Если вы запустите `black` для репозитория примеров из этой книги, то получите результат следующего вида:

```
(...)
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/01 - One step deeper: class decorators/autorepr.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 6/07 - Throttling/throttling.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/01 - One step deeper: class decorators/autorepr_
subclassed.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 7/04 - Subject-based style/observers.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/04 - Using __init__subclass__ method as alternative
to metaclasses/autorepr.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 9/06 - Calling C functions using ctypes/qsort.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/04 - Using __init__subclass__ method as alternative
to metaclasses/autorepr_with_init_subclass.py
All done ! ✨🍰✨
64 files reformatted, 37 files left unchanged.
```

Реальный вывод команды `black` был в несколько раз длиннее, так что нам пришлось его значительно сократить.

То, что потребовало бы многих часов утомительной работы, благодаря программе `black` получилось всего за несколько секунд.

Конечно, нельзя рассчитывать, что каждый разработчик в проекте будет последовательно запускать `black` для каждого изменения, которое фиксируется в центральном репозитории. Поэтому `black` можно запускать в проверочном режиме с флагом `--check`. Таким образом, `black` также можно использовать на стадии проверки стиля в общих системах сборки, обеспечивая непрерывную интеграцию изменений.

Такие программы, как `black`, безусловно, повышают качество кода, обеспечивая его простое и последовательное форматирование. Благодаря этому код проще читается и становится (хочется надеяться) более понятным. Другое преимущество заключается в том, что экономится значительное время, которое было бы потрачено впустую на споры о форматировании кода. Впрочем, это лишь один небольшой аспект качества. Никто не гарантирует, что в хорошо отформатированном коде будет меньше ошибок или что его сопровождение ощутимо упростится.

По части выявления дефектов и проблем сопровождаемости классические линтеры обычно намного лучше любых автоматических корректоров. Есть категория линтеров, которые особенно хорошо умеют обнаруживать проблемные части кода, — это линтеры, способные выполнять статический анализ типов. Один из таких инструментов более подробно рассматривается в следующем разделе.

Статический анализ типов

Python не является языком со статической типизацией, но поддерживает необязательные аннотации типов. Эта возможность в сочетании с высокоспециализированными статическими анализаторами может сделать код на Python почти таким же типобезопасным, как на классических языках со статической типизацией.

Есть еще одно преимущество того, что аннотации типов необязательны. Вы можете в любой момент решить, применять их или нет. Типизированные аргументы, переменные-функции и методы помогают обеспечивать логическую целостность и предотвращать глупые ошибки, но они же могут помешать, если вы пытаетесь сделать что-то необычное. Иногда вам нужно просто добавить дополнительный атрибут в уже существующий экземпляр путем «горячей подмены» или подправить стороннюю библиотеку, которая плохо себя ведет. Конечно, такие вещи проще делать, если язык не требует принудительной проверки типов.

Самое популярное средство статической проверки типов для Python — пакет `mypy`. Он анализирует аннотации функций и переменных, которые могут определяться с использованием иерархии аннотаций типов из модуля `typing`. Чтобы работать с `mypy`, вам не обязательно размечать весь код аннотациями. Эта особенность `mypy` особенно полезна при сопровождении унаследованных кодовых баз, потому что аннотации типов можно добавлять постепенно.

За дополнительной информацией об иерархии типов в Python обращайтесь к документу «PEP 484 — Аннотации типов», который доступен по адресу https://www.python.org/dev/peps/pep-0484/.

Как и в случае других статических анализаторов, базовое использование `mypy` весьма прямолинейно. Просто напишите свой код (с аннотациями типов или без) и проверьте его правильность командой `mypy <путь>`, где `<путь>` — исходный файл или каталог, содержащий несколько исходных файлов. `mypy` распознает части кода с аннотациями типов и проверяет, что использование функций и переменных соответствует объявленным типам.

Хотя `mypy` — это независимый пакет, доступный в PyPI, аннотации типов для статического анализа полностью поддерживаются массовой разработкой Python в виде проекта Typeshed. Typeshed (https://github.com/python/typeshed) — набор библиотечных заглушек со статическими определениями типов как для стандартной библиотеки, так и для многих популярных сторонних проектов.

Дополнительная информация о пакете `mypy` и о том, как использовать его в командной строке, доступна на официальной странице проекта по адресу http://mypy-lang.org.

К этому моменту мы обсудили тему автоматизации контроля качества в контексте кода приложений. Мы использовали тесты, чтобы повысить общее качество кода, и измеряли тестовое покрытие, чтобы проверить, хорошо ли написаны тесты. Однако мы пока не говорили о качестве самих тестов, а оно не менее важно, чем качество тестируемого кода. Плохие тесты создают ложное ощущение безопасности и качества кода, а это может быть почти так же вредно, как полное отсутствие тестов.

В общем случае базовые инструменты автоматизации контроля качества можно применять и к коду тестов. Это означает, что с помощью статических анализаторов и корректоров стиля можно сопровождать кодовую базу тестов. Однако эти инструменты не дают никаких количественных характеристик того, насколько эффективно тесты обнаруживают новые и существующие ошибки. Нужны другие методы, чтобы измерять эффективность и качество тестов. Один из таких методов — мутационное тестирование, о котором мы сейчас поговорим.

Мутационное тестирование

Хорошо, когда в вашем проекте достигается стопроцентное тестовое покрытие. Но чем оно выше, тем быстрее вы осознаете, что покрытие не гарантирует надежности кода. Во множестве проектов с высоким покрытием новые ошибки обнаруживаются в частях кода, уже покрытых тестами. Как это получается?

Причины могут быть разными. Иногда требования недостаточно ясны, а тесты покрывают не то, что нужно. Иногда в тестах присутствуют ошибки. В конце концов, тесты подвержены ошибкам так же, как и любой другой код.

Но иногда проблема тестов оказывается в том, что они представляют собой пустые оболочки: они формально выполняют блоки кода и сравнивают результаты, но на самом деле не стремятся проверить правильность кода. Как ни

странно, в эту ловушку проще понасть, если вы чрезмерно беспокоитесь о качестве и оценке тестового покрытия. Такими пустыми оболочками часто становятся тесты, которые написаны на завершающей стадии просто для того, чтобы обеспечить идеальное покрытие.

Один из способов проверить качество тестов — намеренно вносить в код изменения, которые заведомо должны нарушить его работу, и смотреть, обнаружат ли тесты проблему. Если хотя бы один тест не проходит, можно сделать вывод, что им удалось выявить эту конкретную ошибку. Если все тесты проходят успешно, то вам, вероятно, стоит переработать тестовый набор.

Возможности для ошибок бесчисленны, поэтому эту процедуру трудно выполнять достаточно часто и многократно без специальных инструментов и методологий. Одна из таких методологий — мутационное тестирование.

Мутационное тестирование опирается на гипотезу о том, что большинство дефектов в коде возникает из-за мелких ошибок вроде смещения на единицу, перепутанных операторов сравнения, неправильных диапазонов и т. д. Также предполагается, что эти мелкие ошибки накапливаются и приводят к более серьезным сбоям, которые должны распознаваться тестами.

В мутационном тестировании используются **мутации** — четко определенные модификации, которые моделируют типичные мелкие ошибки программистов. Некоторые примеры мутаций:

- Заменить оператор `==` оператором `is`.
- Заменить литерал `0` литералом `1`.
- Переставить местами операнды оператора `<`.
- Добавить суффикс к строковому литералу.
- Заменить команду `break` командой `continue`.

На каждом этапе мутационного тестирования исходная программа слегка изменяется и становится так называемым **мутантом**. Если мутант проходит все тесты, говорят, что он выжил в процессе тестирования. Если хотя бы один тест не прошел, говорят, что мутант погиб. Цель мутационного тестирования — укрепить набор тестов, чтобы он не позволял мутантам выживать.

Вся эта теория выглядит немного абстрактно, поэтому рассмотрим практический пример мутационного тестирования. Попробуем протестировать функцию `is_prime()`, которая проверяет, является ли целое число простым.

Простым числом называется натуральное число, большее 1, которое делится только на самого себя и на 1. Существует только один простой способ протестировать функцию `is_prime()`: передать ей тестовые данные. Начнем с простого теста:

```
from primes import is_prime

def test_primes_true():
    assert is_prime(5)
    assert is_prime(7)

def test_primes_false():
    assert not is_prime(4)
    assert not is_prime(8)
```

Здесь можно было бы воспользоваться параметризацией, но оставим это на потом. Сохраним код в файле test_primes.py и перейдем к фупкции `is_prime()`. Сейчас мы стремимся к простоте, поэтому начнем с очепь наивпой реализации:

```
def is_prime(number):
    if not isinstance(number, int) or number < 0:
        return False

    if number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False
    return True
```

Это пе самая эффективная реализация, но она предельно проста, поэтому ее легко попять. Простыми могут быть только целые числа, большие 1. Мы начипаем с того, что проверяем тип и сравпиваем аргумент со значениями 0 и 1. Для остальпых чисел мы перебираем целые числа, которые меньше `number` и больше 1. Если `number` не делится ни на одно из этих чисел, знпачит, это простое число. Сохраним эту функцию в файле primes.py.

Пришло время оценить качество наших тестов. В PyPI есть несколько инструментов для мутациопного тестирования. Пожалуй, самый простой из них — `mutmut`, и мы воспользуемся им в пашем сеансе тестирования. `mutmut` требует определить вторичпую копфигурацию, которая описывает, как должны запускаться тесты и как должен мутировать код. Для этого используется специальный раздел `[mutmut]` в стандартном файле setup.cfg. В нашем примере конфигурация будет выглядеть так:

```
[mutmut]
paths_to_mutate=primes.py
runner=python -m pytest -x
```

Переменная `paths_to_mutate` задает пути к исходпым файлам, в которых `mutmut` сможет примепять мутации. Мутационное тестирование в больших проектах может занять значительпое время, поэтому, чтобы его сэкономить, важпо направить `mutmut` к тому коду, который должен мутировать.

Переменная `runner` задает команду, которая будет использоваться для запуска тестов. Пакет `mutmut` нейтрален по отношению к фреймворкам, поэтому он поддерживает любые виды тестовых фреймворков, исполнитель которых можно запускать как команду оболочки. В данном случае используется `pytest` с флагом `-x`. Этот флаг заставляет `pytest` прервать тестирование при первом сбое. Вся суть мутационного тестирования в том, чтобы выявлять выживших мутантов. Если какие-либо тесты не проходят, то сразу понятно, что мутант не выжил.

Пришло время запустить сеанс мутационного тестирования. `mutmut` используется практически так же, как и `coverage`, поэтому работа начинается с подкоманды `run`:

```
$ mutmut run
```

Выполнение занимает пару секунд. После того как `mutmut` выполнит проверку мутантов, вы получите сводку следующего вида:

```
- Mutation testing starting -

These are the steps:
1. A full test suite run will be made to make sure we
   can run the tests successfully and we know how long
   it takes (to detect infinite loops for example)
2. Mutants will be generated and checked

Results are stored in .mutmut-cache.
Print found mutants with `mutmut results`.

Legend for output:
🎉 Killed mutants.   The goal is for everything to end up in this
bucket.
⏰ Timeout.          Test suite took 10 times as long as the baseline so
were killed.
🤔 Suspicious.       Tests took a long time, but not long enough to be
fatal.
😐 Survived.         This means your tests needs to be expanded.
🔇 Skipped.          Skipped.

1. Running tests without mutations
⠹ Running...Done

2. Checking mutants
⠼ 15/15  🎉 8  ⏰ 0  🤔 0  😐 7  🔇 0
```

В носледней строке содержится краткая сводка результатов. Чтобы получить подробный отчет, можно воспользоваться командой `mutmut results`. В нашем сеансе результат будет выглядеть так:

```
$ mutmut results
To apply a mutant on disk:
    mutmut apply <id>

To show a mutant:
    mutmut show <id>

Survived ☹ (7)

---- primes.py (7) ----

8-10, 12-15
```

В последней строке перечислены идентификаторы мутантов, которые пережили тест. Мы видим, что выжили 7 мутантов с идентификаторами в диапазонах 8–10 и 12–15. В выводе также приводится справочная информация о том, как просматривать мутантов командой `mutmut show <идентификатор>`. Также можно просматривать информацию о целой группе мутантов, если указать имя исходного файла в качестве значения `<идентификатор>`.

Мы делаем это все только для демонстрационных целей, поэтому просмотрим данные всего двух мутантов. Начнем с мутанта с идентификатором 8:

```
$ mutmut results
To apply a mutant on disk:
    mutmut apply <id>

To show a mutant:
    mutmut show <id>

Survived ☹ (7)

---- primes.py (7) ----

8-10, 12-15
```

`mutmut` изменил значения диапазонов в `if number in (...)`, но тесты не обнаружили пикакой проблемы. Возможно, эти значения стоит включить в тестируемые условия.

Теперь посмотрим на последнего мутанта с идентификатором 15:

```
$ mutmut show 15
--- primes.py
+++ primes.py
@@ -1,6 +1,6 @@
 def is_prime(number):
     if not isinstance(number, int) or number < 0:
-        return False
```

```
+         return True

      if number in (0, 1):
          return False
```

mutmut инвертировал значение литерала bool после того, как проверил диапазоны типов и значений. Мутант выжил, потому что мы включили проверку типа, но не проверили, что происходит, если входное значение имеет неправильный тип.

В нашей ситуации всех этих мутантов можно было бы уничтожить, включив в тесты больше проверок. Если расширить тестовый набор новыми граничными случаями и недопустимыми значениями, он может стать более надежным. Ниже приведен обновленный тестовый набор:

```
from primes import is_prime

def test_primes_true():
    assert is_prime(2)
    assert is_prime(5)
    assert is_prime(7)

def test_primes_false():
    assert not is_prime(-200)
    assert not is_prime(3.1)
    assert not is_prime(0)
    assert not is_prime(1)
    assert not is_prime(4)
    assert not is_prime(8)
```

Мутационное тестирование относится к гибридным методологиям, потому что оно не только проверяет качество тестирования, но и может указать на потенциально избыточный код. Например, если улучшить тесты в соответствии с приведенным примером, два мутанта все равно выживут:

```
# мутант 12
--- primes.py
+++ primes.py
@@ -1,5 +1,5 @@
 def is_prime(number):
-    if not isinstance(number, int) or number < 0:
+    if not isinstance(number, int) or number <= 0:
         return False

     if number in (0, 1):

# мутант 13
--- primes.py
+++ primes.py
```

```
@@ -1,5 +1,5 @@
 def is_prime(number):
-    if not isinstance(number, int) or number < 0:
+    if not isinstance(number, int) or number < 1:
         return False

     if number in (0, 1):
```

Эти мутанты выжили, потому что две инструкции `if`, которые мы использовали, потенциально могут обрабатывать одно условие. Это означает, что наш код излишне сложен и его можпо упростить. Оба мутанта погибнут, если свернуть две инструкции `if` в одну:

```
def is_prime(number):
    if not isinstance(number, int) or number <= 1:
        return False

    for element in range(2, number):
        if number % element == 0:
            return False
    return True
```

Мутационное тестирование — весьма интересный метод, который может повысить качество тестов. К сожалению, он плохо масштабируется. В больших проектах оказывается очень много потенциальных мутантов, а для их проверки приходится запускать весь набор тестов. Если в нем много продолжительных тестов, то каждый сеанс мутационного тестирования будет запимать много времени. В результате мутационпое тестировапие хорошо подходит для простых модульных тестов, но его возможности весьма ограничепны в интеграционпом тестировапии. Тем не менее это отличный инструмент для поиска дефектов в тестовых наборах с идеальпым покрытием.

В нескольких предыдущих разделах наше внимание было сосредоточено на инструментах и приемах написания тестов и автоматизации контроля качества. Эти систематические подходы формируют хорошую основу для онераций тестировапия, однако пе гарантируют, что вы будете эффективно писать тесты или что тестировать код будет легко. Тестирование иногда бывает утомительным и однообразным. Более интереспым его делает большая подборка утилит из PyPI, которые позволяют избавиться от рутинной работы.

Полезные средства тестирования

Когда речь заходит о том, как эффективпо писать тесты, все обычно сводится к решению рутинпых или громоздких проблем: как предоставлять реалистичные данные, как учитывать критические ограничепия по времени обработки, как

взаимодействовать с удаленными службами и т. д. Чтобы повысить эффективность своей работы, опытные программисты обычно пользуются большой подборкой специализированных инструментов для решения этих мелких типичных проблем. Рассмотрим некоторые из таких инструментов.

Моделирование реалистичных значений данных

Если мы пишем тесты для кода, который обрабатывает входные или выходные данные, часто требуется предоставить значения, имеющие смысл в приложении, например:

- имена людей;
- почтовые адреса;
- номера телефонов;
- адреса электронной почты;
- идентификаторы (например, ИНН или номер социального страхования).

Проще всего воспользоваться фиксированными значениями. Мы уже делали это в функции `test_send()` из раздела «Mock-объекты и модуль unittest.mock»:

```
def test_send():
    sender = "john.doe@example.com"
    to = "jane.doe@example.com"
    body = "Привет, Джейн!"
    subject = "Как дела?"
    ...
```

Преимущество такого подхода заключается в том, что при чтении кода сразу видно, какие значения используются, так что они служат еще и целям документирования кода. Но у фиксированных значений есть недостаток: они не позволяют тестам эффективно проводить поиск по необъятному пространству потенциальных ошибок. В разделе «Мутационное тестирование» уже было показано, как небольшой набор тестовых данных может привести к дефектным тестам и ложному ощущению безопасности относительно качества вашего кода.

Конечно, проблему можно решить, если параметризировать тесты и использовать более реалистичные выборки данных. Однако это требует большого объема рутинной однообразной работы, которой многие разработчики не желают плотно заниматься.

Чтобы не формировать наборы данных вручную, можно воспользоваться готовыми генераторами данных, которые предоставляют реалистичные значения. Примером такого генератора служит пакет `faker`, доступный в PyPI. `faker` включает встроенный плагин для `pytest` с фикстурой `faker`, которую можно

легко использовать в любом из ваших тестов. Вот измененная часть функции `test_send()`, в которой применяется фикстура `faker`:

```
from faker import Faker

def test_send(faker: Faker):
    sender = faker.email()
    to = faker.email()
    body = faker.paragraph()
    subject = faker.sentence()
    ...
```

При каждом запуске `faker` инициализирует тест разными данными. Благодаря этому увеличиваются шансы обнаружить потенциальные проблемы. А если вы захотите многократно выполнить одни и те же тесты для разных случайных значений, можно воспользоваться следующим приемом параметризации `pytest`:

```
import pytest

@pytest.mark.parametrize("iteration", range(10))
def test_send(faker: Faker, iteration: int):
    ...
```

В `pytest` есть десятки классов поставщиков данных, каждый из которых содержит несколько методов генерации данных. Каждый метод можно вызвать непосредственно через экземпляр класса `Faker`. Этот пакет также поддерживает локализацию, так что многие классы провайдеров доступны в версиях для разных языков.

Кроме того, `faker` может предоставлять значения даты и времени в разных стандартах. К сожалению, он не умеет останавливать время. Но не беспокойтесь, для этого есть другой пакет.

Моделирование значений времени

Может оказаться, что вам по какой-то причине понадобится повлиять на течение времени в вашем приложении. Это может быть полезно при тестировании обработки, завязанной на время, такой как планирование работы или анализ временных меток, которые автоматически присваиваются каким-либо объектам.

Конечно, приложение всегда можно приостановить. В системах POSIX для приостановки процесса используется системный вызов `pause()`, а в Python можно установить точку останова с помощью функции `breakpoint()`. Но все это не влияет на течение времени. Кроме того, когда приложение приостановлено, оно не может продолжить выполнение, а следовательно, вы не сможете продолжить тестирование.

Вместо этого нужно, не вмешиваясь в обычное выполнение кода, создать у него иллюзию, будто время течет в другом темпе или остановилось в какой-то точке. На это способен замечательный пакет `freezegun`, который доступен в PyPI.

Применять `freezegun` достаточно просто. Пакет предоставляет декоратор `@freeze_time`, который можно использовать с тестовой функцией, чтобы остановить время в конкретной точке:

```
from freeze_gun import freeze_time

@freeze_time("1988-02-05 05:10:00")
def test_with_time():
    ...
```

Во время теста все вызовы функций стандартной библиотеки, которые должны возвращать текущее время, будут возвращать значение, определяемое параметром декоратора. Среди прочего это означает, что `time.time()` вернет значение эпохи и `datetime`. А метод `datetime.now()` будет возвращать объект `datetime`, соответствующий одному и тому же моменту времени, а именно `1988-02-05 05:10:00`.

Вызов `freeze_time()` также можно использовать как диспетчер контекстов. Он возвращает специальный объект `FrozenDateTimeFactory`, который позволяет точно управлять течением времени, как в следующем примере:

```
from datetime import timedelta
from freeze_gun import freeze_time

with freeze_time("1988-02-04 05:10:00") as frozen:
    frozen.move_to("1988-02-05 05:10:00")
    frozen.tick()
    frozen.tick(timedelta(hours=1))
```

Метод `move_to()` переводит текущий контекст времени в заданную точку времени (которая указана в виде отформатированной строки или объекта `datetime`), а `tick()` смещает время вперед на заданный интервал (по умолчанию — 1 секунда).

Конечно, останавливать время следует очень осторожно. Если ваше приложение активно проверяет текущее время вызовом `time.time()` и ожидает, пока пройдет определенный промежуток времени, оно легко может попасть в состояние бесконечного ожидания.

Итоги

Самый важный принцип разработки через тестирование (TDD) — всегда начинать с тестов. Только так можно гарантировать, что ваш код будет легко

тестироваться. Кроме того, такой поход естественным образом поощряет эффективные принципы проектирования, такие как принцип единственной ответственности или инверсия управления. Соблюдение этих принципов помогает писать хороший и простой в сопровождении код. А вы уже видели, как трудно надежно протестировать код, если тесты для него пишутся задним числом.

Однако забота о правильности и удобстве сопровождения кода не ограничивается тестированием и автоматизацией контроля качества. Два этих направления работы позволяют проверить, выполняются ли известные нам требования, и исправить обнаруженные ошибки. Конечно, тестовый набор можно расширить, и вы узнали, что мутационное тестирование неплохо помогает обнаружить в тестах потенциальные «белые пятна», но у этого метода есть свои ограничения.

На следующем этапе вам обычно предстоит постоянно отслеживать работоспособность приложения и обрабатывать отчеты об ошибках от пользователей. Не стоит рассматривать пользователей как бесплатную рабочую силу, которая будет за вас отлавливать многочисленные ошибки, но в долгосрочной перспективе пользователи становятся вашим лучшим источником информации. Это объясняется как их численностью, так и тем, что им в первую очередь важно получить работоспособный продукт.

Но прежде чем пользователи смогут работать с приложением, его нужно упаковать и передать для распространения. Этой теме полностью посвящена следующая глава. Мы поговорим о том, как подготовить пакет Python для распространения в PyPI, а также обсудим популярные приемы публикации веб-приложений и настольных приложений.

11

Упаковка и распространение кода Python

В этой главе пойдет речь о том, как формировать и распространять разные виды пакетов Python. Мы рассмотрим как полноценные приложения, предназначенные для конечных пользователей, так и библиотеки, которые обычно используют только разработчики.

У всякого, кто пишет программный код, на это есть определенные причины. Возможно, вы — энтузиаст, который создает приложения для собственного удовольствия и хочет поделиться ими с друзьями. А может, вы ученый или исследователь, который занимается важной задачей и желает передать свой код другим людям, чтобы упростить их жизнь. Или, может быть, вы — профессионал, который пишет код, чтобы зарабатывать на жизнь, и вы хотите предлагать свое приложение или службу платежеспособным клиентам.

Все причины для написания кода хороши, но каждой обычно присущ свой способ распространения продукта. В этой главе мы обсудим три основных сценария:

- упаковка и распространение библиотек;
- упаковка приложений и веб-служб;
- создание автономных исполняемых файлов.

Сначала мы сосредоточимся на упаковке и распространении библиотек, потому что этот сценарий может обеспечивать другие процессы упаковки и распространения. Но начнем мы с технических требований для этой главы.

Технические требования

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `twine`
- `wheel`
- `cx_Freeze`
- `py2exe`
- `pyinstaller`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python».

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2011.

Упаковка и распространение библиотек

Программная библиотека — это повторно используемый объем кода, который можно применять как компонент более крупного приложения или другой библиотеки. Библиотеки обычно предназначены для решения ограниченного набора задач в конкретной технической области, однако на размер библиотеки нет ограничений. В рамках этой главы будем считать, что фреймворки тоже являются библиотеками, ведь их тоже можно рассматривать как компоненты приложения, хотя и более масштабные и универсальные.

Библиотеки в Python распространяются в виде пакетов (или модулей) вроде тех, которые мы уже многократно использовали в книге. Бóльшую часть пакетов, которые загружались из PyPI в предыдущих главах, можно было считать библиотеками. Большинство библиотек Python с открытым кодом распространяются через PyPI.

Стоит уметь создавать пакеты, даже если вы не собираетесь открыто распространять свой код. Это умение помогает лучше понять экосистему пакетов, а также работать со сторонним кодом, доступным в PyPI (с которым вы, скорее всего, уже работали).

Поначалу разобраться в пакетах Python может быть непросто. В основном это объясняется путаницей с выбором инструментов для создания пакетов. Впрочем, как только вы создадите свой первый пакет, вы увидите, что это не так уж сложно. И в этом вам сильно поможет знание хороших современных средств создания пакетов.

Но прежде чем переходить к этим средствам, стоит поближе познакомиться с устройством пакетов Python.

Как устроен пакет Python

Минимальным распространяемым фрагментом кода на Python является модуль — отдельный исходный файл с расширением .py. Набор модулей называется пакетом. Хотя теоретически пакеты и модули Python можно распространять в виде сырого исходного кода, чтобы пользователи запускали их в интерпретаторе Python, это создаст немало проблем у пользователей, далеких от техники. Даже разработчики рассчитывают хотя бы на минимальную упаковку, которая позволит им установить приложение или библиотеку с помощью таких инструментов, как `pip` или Poetry.

У пакетов Python, которые должны распространяться через PyPI, возможны различные структуры дерева исходного кода. Есть несколько стандартных схем, и почти каждый пакет содержит несколько стандартных файлов. Трудно сказать, какая схема лучше, поэтому просто рассмотрим структуру, которая нравится авторам:

```
.
├── packagename/
│   └── __init__.py
├── tests/
│   ├── __init__.py
│   └── conftest.py
├── bin/
├── data/
├── docs/
├── README.md
├── LICENSE
├── setup.py
├── setup.cfg
├── MANIFEST.in
└── CHANGELOG.md
```

Основная структура исходного кода проекта определяется структурой подкаталогов. Каждый подкаталог играет определенную роль:

- **имя_пакета/**: каталог с исходным кодом Python для проекта. Это основное содержимое, ради которого пакет распространяется через PyPI. Желательно, чтобы имя каталога точно совпадало с именем, с которым пакет зарегистрирован в PyPI, хотя многие разработчики при регистрации используют дефисы вместо подчеркиваний. Обычно в дереве исходного кода бывает только один пакет верхнего уровня.
- **tests/**: каталог с тестовым пакетом. Содержит тестовые модули и (не обязательно) тестовые подпакеты. В ранее приведенном примере мы познако-

мились с модулем `conftest` — это специальный тестовый модуль фреймворка `pytest`, который обычно содержит фикстуры и необязательные плагины `pytest`. Этот каталог обычно не распространяется в PyPI. Кроме того, имя `tests` встречается довольно часто, так что после установки ваш пакет `tests` с большой вероятностью будет конфликтовать с другими тестовыми пакетами в каталоге site-packages. Если вы хотите распространять тесты со своим пакетом, поместите их в отдельное пространство имен, вложив в главный каталог пакета (то есть каталог имя_пакета/).

Некоторые разработчики предпочитают размещать каталог с исходным кодом пакета и каталог с тестовым пакетом в дополнительном каталоге верхнего уровня src/. Это ни на что не влияет и скорее является делом личного предпочтения.

- bin/: каталог для сценариев оболочки и утилит, которые могут пригодиться при разработке пакета. Например, здесь могут содержаться сценарии для сборки документации, специфические статические анализаторы или утилиты, которые задействованы в процессе распространения пакета. Эти сценарии не распространяются через PyPI.

Если пакет должен распространяться со сценариями оболочки, по общепринятому соглашению его помещают в каталог scripts/.

- data/: каталог для важных файлов данных, которые должны поставляться вместе с пакетом. Например, это могут быть заранее обученные модели машинного обучения, графические файлы или файлы локализации.
- docs/: каталог для документации пакета. Документация может существовать в любой форме, но многие разработчики применяют автоматизированные системы сборки документации (такие, как Sphinx или MkDocs). В таких случаях в каталоге docs/ находится исходный код документации и данные конфигурации для таких систем, но не готовые файлы документации. Этот каталог часто не распространяется через PyPI.

Sphinx — генератор документации, с помощью которого формируется официальная документация Python. Дополнительную информацию о нем можно найти по адресу https://www.sphinx-doc.org.

Sphinx мощный, но достаточно тяжеловесный. Иногда более легкое решение может оказаться более удачным, особенно для небольших пакетов. MkDocs — популярный генератор статических сайтов, адаптированный под сборку документации проектов. За дополнительной информацией о MkDocs обращайтесь по адресу https://www.mkdocs.org.

Файлы, которые находятся вне этих каталогов, обычно содержат средства конфигурации или метаданные пакета. В предложенной структуре перечислены шесть файлов, которые образуют необходимый минимум для каждого пакета с открытым кодом:

- README.md: этот файл содержит минимальное описание и/или документацию пакета. Расширение .md обозначает язык разметки Markdown, популярный среди разработчиков. Использовать специализированный язык разметки совершенно не обязательно, и этот файл часто хранится под именем README или README.txt. Хороший тон велит включать этот файл в распространяемый пакет.

Еще один популярный язык разметки для документирования проектов Python — reStructuredText (обозначается расширением файла .rst). Этот язык разметки используется по умолчанию в Sphinx. За дополнительной информацией о reStructuredText обращайтесь по адресу https://docutils.sourceforge.io/rst.html.

- LICENSE: файл содержит текст лицензии на ПО для пользователей пакета. Обычно это простой текстовый файл, в котором не используется никакой язык разметки. Этот файл следует включать в распространяемый пакет.
- setup.py: сценарий, который формирует дистрибутив пакета и загружает его в репозиторий пакетов. Среди прочего он содержит метаданные пакета и определения расширений (если пакет их предоставляет). Он включается только в дистрибутивы с исходным кодом (они будут рассматриваться в разделе «Формы распространения пакетов»).
- setup.cfg: необязательный файл конфигурации пакета Python (в стиле INI-файлов). Может включать метаданные пакета и параметры по умолчанию для подкоманд сценария setup.py. Многие инструменты разработчика Python (тестовые фреймворки, статические анализаторы) хранят собственные данные конфигурации в разделах этого файла.
- MANIFEST.in: шаблон для манифеста пакетного файла. С его помощью можно передавать сценарию setup.py информацию о том, какие из файлов, не содержащих исходный код, нужно включить в дистрибутив.
- CHANGELOG.md: необязательный файл с перечнем всех изменений, внесенных в пакет вплоть до текущего выпуска. Считается хорошим тоном включать его в дистрибутивы пакетов. Короткие журналы изменений также можно включить в файл README, хотя если новые версии проекта выходят часто, для этой цели лучше использовать специальный файл.

Многие разработчики предпочитают вести журнал изменений в более удобной форме за пределами дерева исходного кода. Популярный пример — раздел Releases проекта на GitHub. Тем не менее в дистрибутив пакета желательно включить хотя бы минимальный журнал изменений.

Некоторые из этих файлов имеют особенный синтаксис или структуру, которые будут рассмотрены чуть позднее. Начнем с самого важного файла — сценария setup.py.

setup.py

Если у проекта есть распространяемый пакет Python, в корневом каталоге этого проекта содержится сценарий setup.py. Он предоставляет основные метаданные пакета: номер версии, описание, имена авторов, тип лицензии, необходимые зависимости и т. д. Метаданные пакета выражаются как аргументы функции `setuptools.setup()`.

В Python есть встроенный модуль `distutils` для упаковки кода, однако вместо него лучше использовать `setuptools`, потому что в этом пакете много улучшений по сравнению с `distutils`. Кроме того, начиная с Python 3.10, пакет `distutils` официально считается устаревшим, а кодовая база `setuptools` больше не зависит от модуля `distutils`. Поэтому в этой главе будет рассматриваться пакет `setuptools`.

Таким образом, минимальное содержимое файла setup.py выглядит так:

```
from setuptools import setup

setup(
    name='mypackage',
)
```

Обратите внимание: чтобы зарегистрировать пакет в репозитории пакетов, достаточно передать единственный аргумент `name`, но это еще не позволит вам создавать полноценные дистрибутивы. Для полноценного дистрибутива необходимо предоставить дополнительные метаданные, которые позволят пакету `setuptools` правильно собрать исходные файлы. Основные метаданные будут рассмотрены позднее в разделе «Важнейшие метаданные пакета».

Аргумент `name` определяет полное имя дистрибутива. Если вы решите опубликовать пакет в репозитории (таком, как PyPI), он будет зарегистрирован под этим именем. Сценарий setup.py предоставляет ряд команд, список которых можно вывести с ключом `--help-commands`:

```
$ python3 setup.py --help-commands
Standard commands:
  build             build everything needed to install
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
```

```
    sdist             create a source distribution (tarball, zip file,
etc.)
    register          register the distribution with the Python package
index
    bdist             create a built (binary) distribution
    check             perform some checks on the package
    upload            upload binary package to PyPI

Extra commands:
    bdist_wheel       create a wheel distribution
    alias             define a shortcut to invoke one or more commands
    develop           install package in 'development mode'

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

Фактический список команд длиннее и к тому же зависит от доступных расширений setuptools. Здесь мы сократили список, оставив только те команды, которые наиболее важны и актуальны для этой главы.

Стандартные команды (Standard commands) — это встроенные команды, предоставляемые `distutils`, а дополнительные команды (Extra commands) предоставляются сторонними пакетами, такими как `setuptools` или любой другой пакет, который определяет и регистрирует новые команды. В нашем примере к дополнительным командам относится `bdist_wheel`, предоставляемая пакетом `wheel`.

setup.cfg

Файл setup.cfg содержит параметры по умолчанию для команд сценария setup.py. Он особенно нолезен, если процесс сборки и распространения пакета сложен и командам сценария setup.py нужно передавать множество дополнительпых аргументов. Файл setup.cfg позволяет хранить такие нараметры по умолчапию вместе с исходным кодом каждого отдельного проекта. Он связывает ноток распространения с проектом, а также четко показывает нользователям или участникам команды, как собирается и распространяется ваш пакет.

Синтаксис файла setup.cfg совместим со встроенным модулем `configparser` и похож на синтаксис INI-файлов Microsoft Windows. Рассмотрим пример конфигурационного файла setup.cfg, который предоставляет глобальную настройку по умолчанию, а также настройки по умолчанию для комапд `sdist` и `bdist_wheel`:

```
[global]
quiet=1

[sdist]
formats=tar,zip

[bdist_wheel]
universal=1
```

С этой конфигурацией дистрибутивы с исходным кодом (раздел `sdist`) всегда будут создаваться в двух форматах (ZIP и TAR), а дистрибутивы в формате wheel (раздел `bdist_wheel`) — в универсальном формате, не зависящем от версии Python. Кроме того, из-за глобального ключа `--quiet` для каждой команды будет подавляться бóльшая часть вывода.

Глобальное подавление вывода включено только для демонстрационных целей; не стоит использовать его по умолчанию для всех команд. Вы также можете создать глобальный персональный конфигурационный файл с именем .pydistutils.cfg в своем домашнем каталоге.

MANIFEST.in

При сборке дистрибутива с исходным кодом командой `sdist` модуль `setuptools` просматривает каталог пакета и ищет в нем файлы, которые будут включены в архив. По умолчанию `setuptools` включает следующие файлы, опираясь на аргументы функции `setup()`:

- Все файлы с исходным кодом Python, определяемые аргументами `py_modules` и `packages`.
- Все исходные файлы расширений, указанные в аргументе `ext_modules`.
- Все сценарии, заданные аргументом `scripts`.
- Все файлы, определяемые аргументами `package_data` и `data_files`.
- Файлы лицензий, определяемые аргументами `license_file` и `license_files`.
- Файлы, подходящие по шаблону test/test*.py.
- Файлы с именами setup.py, pyproject.toml, setup.cfg и MANIFEST.in.
- Файлы с именами README, README.txt, README.rst и README.md.

Кроме того, если ваш пакет включен в систему управления версиями (такую, как Subversion, Mercurial или Git), можно автоматически включить все управляемые файлы с помощью расширений `setuptools`, а именно `setuptools-svn`, `setuptools-hg` и `setuptools-git`. С помощью других специализированных расширений возможна интеграция и с прочими системами управления версиями.

Независимо от используемой стратегии сбора файлов — встроенной или определяемой специальным расширением, `sdist` создает файл `MANIFEST`, в котором перечислены все файлы, и включает его в итоговый архив.

Хотя аргументы функции `setup()` позволяют указать произвольный тип файлов, которые должны включаться в дистрибутив, перечислять их друг за другом — не самый удобный вариант. Кроме того, расширения для отдельных систем управления версиями могут отобрать и такие файлы, которые не должны входить в дистрибутив. В обоих случаях с помощью шаблона MANIFEST.in можно определить дополнительный шаблон манифеста, который позволяет автоматически включать или исключать файлы по маске имен. Допустим, что вы не используете дополнительные расширения и вам нужно включить в дистрибутив файлы, которые не входят в него по умолчанию. Для этого можно определить шаблон с именем MANIFEST.in в корневом каталоге проекта (в котором находится файл setup.py). Этот шаблон сообщает команде `sdist`, какие файлы включить.

В каждой строке шаблона MANIFEST.in определяется по одному правилу включения или исключения. Ниже приведен пример шаблона MANIFEST.in, который включает файл LICENSE, дополнительную текстовую информацию из файлов .txt, а также все файлы с разметкой Markdown:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.md
```

Полный список команд файла MANIFEST.in доступен в официальной документации `distutils` по адресу https://packaging.python.org/guides/using-manifest-in/#manifest-in-commands.

Важнейшие метаданные пакета

Самый важный аргумент фупкции `setup()` — это `name`. Без него пакет `setuptools` предполагает имя `UNKNOWN`, с которым вам будет нелегко различать разные дистрибутивы.

Конечно, одного аргумента `name` недостаточно для того, чтобы обеснечить правильную и функциональпую упаковку для вашего кода. Вот остальные важные аргументы, которые может принимать функция `setup()`:

- `version`: текущая версия пакета.
- `description`: краткое описание пакета. Обычно состоит из одного предложения, которое описывает назначение пакета.

- `long_description`: полное описание пакета в формате reStructuredText (по умолчанию) или на другом поддерживаемом языке разметки.
- `long_description_content_type`: определяет тип MIME развернутого описания; сообщает репозиторию пакетов, на каком языке разметки составлено описание.
- `keywords`: список ключевых слов, которые характеризуют пакет и позволяют лучше индексировать его в репозитории пакетов.
- `author`: имя автора пакета или название организации, которая отвечает за пакет.
- `author_email`: контактный адрес электронной почты автора пакета.
- `install_requires`: список пакетов и их версий, которые являются необходимыми зависимостями вашего пакета. Например, если пакету для работы требуются другие пакеты, доступные в PyPI, здесь перечисляются их имена и требования к версиям.
- `url`: URL-адрес проекта. Часто это адрес сайта, на котором размещается исходный код и/или документация.
- `license`: название лицензии (GPL, LGPL и т. д.), на условиях которой распространяется пакет.
- `py_modules`: список модулей Python, включенных в дистрибутив. Может применяться для простых проектов, которые содержат только модули верхнего уровня, не использующие общее пространство имен пакета.
- `packages`: список всех имен пакетов в дистрибутиве; `setuptools()` предоставляет вспомогательную функцию `find_packages()`, которая может автоматически находить имена включаемых пакетов.
- `namespace_packages`: список пакетов с пространствами имен в дистрибутиве.

Перечисленные аргументы — это важнейшие метаданные, которые позволяют правильно собрать дистрибутив пакета и указать себя в качестве его создателя. Обратите внимание на лицензионную информацию и адреса (электронная почта и URL), по которым пользователи смогут получить дополнительную информацию о вашем пакете и условиях его использования или обратиться к вам за помощью.

Пакет `setuptools` предусматривает ряд других метаданных, которые здесь не перечислены. Подробное описание всех метаданных пакетов приводится в документе PEP 345, доступном по адресу https://www.python.org/dev/peps/pep-0345/.

Один из важных, но не критичных аргументов — `classifiers` — позволяет определить категорию вашего приложения с помощью стандартизированного на-

бора программных категорий, называемых классификаторами (trove classifiers). Эта возможность особенно полезна, если вы собираетесь опубликовать свое приложение в PyPI. Познакомимся с ней поближе.

Классификаторы

PyPI позволяет классифицировать приложения с помощью набора классификаторов (trove classifiers, дословно — «классификаторы сокровищ»). Все классификаторы образуют древовидную структуру. Каждая строка классификатора определяет список вложенных пространств имен с разделителем `::`. Список передается определению пакета в аргументе `classifiers` функции `setup()`.

Следующий пример списка классификаторов взят из проекта `solrq`, доступного в PyPI:

```
from setuptools import setup

setup(
    name="solrq",
    # (...)

    classifiers=[
        'Development Status :: 4 - Beta',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Programming Language :: Python :: 2',
        'Programming Language :: Python :: 2.6',
        'Programming Language :: Python :: 2.7',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.2',
        'Programming Language :: Python :: 3.3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: Implementation :: PyPy',
        'Topic :: Internet :: WWW/HTTP :: Indexing/Search',
    ],
)
```

Использовать классификаторы в определении пакета не обязательно, но они эффективно дополняют базовые метаданные, которые доступны в интерфейсе `setup()`. Среди прочего, классификаторы могут предоставлять информацию о поддерживаемых версиях Python и операционных системах, стадии разработки проекта или лицензии, на условиях которой публикуется код. Многие пользователи PyPI пользуются средствами поиска доступных пакетов по категориям, так что правильная классификация помогает пакетам достичь своей аудитории.

Классификаторы играют важную роль в экосистеме пакетов, и ими не стоит пренебрегать. Никакая центральная организация не проверяет классификацию пакетов, так что вы сами отвечаете за то, чтобы указать правильные классификаторы для своих пакетов и не создавать хаос в каталоге пакетов.

На момент написания книги в PyPI существовало 756 классификаторов, разделенных на следующие категории:

- статус разработки;
- окружение;
- фреймворк;
- целевая аудитория;
- лицензия;
- естественный язык;
- операционная система;
- язык программирования;
- тема;
- типизация.

Время от времени в этот список добавляются новые классификаторы, так что возможно, что когда вы будете читать книгу, их общее количество будет другим. Полный список классификаторов, доступных в текущий момент, находится по адресу https://pypi.org/classifiers/, и к нему можно обратиться из кода Python с помощью пакета `trove-classifier`, доступного по адресу https://github.com/pypa/trove-classifiers.

Итак, теперь вы знаете типичную структуру пакета Python. Пришло время обсудить разные виды дистрибутивов пакетов, которые поддерживаются стандартными средствами упаковки Python.

Дистрибутивы пакетов

Дистрибутив пакета объединяет исходный код пакетов Python, метаданные и дополнительные файлы в один файл архива, который можно передать другим разработчикам либо в исходной форме, либо через репозиторий пакетов.

В общем случае существуют две разновидности пакетов Python:

- дистрибутивы с исходным кодом;
- двоичные дистрибутивы.

Дистрибутивы с исходным кодом — самая простая разновидность, не зависящая от платформы. Для пакетов на чистом Python они создаются элементарно. Такие

дистрибутивы содержат только исходный код Python, поэтому они должны быть переносимыми.

Ситуация усложняется, если в пакете есть расширения, например, нанисанные на C. В этом случае дистрибутивы с исходным кодом по-прежнему будут работать, если в среде нользователя пакета есть необходимые инструменты разработки, нрежде всего компилятор и необходимые заголовочные файлы C. В таких случаях формат двоичного дистрибутива может оказаться более удобным, нотому что он способен обеспечить готовые сборки расширений для конкретных платформ.

Дистрибутивы с исходным кодом создаются с помощью команды `sdist` сценария setup.py. Поэтому они также называются `sdist`-дистрибутивами. Эту разновидность дистрибутивов проще создавать, поэтому начнем с нее.

Дистрибутивы sdist

`sdist` — простейшая из команд сценария setup.py. Она создает дерево релиза и копирует в него все, что нужно для запуска пакета. Затем дерево архивируется в один или несколько файлов (часто создается один tar-архив). По сути, этот архив — кония дерева исходного кода.

С командой `sdist` легче всего распространять пакеты, не зависящие от целевой системы. Она создает каталог dist/, в котором хранятся распространяемые архивы. Прежде чем создавать нервый дистрибутив, необходимо передать вызову `setup()` номер версии. Если этого не сделать, модуль `setuptools` использует значение но умолчанию 0.0.0.

В качестве нрактического нримера рассмотрим следующий сценарий setup.py:

```
from setuptools import setup

setup(name='acme.sql', version='0.1.1')
```

Выполним команду `sdist` для пакета acme.sql версии `0.1.1`:

```
$ python setup.py sdist
```

Результат должен выглядеть так:

```
running sdist
...
creating dist
tar -cf dist/acme.sql-0.1.1.tar acme.sql-0.1.1
gzip -f9 dist/acme.sql-0.1.1.tar
removing 'acme.sql-0.1.1' (and everything under it)
```

Если теперь вывести содержимое каталога dist/, мы увидим следующий результат:

```
$ ls dist/
acme.sql-0.1.1.tar.gz
```

Для архивов в системе Windows по умолчанию будет использоваться формат ZIP.

В имени архива указан спецификатор версии. Теперь архив можно распространять и устанавливать в любой системе, где есть Python. Если пакет в формате `sdist`-дистрибутива содержит расширения или библиотеки C, за их компиляцию отвечает целевая система. Это распространенная ситуация в системах на базе Linux или macOS, потому что они обычно предоставляют компилятор. Менее вероятно, что такой пакет сможет работать в Windows без дополнительных ухищрений.

Если пакет с расширениями предназначен для использования на разных платформах, его также всегда следует распространять в двоичном формате.

Двоичные дистрибутивы создаются другими командами сценария setup.py. Они рассматриваются в следующем разделе.

Дистрибутивы bdist и wheel

Чтобы дистрибутивы можно было распространять в двоичном формате, `setuptools` предоставляет команду `build`, которая компилирует пакет за четыре фазы:

- build_py: собирает модули на чистом Python, компилируя их в байт-код и копируя в папку сборки.
- build_clib: собирает библиотеки C (если они есть в пакете), используя компилятор Python и создавая статическую библиотеку в папке сборки.
- build_ext: собирает расширения C и помещает результат в папку сборки (например, build_clib).
- build_scripts: собирает модули, помеченные как сценарии. Также изменяет путь к интерпретатору, если он задается в первой строке (с префиксом `!#`), и корректирует режим доступа к файлу, чтобы он был исполняемым.

Каждая фаза представляет собой команду, которую также можно запустить независимо. Результатом процесса компиляции становится папка build/, которая содержит все необходимое для установки пакета. Пакет `setuptools` не поддер-

живает кросс-компиляцию: это означает, что результат выполнения команды всегда специфичен для системы, в которой она была запущена.

Если нужно создать расширения C, в процессе сборки используется системный компилятор по умолчанию и заголовочный файл Python (Python.h). Для пакетных дистрибутивов Python, вероятно, потребуется дополнительный системный пакет, зависящий от вашей ОС,— в популярных дистрибутивах Linux он часто называется `python-dev` или `python3-dev`. В нем содержатся все необходимые заголовочные файлы для сборки расширений Python.

В процессе сборки используется компилятор C, назначенный по умолчанию в вашей операционной системе. В системах на базе Linux или macOS это будет `gcc` или `clang` соответственно. В Windows может использоваться Microsoft Visual C++ (существует бесплатная версия с интерфейсом командной строки). Также доступен компилятор из проекта с открытым кодом MinGW. Выбор компилятора можно настроить в `setuptools`.

Команда `build` используется командой `bdist` для сборки двоичных дистрибутивов. `bdist` запускает `build` и все зависимые команды, после чего создает архив так же, как `sdist`.

Двоичный дистрибутив для acme.sql создается следующим образом:

```
$ python setup.py bdist
```

При запуске в macOS результат выглядит так:

```
running bdist
running bdist_dumb
running build
...
running install_scripts
tar -cf dist/acme.sql-0.1.1.macosx-10.3-fat.tar .
gzip -f9 acme.sql-0.1.1.macosx-10.3-fat.tar
removing 'build/bdist.macosx-10.3-fat/dumb' (and everything under it)
```

Если теперь вывести содержимое каталога dist/, вы увидите такой результат:

```
$ ls dist/
acme.sql-0.1.1.macosx-10.3-fat.tar.gz    acme.sql-0.1.1.tar.gz
```

Обратите внимание: имя созданного архива содержит название системы и версию, в которой он собран (macOS 10.3). В Windows та же команда создаст другой дистрибутив:

```
$ ls dist/
acme.sql-0.1.1.macosx-10.3-fat.tar.gz    acme.sql-0.1.1.tar.gz
```

```
C:\acme.sql> dir dist
25/02/2008  08:18    <DIR>          .
25/02/2008  08:18    <DIR>          ..
25/02/2008  08:24            16 055 acme.sql-0.1.1.win32.zip
               1 File(s)         16 055 bytes
               2 Dir(s)  22   2222        2 D free
```

Если пакет содержит код на C, то помимо дистрибутива с исходным кодом важно опубликовать как можно больше разных двоичных дистрибутивов. Как минимум следует предоставить двоичный дистрибутив Windows для пользователей, у которых не установлен компилятор C.

Двоичный дистрибутив содержит все ресурсы, необходимые для использования пакета в соответствующей системе. Прежде всего в нем есть папка, которая копируется в папку Python site-packages. В нем также могут содержаться файлы с кэшированным байт-кодом (файлы __pycache__/*.pyc).

Другая разновидность двоичных дистрибутивов — wheel-дистрибутивы (в буквальном переводе — «колёса»), поддержку которых предоставляет пакет `wheel`. Когда пакет `wheel` устанавливается (например, с помощью `pip`), он добавляет новую команду `bdist_wheel` в сценарий setup.py. Эта команда позволяет создавать дистрибутивы для конкретных платформ (в настоящее время только для Windows, macOS и Linux), которые становятся улучшенной альтернативой обычным дистрибутивам `bdist`. `bdist_wheel` была разработана, чтобы заменить eggs — другой формат дистрибутивов, который использовался в `setuptools`. Формат eggs сейчас считается устаревшим, поэтому в книге он не рассматривается. Список преимуществ wheel-дистрибутивов весьма обширен. Ниже перечислены преимущества, упомянутые на странице Python Wheels по адресу http://pythonwheels.com/:

- Ускоренная установка пакетов на чистом Python и платформенных расширений на C.
- Предотвращение выполнения произвольного кода для установки (обходит setup.py).
- Установка расширений C не требует компилятора в Windows, macOS или Linux.
- Более эффективное кэширование для тестирования и непрерывной интеграции.
- В процессе установки создаются файлы .pyc, которые гарантированно соответствуют используемому интерпретатору Python.
- Более единообразный процесс установки на разных платформах и машинах.

Согласно рекомендациям PyPA (Python Packaging Authority), формат wheel должен стать форматом дистрибутивов по умолчанию. Очень долгое время двоичный формат wheel не поддерживался в Linux, но, к счастью, ситуация изменилась. Двоичные дистрибутивы для Linux называются **manylinux wheel**.

PyPA — сообщество, сформированное, чтобы навести порядок в пакетной экосистеме Python. Документ Python Packaging User Guide (https://packaging.python.org), сопровождением которого занимается PyPA, является авторитетным источником информации о новейших средствах упаковки и лучших практиках.

К сожалению, процесс сборки дистрибутивов manylinux wheel не такой прямолинейный, как сборка двоичных wheel-дистрибутивов для Windows и macOS. Для этой разновидности дистрибутивов в PyPA поддерживаются специальные образы Docker, которые служат готовой средой сборки. Эти образы и информация о том, как ими пользоваться, доступны на странице GitHub проекта по адресу https://github.com/pypa/manylinux.

Регистрация и публикация пакетов

Пакеты были бы бесполезны без организованной системы их хранения, загрузки в репозиторий и скачивания. Каталог пакетов Python (Python Package Index) — главный источник пакетов с открытым кодом в сообществе Python. Любой желающий может свободно загружать в него новые пакеты; единственное требование — зарегистрироваться на сайте PyPI по адресу https://pypi.python.org/pypi.

Пакеты в PyPI закрепляются за конкретным пользователем, так что по умолчанию только пользователь, который зарегистрировал имя пакета, является его администратором и может загружать новые дистрибутивы. Это может создать проблемы в больших проектах, поэтому существует возможность пометить других пользователей как ответственных за сопровождение пакета, чтобы они тоже могли загружать дистрибутивы.

Конечно, вы пе ограничены только этим каталогом: все средства упаковки Python поддерживают работу с альтернативными репозиториями пакетов. Это особенно удобно для того, чтобы распространять закрытый код внутри организации или для целей развертывания. В этой главе мы сосредоточимся на том, как загружать открытый код в PyPI, и лишь кратко затронем тему выбора альтернативпых репозиториев.

Простейший способ загрузить пакет — это использовать следующую команду `upload` сценария setup.py:

```
$ python setup.py <команды-dist> upload
```

Здесь `<команды-dist>` — список команд, которые создают дистрибутивы для загрузки. В репозиторий будут загружены только дистрибутивы, которые созданы при этом же выполнении setup.py. Таким образом, если вы захотите загрузить дистрибутив с исходным кодом, двоичный дистрибутив и wheel-пакет одновременно, введите такую команду:

```
$ python setup.py <команды-dist> upload
```

При загрузке с помощью setup.py нельзя повторно использовать дистрибутивы, которые были собраны при предыдущих выполнениях этой команды; их придется заново собирать при каждой отправке. Это может быть неудобно в больших или сложных проектах, где создание дистрибутива иногда занимает значительное время. Заметные примеры — пакеты, использующие расширения Python/C API (см. главу 9).

Другая проблема загрузки с помощью setup.py заключается в том, что в некоторых старых версиях Python (или при неправильной настройке системы) она может использовать незашифрованный HTTP или небезопасные подключения HTTPS. Поэтому вместо `setup.py upload` рекомендуется использовать Twine. Это программа для работы с PyPI, которая на данный момент выполняет всего одну функцию — безопасную загрузку пакетов в репозиторий. Она поддерживает все форматы упаковки и всегда гарантирует, что подключение будет защищено. Кроме того, она позволяет загружать ранее созданные файлы, благодаря чему можно тестировать дистрибутивы перед выпуском.

В следующем примере использования Twine все еще требуется запускать сценарий setup.py для сборки дистрибутивов:

```
$ python setup.py sdist bdist_wheel
$ twine upload dist/*
```

Конечно, Twine не знает ваших регистрационных данных, и их нужно предоставить в специальном конфигурационном файле .pypirc, где хранится информация о репозиториях пакетов Python. Этот файл должен находиться в вашем домашнем каталоге, и он имеет следующий формат:

```
[distutils]
index-servers =
    pypi
    other
```

```
[pypi]
repository: <url-адрес репозитория>
username: <пользователь>
password: <пароль>
[other]
repository: https://example.com/pypi
username: <пользователь>
password: <пароль>
```

В разделе `distutils` должна присутствовать переменная `index-servers`, в которой перечисляются все разделы с описаниями всех доступных репозиториев и соответствующих регистрационных данных. Есть только три переменные, которые можно изменить для каждой секции репозитория:

- `repository`: URL-адрес репозитория пакетов (по умолчанию https://pypi.org/).
- `username`: имя пользователя для аутентификации в данном репозитории.
- `password`: пароль для аутентификации пользователя в данном репозитории (в текстовом виде).

Учтите, что хранить пароль репозитория в виде простого текста — не самое правильное решение с точки зрения безопасности. Лучше всегда оставлять его пустым. Twine запросит данные, когда в них возникнет необходимость.

Для безопасной работы с регистрационными данными PyPI также можно использовать пакет `keyring`. Он позволяет Twine взаимодействовать с системным менеджером паролей, например Keychain в macOS или Windows Credential Locker. Об этой возможности можно больше узнать по адресу https://twine.readthedocs.io/en/latest/index.html#keyring-support.

В идеале все упаковщики для Python должны поддерживать файл .pypirc. Сейчас этого нельзя сказать обо всех существующих инструментах, но он поддерживается самыми важными из них, такими как `pip`, `twine`, `distutils` и `setuptools`.

Опасность использования файла .pypirc с Twine связана с тем, что Twine по умолчанию настроена для публикации пакетов в PyPI. Это может создать проблемы, если вы работаете с закрытым кодом и хотите опубликовать свой пакет в частном каталоге. Если вы забудете о необходимости использовать правильный аргумент для выбора репозитория (флаг `-r`), а файл .pypirc будет настроен для работы с PyPI, ваш закрытый код может случайно оказаться доступным для всех желающих.

За дополнительной информацией о сборке и публикации пакетов с помощью Poetry обращайтесь по адресу https://python-poetry.org/docs/cli/#publish.

Poetry — инструмент, который решает многие проблемы упаковки кода Python. Poetry не требует специальных сценариев распространения (вместо сценариев setup.py используется конфигурационный файл pyproject.toml), программа полностью интерактивна и позволяет указать нужный репозиторий пакетов вместе с исходным кодом вашего проекта. Обычно распространение пакетов в Poetry сводится к выполнению двух команд:

```
$ poetry build
$ poetry publish
```

Управление версиями пакетов и зависимостями

Если ваш пакет опубликован в репозитории, то весьма вероятно, что в какой-то момент вы захотите его изменить и опубликовать новую версию. Чтобы разработчики могли решить, хотят ли они использовать новую версию пакета, выпуски пакета помечаются **спецификаторами версий**.

Спецификатор версии — это обычно строка, которая состоит из чисел, разделенных точками (например, 1.0, 3.6.5 или 4.0.0). Поэтому спецификаторы версий также часто называются **номерами версий**. В такой форме спецификаторы легко сортировать. По соглашению более высокий номер соответствует более новому релизу. Это соглашение поддерживается практически всеми средствами управления версиями пакетов и позволяет легко заменять устаревшие пакеты новыми версиями. Например, `pip` позволяет установить новую версию пакета с помощью ключа `-U`, как в следующем примере:

```
$ pip install -U pip
Collecting pip
  Using cached pip-21.0.1-py3-none-any.whl (1.5 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 20.2.4
    Uninstalling pip-20.2.4:
      Successfully uninstalled pip-20.2.4
Successfully installed pip-21.0.1
```

В этом примере мы использовали `pip` для обновления самого себя (он распространяется как пакет). Из вывода следует, что на момент выполнения команды в системе была установлена версия `pip` 20.2.4, а новейшей версией `pip` в PyPI была версия 21.0.1. `pip` сравнил два спецификатора и определил, что в PyPI доступна версия с более высоким номером. Он удалил старую версию и установил вместо нее новую в текущей среде.

Хотя версии пакетов обычно состоят только из чисел, Python разрешает использовать в спецификаторах буквы. Это позволяет, например, помечать от-

дельные версии как предвыпускные, поствыпускные или версии для разработчиков. Эти дополнительные компоненты спецификаторов версий обычно включаются в последний сегмент спецификатора сразу после числовых сегментов.

Документ PEP 440 («Идентификация версий и спецификация зависимостей») — официальный стандарт по управлению версиями пакетов, в котором среди прочего задаются следующие соглашения для этих специальных меток выпусков:

- `{a|b|rc}N`: обозначает предвыпускную версию (альфа-, бета- или кандидат на выпуск). Эти метки соответствуют версиям, находящимся на разных стадиях разработки. Альфа-версии — самые ранние стадии, а кандидаты на выпуск близки к финальным версиям. На каждой предвыпускной стадии у пакета может быть несколько версий, которые различаются значением `N`. Например, предвыпускные версии могут формировать последовательность 1.0.0a1, 1.0.0a2, 1.0.0b1 и 1.0.0rc. Версии без предвыпускных меток считаются финальными и имеют более высокий приоритет перед предвыпускными версиями с тем же номером.

По умолчанию `pip` не устанавливает предвыпускные версии и версии для разработчиков. Если вы хотите установить предвыпускную версию, добавьте в команду `pip install` ключ `--pre`.

- `postN`: обозначает поствыпускную версию. Такие версии часто используются для публикации обновлений, которые не содержат функциональных исправлений или улучшений. Например, это могут быть обновления метаданных пакета или документации (если она включена в дистрибутив). У одного номера версии может быть несколько поствыпускных модификаций, которые различаются значением `N`. Поствыпускные версии могут быть и у предвыпускных версий. Примеры спецификаторов поствыпускных версий: 1.0.0-post1, 1.0.0a1.post1 и 1.0.0.a1.post2.
- `devN`: обозначает версию для разработчиков. Некоторые специалисты по сопровождению пакетов решают публиковать пакеты в составе систем с непрерывной интеграцией, и с помощью версий для разработчиков можно различать последовательные сборки пакета. У одного номера версии может быть несколько выпусков для разработчиков, которые различаются значением `N`. Версии для разработчиков также можно совмещать с предвыпускными и поствыпускными версиями, хотя в общедоступных каталогах пакетов категорически не рекомендуется так поступать.

Полная версия документа PEP 440 доступна по адресу https://www.python.org/dev/peps/pep-0440/.

Предвыпускные и поствыпускные версии, а также версии для разработчиков усложняют управление версиями пакетов, поэтому многие специалисты по сопровождению пакетов их не используют. Тем не менее по крайней мере предвыпускные версии могут стать полезным инструментом, который позволяет разработчикам предварительно изучить и испытать будущий выпуск пакета в своей среде.

Самое важное — номер окончательной версии пакета. Существуют две популярные стратегии для выбора номера, который присваивается новой версии пакета:

- **Семантическое версионирование.** Эта стратегия предполагает, что у каждого числового компонента есть семантическое значение, по которому потребители могут оценить объем и масштаб изменений между двумя версиями.
- **Календарное версионирование.** Эта стратегия предполагает, что числовые компоненты определяются датой создания нового выпуска (реальной или предполагаемой). Это позволяет пользователям оценить время разработки, прошедшее между двумя версиями.

Чтобы упростить ситуацию, сообщество выработало стандарты для каждой из этих стратегий версионирования. Присмотримся к ним поближе.

Стандарт SemVer для семантического версионирования

Стандарт SemVer предполагает, что спецификатор версии состоит не более чем из трех числовых сегментов:

- Старший сегмент (`MAJOR`): изменение сегмента `MAJOR` указывает на модификации, не обладающие обратной совместимостью. Пользователям, которые обновляют программу с одной старшей версии на другую, следует иметь в виду, что их код может перестать корректно работать.
- Младший сегмент (`MINOR`): изменение сегмента `MINOR` указывает на обновление функциональности с сохранением обратной совместимости. Если пользователи обновляют программу с одной младшей версии на другую (при одной и той же старшей версии), они могут рассчитывать, что их код не сломается, зато можно будет воспользоваться новой расширенной функциональностью.
- Сегмент исправления (`PATCH`): изменение сегмента `PATCH` указывает на исправление ошибки. Обновляя программу с одной версии исправления на другую (при одной и той же старшей и младшей версии), пользователи вправе ожидать, что в новой версии будут исправлены некоторые ошибки, но не должны рассчитывать на улучшения или новую функциональность.

Корректная версия SemVer обязательно содержит все три сегмента в следующем порядке:

```
MAJOR.MINOR.PATCH
```

Например, версия 20.2.4 пакета означает, что пакет находится на стадии 20-го старшего обновления с 2 младшими обновлениями и 4 исправлениями. Согласно принципам версионирования SemVer, пользователям, переходящим на него с версии 20.2.0 или 20.1.0, не стоит ожидать никаких критических изменений.

Полная спецификация SemVer также описывает нумерацию предвыпускных версий и отдельных сборок и содержит рекомендации о том, как анонсировать изменения API и работать с политиками устаревания функциональности. Полный текст спецификации доступен по адресу https://semver.org.

Стандарт CalVer для календарного версионирования

CalVer — это скорее шаблон версионирования, чем полноценный стандарт (особенно по сравнению с SemVer). Он предполагает, что спецификатор версии состоит из сегментов, которые соответствуют элементам даты конкретного выпуска.

На сайте, описывающем соглашения CalVer, представлены следующие стандартные сегменты:

- `YYYY`: год в полном формате: 2006, 2016, 2106;
- `YY`: год в кратком формате: 6, 16, 106;
- `0Y`: год с дополнением нулями: 06, 16, 106;
- `MM`: месяц в кратком формате: 1, 2 ... 11, 12;
- `0M`: месяц с дополнением нулями: 01, 02 ... 11, 12;
- `WW`: номер недели (с начала года): 1, 2, 33, 52;
- `0W`: номер недели с дополнением нулями: 01, 02, 33, 52;
- `DD`: день в кратком формате: 1, 2 ... 30, 31;
- `0D`: день с дополнением нулями: 01, 02 ... 30, 31.

Все сегменты CalVer основаны на григорианском календаре.

Эта схема лучше всего подходит для проектов с четко определенным графиком выпуска или иной привязкой к временной шкале. Примеры таких проектов — `certify` (регулярно изменяемый набор списков доверенных корневых сертифи-

катов под управлением Mozilla) и `tzdata` (набор баз данных часовых поясов IANA — см. главу 3).

Не существует стандартного формата версий CalVer, и его пользователям приходится самостоятельно решать, какие сегменты задействовать. Ключевым фактором обычно становится периодичность релизов проекта. Эту схему также можно до определенной степени смешивать с семантическим версионированием. Например, в проекте `pip` используется схема версионирования из сегментов `YY.MINOR.PATCH`.

Официальный сайт CalVer не настолько подробен, как сайт спецификации SemVer, но на нем представлены некоторые интересные примеры и рекомендации по календарному версионированию. Он доступен по адресу https://calver.org.

Установка собственных пакетов

Работа с `setuptools` в основном связана со сборкой и распространением пакетов. Однако `setuptools` приходится использовать и для того, чтобы устанавливать пакеты непосредственно из исходного кода проекта. Причина проста: прежде чем загружать пакет в PyPI, желательно протестировать, правильно ли работает упаковщик. А самый простой способ тестирования — установить пакет в своей системе. Если вы загрузите неработоспособный пакет в репозиторий, то чтобы отправить его повторно, придется увеличить номер версии.

Если тестировать, правильно ли упакован код, перед тем, как распространять его, это избавит вас от избыточного нарастания номеров версий, а также, очевидно, от напрасных трат времени.

Установка пакетов из исходного кода

Устанавливать пакеты непосредственно из исходного кода с помощью `setuptools` может быть чрезвычайно полезно, если вы одновременно работаете над несколькими взаимосвязанными пакетами:

```
setup.py install
```

Команда `install` устанавливает пакет в текущей среде Python. Она пытается собрать пакет, если до этого сборка не выполнялась, а затем внедряет результат в тот каталог, где Python ищет установленные пакеты. Если у вас есть архив, содержащий дистрибутив с исходным кодом пакета, его можно распаковать во временную папку, а затем установить этой командой. Команда `install` также устанавливает из PyPI зависимости, определенные в аргументе `install_requires`.

Устанавливая пакет, вместо сценария `setup.py` можно использовать `pip`. Поскольку этот инструмент рекомендован PyPA, его следует применять, даже

если вы устанавливаете пакет в своей локальной среде для целей разработки. Чтобы установить пакет из локального исходного кода, выполните следующую команду:

```
pip install <путь_к_проекту>
```

Чтобы установить пакет из архива с дистрибутивом, используйте команду в таком виде:

```
pip install <путь_к_проекту>
```

Как ни странно, в сценарии setup.py нет команды `uninstall`. К счастью, любой пакет Python можно удалить с помощью `pip`:

```
pip install <путь_к_проекту>
```

Удаление пакетов может оказаться опасной операцией, если речь идет о пакетах системного уровня. Это еще одна причина, по которой для разработки желательно использовать виртуальные среды.

При установке пакетов с помощью сценария setup.py или команды `pip install` исходный код пакета (или содержимое дистрибутива) копируется в каталог site-packages. Но иногда требуется настроить доступ к исходному коду пакета в той или иной среде, не копируя его. Этот сценарий называется **установкой в режиме редактирования** и особенно полезен при работе над несколькими взаимосвязанными пакетами с независимыми деревьями исходного кода.

Установка пакетов в режиме редактирования

Пакеты, установленные командой `setup.py install`, копируются в каталог site-packages текущей среды Python. Это означает, что каждый раз, когда вы вносите изменения в исходный код этого пакета, его приходится переустанавливать. Часто это создает проблемы при интенсивной разработке, потому что очень легко забыть о том, что нужна переустановка.

Поэтому `setuptools` предоставляет дополнительную команду `develop`, которая позволяет устанавливать пакеты в режиме разработки. Эта команда не копирует весь пакет в каталог развертывания (`site-packages`), а создает в нем специальную ссылку на исходный код проекта. Исходный код можно редактировать, не переустанавливая пакет после каждого изменения, и он доступен в sys.path так, как если бы был установлен обычным образом.

`pip` тоже позволяет устанавливать пакеты в подобном режиме. Он называется **режимом редактирования** и включается параметром `-e` команды `install`:

```
pip install -e <путь_к_проекту>
```

Если пакет установлен в вашей среде в режиме редактирования, его можно свободно модифицировать на месте, и все изменения будут немедленно видны без переустановки.

Режим редактирования позволяет работать с несколькими взаимосвязанными пакетами без необходимости постоянно их переустанавливать. С проектами, которые состоят из таких пакетов, также удобно работать, используя пакеты пространств имен.

Пакеты пространств имен

Вот что «Дзен Python» говорит о пространствах имен:

Пространства имен — отличная штука! Будем делать их больше!

Это можно понимать как минимум в двух смыслах. Первый смысл — пространства имен в контексте языка. Все мы пользуемся следующими пространствами имен, даже не подозревая об этом:

- Глобальное пространство имен модуля.
- Локальное пространство имен вызова функции или метода.
- Пространство имен класса.

Другая разновидность пространств имен может обеспечиваться на уровне пакетов. Речь идет о пакетах пространств имен. Эта возможность упаковки кода Python, о которой часто забывают, может принести большую пользу при структурировании экосистемы пакетов в организации или в очень крупном проекте.

Пакеты пространств имен можно понимать как способ группировки взаимосвязанных пакетов, при котором каждый из них может устанавливаться независимо.

Пакеты пространств имен особенно полезны, если компоненты вашего приложения разрабатываются, упаковываются и версионируются независимо друг от друга, но при этом вы хотите обращаться к ним из одного пространства имен. Они также помогают ясно обозначить, какой организации или проекту принадлежит каждый пакет. Например, вымышленная компания Acme может использовать общее пространство имен `acme`. Эта организация может создать общий пакет пространства имен `acme`, который будет служить контейнером для других пакетов этой организации. Например, если кто-то из сотрудников Acme захочет добавить в это пространство имен библиотеку для работы с SQL, он может создать новый пакет `acme.sql`, который регистрирует себя в пространстве имен `acme`.

Важно знать, чем пакеты пространств имен отличаются от обычных пакетов и какие задачи они решают. В обычной ситуации (без пакетов пространств имен) вы бы создали пакет с именем `acme` и подпакет `sql` в следующей структуре файлов:

```
acme/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py
```

Если вы захотите добавить новый подпакет, допустим `templating`, вам придется включать его в дерево `acme` следующим образом:

```
acme/
├── acme
│   ├── __init__.py
│   ├── sql
│   │   └── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py
```

При таком подходе становится почти нереально разрабатывать acme.sql и acme.templating независимо друг от друга. В сценарии setup.py тоже придется указать все зависимости для каждого подпакета. Будет невозможно (или по крайней мере очень трудно) сделать установку некоторых компонентов `acme` необязательной. Кроме того, при большом количестве подпакетов непросто избежать конфликтов зависимостей.

С пакетами пространств имен деревья исходного кода для каждого из этих подпакетов можно хранить независимо:

```
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py
```

Эти пакеты также можно зарегистрировать независимо друг от друга в PyPI или любом другом каталоге пакетов, которым вы пользуетесь. Пользователи могут

выбрать, какие подпакеты из пространства имен `acme` они хотят установить, но при этом они никогда не устанавливают общий пакет `acme` (он даже не обязан существовать). Соответствующая команда `pip` может выглядеть так:

```
$ pip install acme.sql acme.templating
```

Учтите, что функция `setuptools.find_packages()` не обнаруживает пакеты пространств имен. Если вы хотите, чтобы ваш сценарий setup.py автоматически собирал такие пакеты (вместо того, чтобы перечислять их по отдельности), используйте функцию `setuptools.find_namespace_packages()`. Она автоматически обнаруживает пакеты пространств имен в структурах каталогов, подобных тем, которые представлены в этом примере.

И обычные пакеты, и пакеты пространств имен в основном предназначены для того, чтобы совместно использовать код между проектами, работающими в разных средах. Если установить такой пакет в конкретной среде, он немедленно становится доступным для импортирования. Однако это не единственное назначение механизма пакетов в Python. Многие проекты Python предоставляют утилиты командной строки, команды и даже приложения с графическими интерфейсами. Отличный пример — команда `pip`, распространяемая в виде пакета `pip`. Инфраструктуру упаковки Python можно использовать для того, чтобы предоставлять свои сценарии и исполняемые модули в целевой среде установки так же, как это делает пакет `pip`. Посмотрим, как это осуществить.

Сценарии пакетов и точки входа

Любой модуль Python можно выполнить как программу командой `python -m`. Это относится как к модулям стандартной библиотеки, так и к модулям из пакетов, которые установлены с помощью `pip`. Например, следующий вызов модуля `json.tool` из стандартной библиотеки позволяет отформатировать текст в формате JSON из командной оболочки:

```
$ echo '{"name": "John Doe", "age": 42}' | python -m json.tool
{
    "name": "John Doe",
    "age": 42
}
```

Это простой, но не самый удобный способ выполнить произвольный модуль из установленного пакета. Прежде всего пользователи пакета должны понимать структуру модулей внутри приложения и знать, какие модули можно запускать в оболочке. Кроме того, пользователям придется вводить команду `python -m`, что вносит в сценарии некоторую избыточность. Используя `pip`, мы бы предпочли запускать команду `pip` вместо `python -m pip`.

Можно сделать так, чтобы ваши собственные пакеты на Python вели себя так же, как `pip`, и нредоставить собственную команду оболочки, которая будет устанавливаться вместе с пакетом. Это делается двумя способами:

- с помощью аргумента `scripts` функции `setuptools.setup()`;
- с помощью аргумента `entry_points` функции `setuptools.setup()`.

Аргумент `scripts` — основной механизм, позволяющий предоставить команды оболочки для пакета. Этот аргумент уже поддерживается модулем `distutils` (модуль стандартной библиотеки, на котором основан `setuptools`), поэтому он достаточно прост. Он принимает список путей к файлам сценариев, которые должны распространяться с вашим пакетом. После установки пакета эти сценарии станут доступными в одном из каталогов `PATH`, связанных со средой Python.

Чтобы показать, как работает этот механизм, мы снова воспользуемся сценарием из главы 3, который ищет директивы импортирования в исходных файлах Python. Полный код и подробное объяснение можно найти в указанной главе. Сначала создадим файл findimports.py:

```
import os
import re
import sys

import_re = re.compile(r"^\s*import\s+\.{0,2}((\w+\.)*(\w+))\s*$")
import_from_re = re.compile(
    r"^\s*from\s+\.{0,2}((\w+\.)*(\w+))\s+import\s+(\w+|\*)+\s*$"
)

def main():
    if len(sys.argv) != 2:
        print(f"usage: {os.path.basename(__file__)} file-name")
        sys.exit(1)

    with open(sys.argv[1]) as file:
        for line in file:
            if match := import_re.match(line):
                print(match.groups()[0])

            if match := import_from_re.match(line):
                print(match.groups()[0])

if __name__ == "__main__":
    main()
```

Затем создадим следующий сценарий setup.py с базовыми метаданными и аргументом `scripts`:

```
from setuptools import setup

setup(
    name="findimports",
    version="0.0.0",
    py_modules=["findimports.py"],
    scripts=["findimports"],
)
```

Теперь пакет можно установить в режиме редактирования с помощью одной из этих команд:

```
$ pip install -e .
$ python setup.py develop
```

При желании пакет устанавливается и в обычном режиме:

```
$ pip install -e .
$ python setup.py develop
```

После установки пакета модуль `findimports` будет доступен как команда оболочки. В macOS или Linux можно воспользоваться командами `compgen` и `grep` для поиска по всем командам и убедиться, что команда действительно доступна:

```
$ pip install -e .
$ python setup.py develop
```

Как видите, сценарий findimports.py теперь доступен под именем, которое полностью совпадает с именем файла сценария. Если вы очень хотите избавиться от расширения .py в команде оболочки, возможны два решения:

- **Удалите расширение .py из имени файла модуля**. Вам придется внести соответствующие изменения в сценарий setup.py. Недостаток этого решения заключается в том, что вы больше не сможете распространять модуль `findimports` как импортируемый модуль Python (аргумент `py_modules`). Кроме того, при этом усложняется модульное тестирование этого модуля.
- **Создайте сценарий-обертку для findimports.py**: аргумент `scripts` позволяет распространять любые виды сценариев, включая сценарии оболочки. Можно создать сценарий-обертку с именем без расширения (например, scripts/findimports) и указать его в аргументе `scripts`. Файл может быть совсем простым:

```
$ pip install -e .
$ python setup.py develop
```

Проблем с расширениями файлов сценариев и сценариями-обертками в `distutils` можно избежать благодаря расширению `entry_points` из модуля `setuptools`. Это стандартизированный способ, с помощью которого можно предоставлять точки входа приложения (подобно сценариям оболочки) через конфигурацию в сценарии setup.py. Он позволяет сделать так, чтобы выбранная функция в исходном коде пакета распространялась как сценарий оболочки. Это сильно упрощает управление точками входа приложения, потому что вам не придется создавать специальные запускаемые модули.

Существуют разные виды точек входа, но самая распространенная из них — `console_scripts`, которая позволяет зарегистрировать модуль или функцию как целевой объект автоматически генерируемой сценарной команды. Вот пример консольной точки входа, которую можно предоставить для сценария `findimports`:

```
from setuptools import setup

setup(
    name="findimports",
    version="0.0.0",
    py_modules=["findimports"],
    entry_points={
        "console_scripts": ["findimports=findimports:main"]
    }
)
```

Консольные точки входа обеспечивают больше гибкости в именовании команд и позволяют избирательно указать, что именно должно выполняться при запуске команды. Слева от знака = помещается нужное имя команды (в нашем случае нросто `findimports`). Справа указывается путь импортирования модуля (снова `findimports`) и имя функции, которую надо выполнить (`main()`).

Аргумент `entry_points` позволяет лучше управлять именами команд, а также упаковывать несколько команд в один модуль Python. Но это не означает, что аргумент `scripts` становится бесполезным. Например, сценарии оболочки (такие, как Bash) пельзя упаковать с помощью `entry_points`, зато можно с помощью аргумента `scripts`.

Функциональность точек входа в пакете `setuptools` — это, по сути, обобщенный метод публикации точек подключения между пакетами. Любой пакет может запросить существующие точки входа других пакетов. Например, на этой основе можно реализовать механизм плагинов. Фреймворк модульного тестирования `pytest` служит примером пакета, который использует механизм точек входа для своей системы плагинов. За дополнительной информацией о написании плагинов для `pytest` обращайтесь по адресу https://docs.pytest.org/en/stable/writing_plugins.html.

Благодаря двоичным wheel-дистрибутивам и сценариям упаковки пакеты Python можно применять для того, чтобы распространять целые приложения. Если вы используете виртуальные среды, можно обеспечить разумную степень изоляции зависимостей между приложениями.

К сожалению, пакеты Python и виртуальные среды не решают всех проблем изоляции среды. Например, с помощью виртуальных сред Python вы не сможете оградить свои приложения от изменений в общих системных библиотеках. Кроме того, не каждая зависимость Python, которую вы используете, будет распространяться в двоичном формате wheel. Расширения Python, написанные на C, C++ и Cython, чрезвычайно популярны, а это значит, что для сложных приложений часто может требоваться компиляция на месте. Отсутствие чистой изоляции зависимостей и распространенная потребность в компиляции на месте — главные причины, из-за которых пакеты Python часто оказываются недостаточно надежными решениями для некоторых практических ситуаций. Одна из них — упаковка веб-приложений и веб-служб.

Упаковка веб-приложений и веб-служб

В распространении программного продукта традиционно участвуют две стороны. Одна сторона (распространитель) предоставляет выпуск продукта для потребления. В прошлом для этого использовались физические носители (такие, как дискеты и компакт-диски), но в наши дни это обычно делается через интернет. Другая сторона (потребитель) получает программу и устанавливает ее на своем компьютере. Эта схема не всегда справедлива для обновления ПО, потому что многие приложения обновляются автоматически. Впрочем, для установки обновлений обычно требуется согласие пользователя.

С пришествием концепции SaaS (Software as a Service, «программа как услуга») все меньше программных продуктов распространяется таким образом, чтобы их можно было установить на компьютере пользователя. Мы видим, что классические программы постепенно замещаются своими SaaS-аналогами:

- традиционные настольные приложения замещаются приложениями на базе веб-технологий;
- традиционные программные библиотеки замещаются веб-API.

ПО на базе веб-технологий не распространяется среди пользователей по принципу традиционных настольных приложений. Пользователи обычно взаимодействуют с веб-приложениями через стандартный браузер или специализированный клиент, который просто служит оболочкой для кода, размещенного на каком-нибудь сервере или кластере серверов. Его все равно требуется распро-

странять на этих серверах, но этот процесс обычно остается скрытым от конечных пользователей.

Поэтому многие разработчики в контексте веб-приложений предпочитают термин **«поставка»** (shipping): потребители сознательно регистрируются в качестве нользователей программного продукта, по практически не контролируют, как и когда им будет доставляться этот продукт. Кроме того, потенциальные обновления нросто появляются принудительно, и пользователи не могут легко отказаться от них.

Популярность приложений на базе веб-технологий непрерывно растет. Даже приложения, предназначенные в первую очередь для настольных систем, часто обеспечивают такую веб-функциональпость, как автоматизированные обновления, облачная синхронизация и совместная работа по сети. А значит, вам стоит хотя бы немного разбираться в том, как распространяются веб-приложения, даже если вы не специализируетесь в этой области.

В этом разделе рассматриваются полезные методы и инструменты для разработки и раснространения веб-приложений, а также некоторые приемы, специфические для Python.

Манифест «12-фокторное приложение»

Если у вас есть возможность размещать ПО только на своих собственных серверах, это исключает из процесса распространения один важный фактор — пользователей. Вам не нужно беспокоиться о том, смогут ли они загрузить приложение и справятся ли с процедурой установки. Не нужно учитывать их операционную систему (хотя, возможно, придется учесть браузер). Наконец, в большинстве случаев не нужно спрашивать разрешения, чтобы применить обновления. Вы можете делать все что угодно. Но стоит ли?

У программных продуктов на базе веб-технологий много преимуществ. Они находятся под вашим полным контролем. Вы можете вносить любые обновления, когда считаете нужным. Тем не менее это палка о двух концах. Пользователи веб-нриложений тоже привыкли к частым обновлениям и ожидают, что обнаруженные нроблемы будут исправляться нрактически немедленно. Кроме того, если ваше ПО окажется успешным, вам придется управляться с большим парком серверов, чтобы поддерживать растущую пользовательскую базу. А большая пользовательская база обычно является целью веб-приложений.

Поэтому чрезвычайно важно разрабатывать программы так, чтобы они могли развиваться в хорошем темне. Ваше приложение должно быть простым в настройке и не должно сковываться зависимостями (такими, как внешние службы и операционная система), чтобы его было просто сопровождать, а новые версии развертывались в элегантной и легковоспроизводимой манере. Развертывать

приложение в среде реальной эксплуатации должно быть так же просто, как запускать его локально для разработки (и наоборот).

Конечно, это нелегко сделать без некоторых практических знаний. Если у вас нет особого опыта работы с ПО в большом масштабе, вы рискуете совершить много ошибок, которые обернутся значительными затратами времени, ресурсов и денег (например, расходами на серверы). Поэтому имеет смысл следовать набору эффективных, проверенных практик.

Манифест «12-факторное приложение» содержит хорошую подборку таких практик. Это обобщенная методология разработки SaaS-приложений, не зависящая от языка. Одна из ее целей — упростить развертывание приложений, но она также уделяет внимание таким вопросам, как удобство сопровождения или простота масштабирования приложений.

Как подсказывает название, «12-факторное приложение» состоит из 12 правил:

1. **Кодовая база.** Используйте единую кодовую базу, которая отслеживается в системе управления версиями, и развертывайте много копечных экземпляров.
2. **Зависимости.** Явно объявляйте зависимости и изолируйте их.
3. **Конфигурация.** Храните конфигурацию в среде выполнения.
4. **Стороиние службы.** Рассматривайте сторонние службы (backing services) как присоединенные ресурсы.
5. **Сборка, выпуск, выполиеиие.** Четко разделяйте этапы сборки и выполнения.
6. **Процессы.** Выполняйте приложение как один или несколько процессов, не сохраняющих состояние.
7. **Привязка портов.** Экспортируйте службы через привязку (binding) портов.
8. **Параллелизм.** Масштабируйте приложение на основе модели процессов.
9. **Утилизируемость.** Максимизируйте надежность: приложение должно быстро запускаться и корректно **завершать работу.**
10. **Паритет разработки и эксплуатации.** Добивайтесь, чтобы среды разработки, промежуточного развертывания (staging) и реальной эксплуатации были максимально похожими.
11. **Журиалирование.** Рассматривайте журналы как нотоки событий.
12. **Администрирование.** Выполняйте задачи администрирования и управления как разовые процессы.

Полный текст манифеста «12-факторное приложение» доступен по адресу https://12factor.net.

Мы не будем подробно обсуждать все факторы, потому что на сайте манифеста приведены отличные объяснения и обоснования по всем пунктам. Тем не менее некоторым правилам мы все же уделим особое внимание, потому что их можно воплотить в жизнь с помощью популярных инструментов, приемов или библиотек из экосистемы Python.

Docker для 12-факторных приложений

Docker уже упоминался в главе 2 «Современные среды разработки для Python»: это легкая система виртуализации, которая может обеспечить превосходную изоляцию среды разработки. Docker просто упаковывает весь ваш код и его зависимости времени выполнения (модули, пакеты, общие библиотеки) в образы, которые могут выполняться как изолированные контейнеры в тех или иных средах.

Кроме того, контейнеры Docker не сохраняют состояния. Это означает, что если запустить два контейнера из одного образа, у них будет одинаковое исходное состояние. Каждое изменение файловой системы, внесенное в контейнере, остается внутри контейнера. Конечно, часть внутренней файловой системы контейнера можно экспортировать, смонтировав специальный том, но это всегда делается явно и не может произойти случайно. Если контейнер завершил работу (независимо от того, завершился ли главный процесс корректно или аварийно), его больше нельзя использовать и его внутреннее состояние недоступно.

На самом деле контейнеры Docker не исчезают по умолчанию после завершения. Чтобы завершенные контейнеры автоматически удалялись, добавьте флаг `--rm` в команду `docker run`. Теоретически можно взаимодействовать с контейнером после его завершения, хотя использовать эту возможность стоит только для целей анализа, а не для текущей работы.

Контейнеры в Docker определяются, выполняются и управляются так, что при этом сами по себе выполняются несколько пунктов манифеста «12-факторное приложение»:

- **Зависимости.** Чтобы создать новый образ Docker, нужно определить Dockerфайл с декларативным описанием всех подготовительных шагов. Эта стадия включает все общие библиотеки, пакеты и ваш собственный код. Более того, многоэтапные сборки Docker позволяют отделить зависимости времени сборки от зависимостей времени выполнения. Зависимости оказываются изолированными: на одном и том же хосте могут работать несколько контейнеров из разных образов Docker, и их зависимости не будут конфликтовать.

- **Сборка, выпуск, выполненне.** Как правило, образы Docker формируются за пределами среды их выполнепия, например, на выделенпом сервере сборки или даже на вашем компьютере, который используется для разработки. Образы обычно хранятся в отдельном репозитории образов, из которого демоны Docker, запущенные в целевых средах, могут загрузить новейшую версию образа. Кроме того, содержательные метки образов позволяют легко отслеживать их версии и даже маркировать их назначение для конкретной среды.
- **Процессы.** Контейнеры Docker не сохраняют состояния. Более того, с точки зрения операционной системы, в которой размещеп контейнер, он выглядит как один процесс. В нем инкапсулируются все потоки и подпроцессы, которые могут выполняться внутри контейнера, а также все ресурсы, которые он может использовать (например, память).
- **Парнтет разработки и эксплуатации.** Упаковка программных продуктов в коптейнеры позволяет сократить разрыв между средой разработки и средой реальной эксплуатации, потому что при этом изолируются многие зависимости от онерационпой системы. Кроме того, Docker Compose позволяет формировать целые приложения из пескольких коптейперов и использовать те же версии сторонних служб (базы данных, кэши, обратные прокси-серверы и т. д.), которые функционируют в среде реальной эксплуатации.

Одно из самых замечательных свойств Docker — переносимость приложений. Если в целевой системе можпо запустить демон Docker, то в ней смогут работать ваши контейперы.

Если вы управляете собственным кластером серверов (физических или виртуальных), вам придется установить в них демон Docker, а также пастроить конфигурацию и/или сценарии, которые будут постоянно поддерживать ваши контейнеры в состоянии готовпости. Но все это вам нришлось бы сделать и с любым другим ПО. Docker может упростить вам жизнь, потому что все приложения используют один формат копечного продукта, а именпо образ коптейнера, и не требуют сложного процесса установки. Для управления контейнерами, в частности, можпо использовать `systemd` — популярный мепеджер системы и служб, который есть в большипстве дистрибутивов Linux.

Создание образов Docker с помощью Docker-файлов обсуждалось в главе 2 «Современные среды разработки для Python». Эффективные приемы создания Docker-файлов описаны на странице https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

Однако не все организации готовы поддерживать собственную инфраструктуру. К счастью, многие облачные провайдеры предоставляют различные службы, которые избавляют пользователей Docker от значительной части хлопот по

обслуживанию. Если вам нужно более масштабное решение, используйте специализированную систему координации коптейнеров, такую как Kubernetes (k8s), разработанную компанией Google. Kubernetes объединяет контейнеры приложений, которые должны выполняться на одном узле кластеров, в специальные группы Pods. Kubernetes может управлять томами контейнеров, картами конфигурации, автоматизированным масштабированием служб и взаимодействием внутри кластера, а также входным трафиком.

За дополнительной информацией о Kubernetes обращайтесь по адресу https://kubernetes.io.

Kubernetes обеспечивает широкий диапазон средств координации контейнеров: от управляемых кластеров Kubernetes, где вы решаете, сколько рабочих узлов вам понадобится и как их настроить, до полностью бессерверных решений, для которых вы просто предоставляете образы Docker с готовой конфигурацией, а облачный провайдер обеспечивает масштабирование инфраструктуры за вас. Гибкое ценообразование по требованию часто означает, что вы платите только за выделенные ресурсы. Это позволяет избежать больших первоначальных затрат на инфраструктуру и масштабировать ПО по мере роста.

Конечно, Docker — не единственный механизм, который обеспечивает переносимость приложений между хостами или провайдерами служб. Но независимо от формата упаковки ваше приложение не будет переносимым, если его нельзя настраивать независимо от операционной системы и других приложений. Рассмотрим типичные приемы конфигурации приложений.

Переменные окружения

Каждому приложению нужна конфигурация, зависящая от среды. Вот примеры параметров конфигурации:

- URL-адреса подключений, имена хостов и порты сторонних служб (кэши, базы данных, прокси-серверы, веб-API).
- Регистрационные данные для этих служб.
- Другие конфиденциальные данные (ключи шифрования, клиентские сертификаты).
- Параметры рабочей среды (например, флаги включения отдельных возможностей или лимиты ресурсов).

Параметры конфигурации всегда стоит отделять от кода приложения и тем более не хранить в модулях в виде констант. Это особенно важно для значений,

которые должны оставаться конфиденциальными. Для этого есть несколько причин:

- Главная причина — безопасность. Если код содержит учетные данные и другую конфиденциальную информацию, то эти сведения будут известны каждому, кто увидит код. А если кто-то получит доступ к репозиторию кода, он узнает все секреты, включая прошлые. Это создает реальный риск безопасности.
- Отделять конфигурацию от приложений стоит еще и потому, что среды недолговечны — они приходят и уходят. Сегодня вы работаете с имеющимся набором сред, а завтра захотите создать несколько новых. А если вы создаете новую «одноразовую» среду для каждой функциональной ветви, над которой работаете? А если это нужно сделать для каждого участника команды проекта? Имеет ли смысл хранить все эти конфигурации в одном репозитории с проектом?
- Наконец, конфигурация не должна зависеть от языка и фреймворка. Со временем технологии для выполнения ваших программ будут меняться. Возможно, вы смените фреймворк или даже перейдете с Python на совершенно другой язык. Может быть, в какой-то момент вы захотите мигрировать с одной инфраструктуры на другую. Сегодня ваше приложение — простая программа, которая выполняется в виртуальной среде на одном хосте, а завтра это контейнер Docker в кластере Kubernetes или даже какая-нибудь бессерверная функция под управлением облачного провайдера. Вы не знаете заранее, как будет развиваться ваше приложение, так что желательно предоставлять ему конфигурацию как можно более универсальным способом.

Самый универсальный способ обеспечивать конфигурацию приложений — **переменные окружения**. Это простые пары «ключ — значение», которые обычно поддерживаются всеми операционными системами и языками программирования. Их можно легко изменить без каких-либо модификаций в коде или файлах. Они хранятся только в среде выполняемого процесса, которая эфемерна по своей природе, и поэтому с их помощью лучше предоставлять приложению конфиденциальные данные.

Самое большое преимущество переменных окружения — в том, что их можно полностью отделить от исходного кода приложения. Благодаря этому можно использовать один и тот же артефакт развертывания (например, образ контейнера Docker или пакет Python) в разных средах и настраивать его под каждую среду, просто определяя новые значения переменных окружения при запуске приложения. Такой подход уменьшает расхождения версий между средами и позволяет не включать секретные переменные в пакеты приложений. Кроме того, когда-нибудь вам захочется использовать код, написанный с помощью

других фреймворков и даже других языков. Переменные окружения позволяют применять один механизм конфигурации для разных технологий (в отличие от специализированных файлов конфигурации или модулей).

Пользоваться переменными окружения несложно. Если вы работаете в Linux, macOS или другой POSIX-совместимой системе, можно создать новую переменную окружения командой `export`:

```
$ export MY_VARIABLE="my-value"
```

В этих системах также можно задать переменные, которые будут находиться в области видимости только одной команды. Для этого набор переменных указывается перед командой:

```
$ export MY_VARIABLE="my-value"
```

В PowerShell для Windows значение переменной окружения можно задать с помощью специальной переменной `$env`:

```
$ export MY_VARIABLE="my-value"
```

Если же вы работаете с CMD в Windows, воспользуйтесь командой `set`:

```
$ export MY_VARIABLE="my-value"
```

Имена переменных окружения в Linux и macOS учитывают регистр символов, но в Windows он игнорируется. Поэтому переменным окружения рекомендуется присваивать имена в верхнем регистре — по аналогии с константами в коде.

Как видите, переменные окружения в разных средах задаются по-разному. Более того, системы координации контейнеров, такие как Kubernetes или облачные службы конкретных провайдеров, обычно не позволяют взаимодействовать с системной оболочкой напрямую, а значения переменных окружения задаются через снециальные файлы манифестов или API провайдера.

Однако чтение переменных во всех этих средах происходит одинаково. В Python переменные окружения доступны через переменную `environ` из встроенного модуля `os`. Переменная содержит объект, подобный словарю, с помощью которого можно обращаться к переменным окружения и изменять их.

`os.environ` можно использовать в любом месте кода, но в приложениях обычно создается один модуль, который обращается ко всем переменным окружения. При таком подходе вы получаете целостное представление обо всех параметрах конфигурации, которые использует приложение, и контролируете обработку и проверку значений.

В небольшом приложении конфигурация может выглядеть так:

```
import os

DATABASE_URI = os.environ["DATABASE_URI"]
ENCRYPTION_KEY = os.environ["ENCRYPTION_KEY"]

BIND_HOST = os.environ.get("BIND_HOST", "localhost")
BIND_PORT = int(os.environ.get("BIND_PORT", "80"))

SCHEDULE_INTERVAL = timedelta(
    seconds=int(os.environ.get("SCHEDULE_INTERVAL_SECONDS", 50))
)
```

Как видите, `os.environ` использует стандартный протокол словаря. Если указанной переменной не существует, то при обращении к ней в синтаксисе [*ключ*] возникнет исключение `KeyError`. Это стандартный способ определять обязательные переменные окружения, без которых приложение не будет работать.

Аналогичным образом метод `os.environ.get()` позволяет задать переменные окружения, которые необязательны или имеют значение по умолчанию. Значения по умолчанию — удобная возможность сократить конфигурацию, которая необходима для конкретной среды. Хорошими кандидатами для значений по умолчанию становятся переменные, которые обычно одинаковы в большинстве сред, но должны переопределяться в отдельных ситуациях (например, в среде тестирования). С точки зрения безопасности в настройках по умолчанию должны фигурировать значения для условий реальной эксплуатации, а не для разработки. Тем самым предотвращаются случайные нарушения конфигурации в самой критичной среде. Конечно, в значениях по умолчанию никогда не должны храниться конфиденциальные данные.

Наконец, некоторые значения требуют приведения к конкретным типам данных. Это связано с тем, что значения переменных окружения в объекте `os.environ` всегда являются строками. Если вам нужен другой тип, который более полезен в вашем коде, строковое значение необходимо разобрать и преобразовать. В предыдущем примере значение `BIND_PORT` преобразуется в целое число, а `SCHEDULE_INTERVAL_SECONDS` — в объект `timedelta`.

Если количество переменных окружения растет, может иметь смысл упаковать их в общий объект конфигурации, который бы автоматизировал разбор значений и придавал конфигурации более строгую структуру. Стандартная библиотека Python не предоставляет такой возможности, но в PyPI можно найти множество инструментов, которые помогают работать с переменными окружения.

К числу таких инструментов относится пакет `environ-config`. Он позволяет автоматически снабжать переменные окружения префиксами и группировать их в содержательные разделы, а также предоставляет простые средства для про-

верки и преобразования значений. Главные компоненты пакета `environ-config` — декоратор класса `environ.config()` и дескриптор `environ.var()`. С их помощью определяются классы конфигурации, которые могут читать значения непосредственно из объекта `os.environ`. Ниже приведена обновленная реализация предыдущего модуля конфигурации с использованием пакета `environ-config`:

```
from datetime import timedelta
import environ

@environ.config(prefix="")
class Config:
    @environ.config()
    class Bind:
        host = environ.var(default="localhost")
        port = environ.var(default="80", converter=int)
    bind = environ.group(Bind)
    database_uri = environ.var()
    encryption_key = environ.var()

    schedule_interval = environ.var(
        name="SCHEDULE_INTERVAL_SECONDS",
        converter=lambda value: timedelta(seconds=int(value)),
        default=50
    )
```

Чтобы создать конкретный объект конфигурации, используется функция `Config.from_environ()`:

```
>>> config = Config.from_environ()
>>> config.bind
Config.Bind(host='localhost', port=80)
>>> config.bind.host
'localhost'
>>> config.schedule_interval
datetime.timedelta(seconds=50)
```

Классы конфигурации, снабженные декоратором `environ.config()`, автоматически ищут переменные окружения, преобразуя их имена атрибутов к верхнему регистру. Например, атрибут `config.database_uri` соответствует переменной окружения `DATABASE_URI`. Если же требуется использовать конкретное имя вместо автоматически сгенерированного, можно передать именованный аргумент `name` дескриптору `environ.var()`. Соответствующим примером служит определение атрибута `schedule_interval`.

Определение класса `Config.Bind` и дескриптор `environ.group()` показывают, как создавать вложенные конфигурации. Пакет `environ-config` достаточно сообразителен, чтобы снабдить запрашиваемые имена переменных окружения нрефиксом с именем атрибута группы. Это означает, что атрибут `Config.`

`bind.host` соответствует переменной окружения `BIND_HOST`, а атрибут `Config.bind.port` — переменной `BIND_PORT`.

Однако полезнее всего то, что модуль `environ-config` позволяет удобно преобразовывать и проверять переменные окружения. Для этого служит именованный аргумент `converter`. В нем можно передавать либо конструктор типа, как в примере `Config.bind.port`, либо собственную функцию, которая принимает один позиционный строковый аргумент.

При этом часто используются одноразовые лямбда-функции, как в примере `Config.schedule_interval`. Обычно аргумента `converter` хватает для проверки того, что переменная имеет правильный тип и значение. Если этого недостаточно, можно передать дополнительный именованный аргумент `validator`. Он должен содержать вызываемый объект, который получает вывод функции `converter` и возвращает окончательный результат.

Роль переменных окружения во фреймворках приложений

Может быть не вполне ясно, какую роль играют переменные окружения во фреймворках, у которых есть специальные конфигурационные файлы или модули. Самый заметный пример такого фреймворка — Django с популярным модулем settings.py. Этот модуль используется в каждом приложении, которое содержит набор различных конфигурационных переменных времени выполнения. У него две цели:

- **Выразить структуру приложения.** Приложения Django складываются из разных составляющих: приложений, представлений, промежуточных компонентов, шаблонов, контекстных процессоров и т. д. Файл settings.py содержит манифест всех установленных приложений и используемых компонентов, а также объявления их конфигурации. Бóльшая часть конфигурации не зависит от среды, в которой выполняется приложение. Другими словами, конфигурация оказывается неотъемлемой частью приложения.
- **Определить конфигурацию времени выполнения.** Модуль settings.py удобен для того, чтобы предоставлять значения, специфические для той или иной среды, к которым можно обращаться из компонентов приложения во время выполнения. Таким образом, settings.py становится единым носителем конфигурации приложений.

Когда структура приложения выражена в форме, специфичной для конкретного фреймворка, и хранится в репозитории кода приложения, это вполне нормальное явление. Такая структура действительно является частью кода

приложения. Проблемы возникают тогда, когда в файле settings.py в явном виде хранятся значения для реальной среды, где будет развернуто приложение.

Среди некоторых разработчиков Django принято определять несколько модулей настроек, чтобы хранить конфигурацию проекта. Эти модули могут быть довольно большими, поэтому обычно создается один базовый файл settings.py с общими параметрами конфигурации и несколько модулей для отдельных сред, которые переопределяют конкретные значения (рис. 11.1).

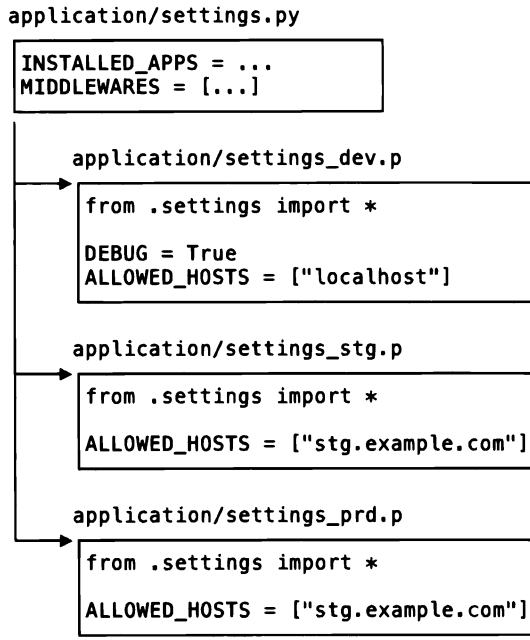

Рис. 11.1. Типичная структура модулей настроек во многих приложениях Django

Эта структура достаточно проста, и Django поддерживает ее «из коробки». При запуске приложение Django читает значение переменной окружения `DJANGO_SETTINGS_MODULE`, чтобы определить, какой модуль настроек импортировать.

И хотя схема с несколькими модулями настроек для разных сред проста и популярна, у нее много недостатков:

- **Оиосредоваииая коифигурация.** Каждый модуль пастроек должен либо храпить копию общих значений, либо импортировать значения из общего файла (обычпо второе). Если после этого вы захотите проверить фактическую конфигурацию конкретной среды, вам придется прочитать оба модуля. В редких случаях разработчики решают импортировать части конфигурации

из одной среды в другую. В таких ситуациях анализ конфигурации становится настоящим кошмаром.

- **Чтобы добавнть новую среду, нужно изменять код.** Модули настроек состоят из кода на Python, а следовательно, относятся к коду приложения. Если вам потребуется создать совершенно новую среду, придется измепять код.
- **Изменепне конфнгурации требует повторной упаковкн приложення.** Когда вы вносите в код изменения конфигурации, нужно создавать новый артефакт для развертывания. Стандартная практика методологий развертывания состоит в том, что каждая новая версия приложения проходит через несколько сред. Типичный путь продвижения выглядит так:

  Разработка → тестирование → предварительное развертывание → эксплуатация.

Когда у вас несколько модулей настроек, одно-единственное изменение для одной конфигурации среды может привести к тому, что придется повторпо развертывать приложепие в средах, не затронутых изменепием. Это приводит к лишним операционным затратам.

- **Копфнгурацин для всех сред содержатся в одном прнложении.** Это может создать риск безопаспости, если одна среда защищена слабее, чем другие. Например, если атакующий получает доступ к среде разработки, он может извлечь больше информации о возможностях для атаки в среде реальной эксплуатации. Это создает еще больше проблем, если в конфигурации хранится конфиденциальная информация.
- **Проблема секретпых значепнй.** Конфиденциальные данные не следует хранить в файловой системе и тем более в коде. Приложения Django, которые используют модули настроек для разных сред, обычно читают секретные значения из перемепных окружения или взаимодействуют со специализированными менеджерами паролей.

Мы использовали Django как пример фреймворка приложений, потому что он чрезвычайпо популярен. Однако это пе единственный фреймворк с концепцией модуля настроек и не единственный фреймворк, в котором встречается схема множественных модулей настроек на уровне среды.

Эти фреймворки часто пе могут работать без собственных модулей настроек. Это связано с тем, что такие модули определяют не только конфигурацию, относящуюся к конкретной среде, но и структуру вашего приложения. А значит, их не удастся легко заменить набором неременных окружения, ведь многие значения, которые описывают приложение, должны предоставляться в виде списков, словарей и специализированных типов данных.

Однако есть промежуточный вариант. Приложение может использовать специальный модуль настроек, но при этом соответствовать правилу «12-факторного приложения» в части хранения конфигурации в среде выполнения. Для этого нужно соблюдать ряд базовых принципов:

- **Используйте только одни модуль настроек.** Модуль настроек должен отражать структуру приложения и поведение по умолчанию (например, значения времени ожидания), которое никак не зависит от среды. Иначе говоря, если то или иное значение остается неизменным в разных средах, его можно безопасно разместить в модуле настроек.
- **Используйте переменные окружения для значений, зависящих от конкретной среды.** Если значение изменяется от одной среды к другой, ему может соответствовать переменная в модуле настроек, но читать его всегда следует из переменных окружения. Тем не менее можно действовать прагматично и использовать значения по умолчанию в ситуациях, когда они должны переопределяться только в особых обстоятельствах. Примером может служить флаг отладки, который обычно устанавливается в средах разработки, но редко в других средах.
- **В среде эксплуатации используйте значения по умолчанию.** Если у переменной конфигурации есть значение по умолчанию, о ней легко забыть при настройке конкретной среды. Если вы используете значения по умолчанию для тех или иных переменных конфигурации, всегда следите за тем, чтобы эти значения можно было безопасно использовать в среде реальной эксплуатации. Примеры значений, с которыми нужно обращаться особенно осторожно, — настройки аутентификации/авторизации или флаги для включения и отключения экспериментальных возможностей. Используя значения по умолчанию в среде реальной эксплуатации, вы защищаете ее от случайных ошибок конфигурации.
- **Никогда не сохраняйте конфиденциальную информацию в модуле настроек.** Можно обеспечить доступ к секретным данным через переменные в модуле настроек (например, читая их значения из переменных окружения), но нельзя хранить их в простом текстовом формате.
- **Не предоставляйте приложению доступ к статусу среды.** Приложение должно знать о своей среде только через конкретные переменные конфигурации. Оно никогда не должно ориентироваться на статус среды — разработка, предварительное развертывание, эксплуатация и т. д. Использовать в приложении статус среды можно разве что для того, чтобы предоставить контекст инструментам журналирования и телеметрии.

Журналирование и телеметрия (включая статусы среды) рассматриваются в главе 12 «Наблюдение за поведением и быстродействием приложений».

- **Не храните в репозитории файлы .env, связанные с отдельными средами.** Существует распространенная практика сохранять переменные среды в файлах .env. Такие переменные можно впоследствии экспортировать с помощью сценария оболочки или прочитать напрямую внутри модуля настроек. Просто избегайте искушения размещать файлы .env уровня отдельных сред в репозитории кода. Такое решение обладает всеми недостатками модулей настроек на уровне отдельных сред и только усугубляет проблему опосредованной конфигурации.

У файлов .env существует один допустимый сценарий использования — предоставление шаблона конфигурации для целей локальной разработки, чтобы программисты могли быстро настроить свою уникальную среду разработки. Такие средства локальной разработки, как Docker Compose, поддерживают файлы .env и экспортируют их значения в контейнер приложения. Тем не менее эту практику не следует распространять на другие среды. Кроме того, лучше использовать уровень сценариев (или поддержку Docker Compose) и экспортировать файлы .env как реальные переменные окружения, а не пользоваться специальными библиотеками, которые позволяют читать эти файлы прямо из файловой системы.

Этот набор принципов — прагматичный компромисс между «чистой» конфигурацией на основе окружения и классическими модулями настроек. Переменные окружения можно удобно прочитать с помощью объекта `os.environ`, пакета `environ-config` или любого другого специализированного источника.

Конечно, чтобы применять эту методологию, нужен некоторый опыт, позволяющий судить о том, какие значения будут специфическими для среды. Нет гарантий, что вам никогда не придется модифицировать код просто ради того, чтобы изменить конфигурацию конкретной среды. Необходимость в этом определенно будет возникать чаще, если код сильно зависит от значений по умолчанию. Вот почему обычно лучше избегать значений по умолчанию, если значение той или иной переменной может быть другим хотя бы в одной среде.

Упаковка приложений, которые должны выполняться на удаленных серверах, нацелена на изоляцию, удобство конфигурации и воспроизводимость. Серверы и инфраструктура, на которых выполняется ваш код, обычно находятся под вашим полным контролем, и вы можете строить специализированные архитектуры (например, системы координации контейнеров), которые поддерживают и упрощают весь процесс упаковки. Однако все кардинально меняется, если вы не являетесь пользователем или администратором целевой среды, а пользователи устанавливают или запускают ваши приложения самостоятельно. Такая ситуация типична для настольных приложений, которые устанавливаются на

персональных компьютерах пользователей. Для таких случаев обычно создаются автономные исполняемые файлы, которые работают как любые другие автономные приложения. Давайте посмотрим, как создавать такие исполняемые файлы для Python.

Создание автономных исполняемых файлов

Тему создания автономных исполняемых файлов часто упускают из виду в материалах, посвященных упаковке кода на Python. Это объясняется прежде всего тем, что в стандартной библиотеке Python нет инструментов для создания простых исполняемых файлов, которые можно запускать, не устанавливая интерпретатор Python.

У компилируемых языков есть большое преимущество перед Python: они позволяют создавать исполняемые приложения для конкретных системных архитектур. Пользователи могут запускать такие приложения, даже если не разбираются в технологиях, которые лежат в их основе. А чтобы выполнить код на Python, который распространяется в виде пакета, потребуется интерпретатор Python. Это создает изрядные неудобства для обычных пользователей, которые не являются техническими специалистами.

В операционных системах, дружественных к разработчикам, таких как macOS и многие дистрибутивы Linux, интерпретатор Python установлен по умолчанию. Таким образом, для пользователей этих систем приложение на Python можно распространять в виде пакета исходного кода. Чтобы этот код выполнялся, в главный файл сценария нужно включить специальную директиву интерпретатора, которая начинается с `#!` и для многих приложений Python выглядит так:

```
#!/usr/bin/env python
```

Такая директива находится в первой строке сценария и означает, что он предназначен для интерпретатора Python в версии по умолчанию для текущей среды. Конечно, ее можно записать в более подробной форме, указав конкретную версию Python, например `python3.9`, `python3`, `python2` и т. д. Этот способ работает в большинстве популярных POSIX-систем, но он непереносим, потому что зависит от наличия конкретных версий Python, а также от того, доступен ли исполняемый файл в каталоге по конкретному пути `/usr/bin/env`. В некоторых ОС оба условия могут нарушаться. Кроме того, директивы с `#!` вообще не работают в Windows. При этом настройка среды Python в Windows может оказаться нетривиальной задачей даже для разработчиков, и не стоит ожидать, что неквалифицированные пользователи справятся с ней самостоятельно.

Также стоит учитывать фактор удобства для пользователей в настольных средах. Пользователи обычно ожидают, что приложение можно занустить двойным щелчком на исполняемом файле или на ярлыке приложения. Не каждая настольная среда поддерживает такую возможность для приложений Python, которые распространяются в форме исходного кода.

Таким образом, было бы неплохо иметь возможность создавать двоичные дистрибутивы, которые будут работать как любые другие скомпилированпые исполпяемые файлы. К счастью, можно создать исполняемый файл, в который будет встроен как интерпретатор Python, так и ваш проект. Это позволит пользователям открыть приложение, не беспокоясь о Python или любых других зависимостях.

Рассмотрим несколько сценариев, в которых используются автономные исполняемые файлы.

Когда стоит использовать автономные исполняемые файлы

Автономные исполняемые файлы удобны там, где простота взаимодействия с пользователем важнее, чем его способность работать с кодом приложения.

Учтите, что если приложение распространяется только в виде исполпяемых файлов, читать или модифицировать его код становится более сложпой, но все-таки не невозможной задачей. Исполпяемые файлы не являются способом защиты кода, и использовать их нужно только для того, чтобы упростить взаимодействие с приложением.

Автономные исполняемые файлы должны быть предпочтительпым способом распространения приложений для пользователей, у которых нет технической квалификации. Похоже, это также единственный разумный способ раснространять приложения на Python для Windows.

Автономпые исполняемые файлы обычно хорошо подходят в таких случаях:

- Приложения, зависящие от конкретных версий Python, доступ к которым может быть затруднен в целевых ОС.
- Приложения, которые зависят от модифицированной и заранее скомпилированпой версии CPython.
- Приложения с графическим интерфейсом.
- Проекты с множеством двоичпых расширений, написанных па разных языках.
- Игры.

Создавать исполняемые файлы Python может оказаться не самым простым делом, но есть инструменты, которые могут упростить этот процесс. Рассмотрим некоторые популярные варианты.

Популярные инструменты

Python сам по себе не поддерживает создание автономных исполняемых файлов. К счастью, сообщество разработало ряд проектов, которые в той или иной мере решают эту проблему. Наибольшего внимания заслуживают следующие проекты:

- `PyInstaller`
- `cx_Freeze`
- `py2exe`
- `py2app`

Все эти инструменты используются по-разному, и их ограничения тоже несколько различаются. Прежде чем выбрать инструмент, нужно понять, на какую платформу вы ориентируетесь, потому что каждый упаковщик поддерживает ограниченный набор операционных систем.

Такое решение лучше принимать в самом начале жизненного цикла проекта. Ни один из этих инструментов не требует сложной интеграции с кодом, однако если вы начнете собирать автономные пакеты на ранней стадии, то весь процесс можно будет автоматизировать, что наверняка сэкономит время разработки в будущем. Если отложить эту задачу на потом, то может оказаться, что весь проект устроен настолько хитроумно, что ни один из существующих инструментов не будет работать «из коробки». Чтобы создать автономный исполняемый файл для такого проекта, придется приложить значительные усилия.

В следующем разделе рассматривается проект PyInstaller.

PyInstaller

На сегодняшний день PyInstaller — самая мощная программа для преобразования пакетов Python в автономные исполняемые файлы. Она обеспечивает самую широкую платформенную совместимость среди всех доступных решений, поэтому обычно стоит на первом месте среди рекомендуемых вариантов. PyInstaller поддерживает следующие платформы:

- Windows (32- и 64-разрядные);
- Linux (32- и 64-разрядные);
- macOS (32- и 64-разрядные);
- FreeBSD, Solaris и AIX.

Документация PyInstaller доступна по адресу http://www.pyinstaller.org/.

На момент написания книги последняя версия PyInstaller поддерживает все версии Python от 3.5 до 3.9. Она доступна в PyPI, и вы можете установить ее в своей рабочей среде командой `pip`. Если с этим способом установки возникнут проблемы, всегда можно загрузить установочный комплект со страницы проекта.

К сожалению, PyInstaller не поддерживает межплатформенную сборку (кросс-компиляцию), так что если вам нужно получить автономный исполняемый файл для конкретной платформы, необходимо выполнить сборку на этой платформе. В наше время это не составляет большой проблемы, потому что доступны многочисленные средства виртуализации. Если на вашем компьютере не установлена нужная ОС, всегда можно воспользоваться VirtualBox или другими средствами виртуализации, которые обеспечат требуемую систему в виде виртуальной машины.

Использовать PyInstaller для простых приложений довольно просто. Допустим, наше приложение — элементарная программа «Hello World» — содержится в сценарии с именем `myscript.py`. Мы хотим создать автономный исполняемый файл для пользователей Windows, а исходный код находится в каталоге `D:\dev\app`. Упаковка приложения производится следующей короткой командой:

```
$ pyinstaller myscript.py
```

Результат выглядит примерно так:

```
2121 INFO: PyInstaller: 3.1
2121 INFO: Python: 3.9.2
2121 INFO: Platform: Windows-7-6.1.7601-SP1
2121 INFO: wrote D:\dev\app\myscript.spec
2137 INFO: UPX is not available.
2138 INFO: Extending PYTHONPATH with paths ['D:\\dev\\app', 'D:\\dev\\
app']
2138 INFO: checking Analysis
2138 INFO: Building Analysis because out00-Analysis.toc is non existent
2138 INFO: Initializing module dependency graph...
2154 INFO: Initializing module graph hooks...
2325 INFO: running Analysis out00-Analysis.toc
(...)
25884 INFO: Updating resource type 24 name 2 language 1033
```

Стандартный вывод PyInstaller получается довольно длинным даже для простых приложений, поэтому в приведенном примере он был сокращен. В Windows

структура каталогов и файлов, которую создал PyInstaller, может выглядеть так:

```
project/
├── myscript.py
├── myscript.spec
├──build/
│   └──myscript/
│       ├── myscript.exe
│       ├── myscript.exe.manifest
│       ├── out00-Analysis.toc
│       ├── out00-COLLECT.toc
│       ├── out00-EXE.toc
│       ├── out00-PKG.pkg
│       ├── out00-PKG.toc
│       ├── out00-PYZ.pyz
│       ├── out00-PYZ.toc
│       └── warnmyscript.txt
└──dist/
    └──myscript/
        ├── bz2.pyd
        ├── Microsoft.VC90.CRT.manifest
        ├── msvcm90.dll
        ├── msvcp90.dll
        ├── msvcr90.dll
        ├── myscript.exe
        ├── myscript.exe.manifest
        ├── python39.dll
        ├── select.pyd
        ├── unicodedata.pyd
        └── _hashlib.pyd
```

Каталог **dist/myscript** содержит готовое приложение, которое теперь можно распространять среди нользователей. Обратите внимание, что распространять нужно весь каталог. Он содержит все дополпительные файлы, необходимые для запуска приложения: DLL-библиотеки, откомпилированные библиотеки расширений и т. д. Чтобы собрать более компактный дистрибутив, добавьте в команду `pyinstaller` ключ `--onefile`:

```
$ pyinstaller --onefile myscript.py
```

После этого структура файлов будет выглядеть так:

```
project/
├── myscript.py
├── myscript.spec
├──build
│   └──myscript
│       ├── myscript.exe
```

```
│       ├── myscript.exe.manifest
│       ├── out00-Analysis.toc
│       ├── out00-COLLECT.toc
│       ├── out00-EXE.toc
│       ├── out00-PKG.pkg
│       ├── out00-PKG.toc
│       ├── out00-PYZ.pyz
│       ├── out00-PYZ.toc
│       └── warnmyscript.txt
└──dist/
    └── myscript.exe
```

При сборке с ключом `--onefile` единственным файлом, который предназначен для распространения среди других пользователей, будет исполняемый файл из каталога dist (в данном случае myscript.exe). Для небольших приложений этот вариант, пожалуй, будет предпочтительным.

Один из побочных эффектов запуска `pyinstaller` — создание файла *.spec. Это автоматически генерируемый модуль Python, который описывает, как создавать исполняемые файлы из вашего исходного кода. Вот пример такого файла для кода myscript.py:

```
# -*- mode: python -*-

block_cipher = None


a = Analysis(['myscript.py'],
             pathex=['D:\\dev\\app'],
             binaries=None,
             datas=None,
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher)
pyz = PYZ(a.pure, a.zipped_data,
             cipher=block_cipher)
exe = EXE(pyz,
          a.scripts,
          a.binaries,
          a.zipfiles,
          a.datas,
          name='myscript',
          debug=False,
          strip=False,
          upx=True,
          console=True )
```

Файл .spec содержит все аргументы `pyinstaller`, указанные ранее. Это очень удобно, если вы вносите много собственных настроек в процесс сборки. После того как файл создан, его можно использовать в аргументе команды `pyinstaller` вместо сценария Python:

```
$ pyinstaller.exe myscript.spec
```

Учтите, что файл .spec — это полноценный модуль Python, так что его можно расширять, добавляя более сложные настройки в процедуру сборки. Настраивать файл .spec особенно нолезно при большом количестве целевых платформ. Кроме того, не все параметры `pyinstaller` доступны в интерфейсе командной строки, а файл .spec позволяет использовать все возможные настройки.

PyInstaller — серьезный инструмент, подходящий для подавляющего большинства программ. Тем не менее если вы намерены использовать его для распространения своих приложений, следует внимательно ознакомиться с его документацией.

В следующем разделе рассматривается `cx_Freeze`.

cx_Freeze

`cx_Freeze` — еще один инструмент для создания автономных исполняемых файлов. Он проще, чем PyInstaller, но также поддерживает три основные платформы: Windows, Linux и macOS.

Документация `cx_Freeze` доступна по адресу https://cx-freeze.readthedocs.io.

На момент нанисания книги нследняя версия `cx_Freeze` поддерживает все версии Python от 3.6 до 3.9. Пакет доступен в PyPI, что нозволяет установить его в рабочей среде с номощью pip.

По аналогии с PyInstaller, `cx_Freeze` не поддерживает кросс-платформенную сборку, так что вам придется создавать исполняемые файлы в той же онерационной системе, для которой вы собираетесь их распространять. Главный недостаток `cx_Freeze` заключается в том, что с его помощью нельзя создавать полноценные одиночные исполняемые файлы. Приложения, собранные на `cx_Freeze`, нужно распространять вместе с сонутствующими DLL-файлами и библиотеками.

Допустим, вы хотите упаковать приложение Python для Windows с номощью `cx_Freeze`. В простейшем случае для него достаточно всего одной команды:

```
$ cxfreeze myscript.py
```

Вот примерный результат выполнения этой команды:

```
copying C:\Python39\lib\site-packages\cx_Freeze\bases\Console.exe ->
D:\dev\app\dist\myscript.exe
copying C:\Windows\system32\python39.dll ->
D:\dev\app\dist\python39.dll
writing zip file D:\dev\app\dist\myscript.exe
(...)
copying C:\Python39\DLLs\bz2.pyd -> D:\dev\app\dist\bz2.pyd
copying C:\Python39\DLLs\unicodedata.pyd -> D:\dev\app\dist\
unicodedata.pyd
```

Полученная структура файлов может выглядеть так:

```
project/
├── myscript.py
└── dist/
    ├── bz2.pyd
    ├── myscript.exe
    ├── python39.dll
    └── unicodedata.pyd
```

Вместо того чтобы определять собственный формат для спецификации сборки (как это делает PyInstaller), `cx_Freeze` расширяет пакет `distutils`. Это означает, что процесс сборки автономного исполняемого файла можно настроить с помощью знакомого сценария setup.py. Благодаря этому система `cx_Freeze` очень удобна, если вы уже распространяете свой пакет средствами `setuptools` или `distutils`, потому что дополнительная интеграция требует минимальных изменений в сценарии setup.py. Приведем пример такого сценария, который использует `cx_Freeze.setup()` для создания автономных исполняемых файлов в Windows:

```
import sys
from cx_Freeze import setup, Executable

# Зависимости обнаруживаются автоматически,
# но процесс может потребовать тонкой настройки.
build_exe_options = {"packages": ["os"], "excludes": ["tkinter"]}

setup(
    name="myscript",
    version="0.0.1",
    description="My Hello World application!",
    options={
        "build_exe": build_exe_options
    },
    executables=[Executable("myscript.py")]
)
```

Имея такой файл, новый исполняемый файл можно создать с помощью новой команды `build_exe`, которая добавляется к сценарию setup.py:

```
$ python setup.py build_exe
```

Благодаря интеграции с `distutils` использование `cx_Freeze` может показаться более «питоническим», чем PyInstaller. К сожалению, этот проект может создать проблемы неопытным разработчикам по следующим причинам:

- Установка с помощью `pip` заметно усложняется в Windows.
- Официальная документация слишком короткая, и в некоторых местах она неполпая.

`cx_Freeze` — не единственный интегрированный с `distutils` инструмент для создания иснолняемых файлов Python. Два других важных примера — `py2exe` и `py2app` — описаны в следующем разделе.

py2exe и py2app

`py2exe` и `py2app` — две родственные программы, которые интегрируются с механизмом упаковки Python через `distutils` или `setuptools` и позволяют создавать автономные исполпяемые файлы. Здесь они рассматриваются вместе, потому что очень похожи как но использованию, так и по ограничениям. Главный недостаток `py2exe` и `py2app` заключается в том, что каждая из этих программ предназначена только для одной платформы:

- `py2exe` позволяет строить исполняемые файлы Windows.
- `py2app` позволяет строить приложения macOS.

Документация `py2exe` доступна по адресу https://www.py2exe.org, а документация `py2app` — по адресу https://py2app.readthedocs.io.

Так как пакеты используются очень похожим образом и требуют лишь небольших изменений сценария setup.py, они дополняют друг друга. В документации нроекта `py2app` приведен следующий пример сценария setup.py, который собирает автономные исполняемые файлы с помощью `py2exe` для Windows или `py2app` для macOS:

```
import sys
from setuptools import setup

mainscript = 'MyApplication.py'
```

```
if sys.platform == 'darwin':
    extra_options = dict(
        setup_requires=['py2app'],
        app=[mainscript],
        # Кросс-платформенные приложения обычно ожидают,
        # что для открытия файлов будет использоваться sys.argv.
        options=dict(py2app=dict(argv_emulation=True)),
    )
elif sys.platform == 'win32':
    extra_options = dict(
        setup_requires=['py2exe'],
        app=[mainscript],
    )
else:
    extra_options = dict(
        # Обычно unix-подобные платформы используют "setup.py install"
        # и устанавливают главный сценарий
        scripts=[mainscript],
    )

setup(
    name="MyApplication",
    **extra_options
)
```

С таким сценарием можно собрать исполняемый файл Windows командой `python setup.py py2exe`, а приложение macOS — командой `python setup.py py2app`.

Хотя `py2app` и `py2exe` обладают очевидными ограничениями и уступают PyInstaller или `cx_Freeze` в гибкости, все же полезно представлять себе, как они работают. Бывает, что PyInstaller или `cx_Freeze` не справляются со сборкой исполняемого файла, и в таких ситуациях всегда стоит посмотреть, не подойдут ли для вашего кода другие решения.

Безопасность кода Python в исполняемых пакетах

Важно понимать, что автономные исполняемые файлы никак не защищают код приложения. С инструментами, доступными на сегодняшний день, не существует надежного способа защитить приложение от декомпиляции: декомпилировать исходный код из исполняемых файлов — не самое простое дело, но технически это возможно. Еще важнее, что результаты такой декомпиляции (если выполнять ее подходящими инструментами) могут оказаться поразительно похожими на первоначальную версию кода.

Тем не менее есть способы затруднить декомпиляцию.

Важно заметить, что «затруднить» не значит «сделать менее вероятной». Для некоторых программистов самые трудные испытания оказываются самыми притягательными. А награда в таких испытаниях очень высока — код, который вы старались хранить в секрете.

Обычно процесс декомпиляции состоит из таких шагов:

1. Извлечь двоичное представление байт-кода проекта из автономных исполняемых файлов.
2. Отобразить двоичное представление в байт-код для конкретной версии Python.
3. Преобразовать байт-код в абстрактное синтаксическое дерево (AST).
4. Воссоздать исходный код по AST.

Предоставлять конкретные решения, которые помешали бы разработчикам провести такой «реверс-инжиниринг» автономных исполняемых файлов, было бы бессмысленно по очевидным причинам: разработчики все равно это сделают. Поэтому мы перечислим лишь несколько приемов, которые затрудняют процесс декомпиляции или снижают ценность полученных результатов:

- Удалить любые метаданные кода, которые доступны во время выполнения (doc-строки), чтобы конечный результат был менее удобочитаемым.
- Изменить значения байт-кода, которые использует интерпретатор CPython, чтобы преобразование из двоичного формата в байт-код и затем в AST требовало бóльших усилий.
- Использовать версию исходного кода CPython, модифицированную таким усложненным способом, чтобы даже декомпилированный исходный код приложения был бесполезен без декомпиляции модифицированного двоичного файла CPython.
- Обрабатывать исходный код специальными обфускаторами перед тем, как упаковывать их в исполняемый файл. Это снизит ценность кода после декомпиляции.

Все эти решения значительно усложняют процесс разработки. Некоторые из описанных приемов требуют очень глубоко разбираться в исполнительной среде Python, и каждый из них скрывает множество ловушек и недостатков. По большей части они лишь откладывают неизбежное. Как только ваш трюк будет раскрыт, все дополнительные усилия обернутся напрасными тратами времени и энергии. Это означает, что автономные исполняемые файлы Python нельзя считать эффективным решением для проектов с закрытым кодом, где утечка кода может причинить ущерб организации.

Существует только один надежный способ предотвратить утечку закрытого кода: не предоставлять его пользователям напрямую ни в какой форме. А это возможно, только если остальные аспекты безопасности организации полностью лишены уязвимостей (используется сильная многофакторная аутентификация, шифрование трафика, VPN и т. д.). Таким образом, если весь ваш бизнес можно продублировать простым копированием исходного кода приложения, вам стоит продумать другие способы его распространения. Возможно, будет лучше поставлять продукт в форме SaaS.

Итоги

В этой главе были рассмотрены разные способы упаковки библиотек и приложений Python, включая приложения для SaaS и облачных сред, а также для настольных ОС. Надеюсь, к этому моменту вы хотя бы в общих чертах представляете себе возможные средства и стратегии распространения вашего проекта. Также вы должны знать популярные приемы решения типичных проблем и понимать, как присоединить к вашему проекту полезные метаданные.

Заодно вы узнали, почему так важна экосистема упаковки и как публиковать дистрибутивы пакетов Python в каталогах пакетов. Вы убедились, что стандартные сценарии распространения (файлы `setup.py`) могут пригодиться, даже если код не публикуется в PyPI напрямую.

Настоящее веселье начинается тогда, когда ваш код попадает к пользователям. Как бы хорошо он ни был протестирован, как бы тщательно ни проектировался, обязательно окажется, что приложение не всегда ведет себя так, как ожидалось. Пользователи сообщают об ошибках. Возникают проблемы с быстродействием. Что-то неизбежно пойдет не так.

Чтобы справиться с этими проблемами, вам потребуется много информации, которая позволит воспроизводить ошибки пользователей и разбираться, что же на самом деле произошло. Мудрые разработчики всегда готовы к неожиданностям. Они знают, как активно собирать данные, которые упрощают диагностику и позволяют прогнозировать будущие сбои. Мы поговорим об этом в следующей главе.

12

Наблюдение за поведением и быстродействием приложений

С выходом каждой новой версии продукта мы чувствуем определенный мандраж. Удалось ли нам наконец-то исправить эти противные ошибки, над которыми мы бились в носледнее время? Будет ли нродукт работать или снова сломается? Будут ли пользователи довольны или продолжат жаловаться на новые ошибки и проблемы с быстродействием?

Обычно мы применяем разные нриемы контроля качества и методологии автоматизации тестирования, чтобы укрепить свою уверенность в качестве и правильности кода. Но эти приемы и методологии только усиливают наши ожидания, что с каждой новой версией все пойдет гладко. А как убедиться в том, что у пользователей все действительно идет гладко? И наоборот, как узнать, если что-то пошло не так?

В этой главе рассматривается тема наблюдаемости приложений. Наблюдаемость (observability) — свойство программной системы, которое позволяет попять и объяснить состояние приложения на основании того, что оно выводит. Если вы наблюдаете состояние системы и понимаете, как она в него перешла, вы сможете судить, нравильное ли это состояние. Рассматривая разные методы наблюдаемости, мы узнаем, как выполнять следующие операции:

- Сохранять информацию об ошибках и вести журналы.
- Снабжать код инструментами анализа для сбора снециализированных метрик.
- Настроить распределенную трассировку приложения.

Многие методы наблюдаемости применимы как к настольным приложениям, которые установлены на компьютерах пользователей, так и к распределенным системам, которые выполняются на удаленных серверах или в облачных службах. Однако в случае настольных приложений возможности наблюдения часто ограничены по соображениям конфиденциальности. Поэтому мы сосредоточимся на наблюдении за поведением и производительностью кода, который выполняется в вашей инфраструктуре.

Хорошей наблюдаемости невозможно достичь без правильно выбранных инструментов, поэтому начнем с технических требований для этой главы.

Технические требования

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `freezegun`
- `sentry-sdk`
- `prometheus-client`
- `jaeger-client`
- `Flask-OpenTracing`
- `redis_opentracing`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python». Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2012.

Сбор информации об ошибках и журналирование

Стандартный вывод — краеугольный камень наблюдаемости, потому что простейшие операции, которые умеют выполнять любые приложения, — читать данные из стандартного ввода и направлять данные в стандартный вывод. Собственно, поэтому каждый программист обычно начинает с вывода сообщения «Hello world!».

Несмотря на важность стандартного ввода и вывода, пользователи современных программ редко знают об их существовании. Настольные приложения обычно не запускаются из терминала, а пользователи взаимодействуют с ними через графический интерфейс.

Веб-приложения, как правило, выполняются на удаленных серверах, а пользователи работают с ними в браузере или в специальных клиентских программах. В обоих случаях стандартный ввод и вывод остаются скрытыми от пользователей.

Но хотя они не видят стандартного вывода, это не значит, что его не существует. Он часто используется, чтобы регистрировать подробную информацию о внутреннем состоянии приложения, предупреждениях и ошибках, произошедших в ходе выполнения программы. Более того, стандартный вывод можно легко направить в файловую систему, чтобы сохранить информацию для последующей обработки. Простота и универсальность стандартного вывода делают его одним из самых гибких средств наблюдаемости. Он также обеспечивает самый элементарный способ сохранять и анализировать информацию об ошибках.

Хотя выводом приложения можно управлять с помощью обычных вызовов функции `print()`, для полноценного журналирования требуется наладить целостную структуру и наглядное форматирование. В Python есть встроенный модуль `logging`, который предоставляет несложную, но мощную систему журналирования. Однако прежде чем более подробно изучать передовые приемы сбора информации об ошибках и ведения журнала, стоит ознакомиться с основами журналирования в Python.

Основы журналирования в Python

Модуль `logging` используется относительно прямолинейно. Прежде всего нужно создать именованный экземпляр регистратора (диспетчера журнала) `logger` и настроить систему журналирования:

```
import logging

logger = logging.getLogger("my_logger")
logging.basicConfig()
```

Стандартная идиома при определении регистратора заключается в том, чтобы присвоить ему имя модуля:

```
logger = logging.getLogger(__name__)
```

Этот прием помогает управлять конфигурацией регистратора в более иерархическом духе. За дополнительной информацией о настройке регистраторов обращайтесь к разделу «Конфигурация журналирования».

У каждого регистратора есть имя. Если аргумент `name` не указан, `logging.getLogger()` возвращает специальный регистратор `"root"`, который служит

основой для других регистраторов. `logging.basicConfig()` позволяет указывать дополнительные параметры: уровень журналирования, форматировщики, обработчики журнала и т. д.

Теперь можно вносить в журнал сообщения с определенным уровнем журналирования с помощью метода `logger.log()`:

```
logger.log(logging.CRITICAL, "Это критическое сообщение")
```

Также можно вызывать вспомогательные методы, каждый из которых связан с конкретным уровнем журналирования:

```
logger.error("Это сообщение об ошибке")
logger.warning("Это предупреждение")
```

Уровень журналирования — положительное целое число, бóльшие значения которого соответствуют более важным сообщениям. Если `logger` настроен с конкретным уровнем, он будет игнорировать сообщения с более низкими уровнями. В системе журналирования Python есть следующие предопределенные уровни (в скобках указаны соответствующие числовые значения):

- `CRITICAL` и `FATAL (50)`: сообщения об ошибках, из-за которых программа вряд ли сможет продолжить свою работу. Например, это может происходить, когда программа исчерпала ресурсы (скажем, дисковое пространство) или не может подключиться к критической сторонней службе (например, базе данных).
- `ERROR (40)`: сообщения о серьезных ошибках, из-за которых программа не может выполнить конкретную задачу или функцию: например, программе не удается разобрать данные, введенные пользователем, или истекло время ожидания сетевого запроса.
- `WARNING` или `WARN (30)`: сообщения об аномальных ситуациях, которые программе удалось преодолеть, или о ситуациях, которые могут привести к более серьезным проблемам в ближайшем будущем. Типичные примеры: программа исправила некорректный пользовательский ввод с помощью резервного значения или обнаружила низкий уровень свободного дискового пространства.
- `INFO (20)`: сообщения, которые подтверждают, что программа работает как задумано. Например, этот уровень может использоваться для подробных сведений об успешных операциях, выполненных в ходе работы программы.
- `DEBUG (10)`: очень подробные отладочные сообщения. Этот уровень позволяет отслеживать поведение приложения во время сеансов отладки.
- `NOTSET (0)`: псевдоуровень, который охватывает все возможные уровни журналирования.

Как видите, стандартные уровни журналирования определяются с интервалом 10. Это позволяет определять собственные уровни между существующими, если вам потребуется бóльшая детализация.

Для каждого предопределенного уровня журналирования, кроме `NOTSET`, в экземпляре `logger` есть специальный вспомогательный метод:

- `critical()` для уровней `CRITICAL` и `FATAL`;
- `error()` или `exception()` для уровня `ERROR` (второй метод автоматически выводит трассировку текущего перехваченного исключения);
- `warning()` для уровней `WARNING` и `WARN`;
- `info()` для уровня `INFO`;
- `debug()` для уровня `DEBUG`.

Конфигурация по умолчанию журналирует сообщения не ниже уровня `logging.WARNING` и направляет их в стандартный вывод. По умолчанию выводится текстовое представление уровня ошибки, имя регистратора и сообщение, как в следующем примере:

```
ERROR:my_logger:Это сообщение об ошибке
WARNING:my_logger:Это предупреждение
CRITICAL:my_logger:Это критическое сообщение
```

Содержание и формат вывода регистраторов можно изменить с помощью компонентов двух типов: обработчиков журналов и форматировщиков.

Компоненты системы журналирования

Система журналирования Python состоит из четырех основных компонентов:

- **Регистраторы**: точки входа системы журналирования. Код приложения использует регистраторы, чтобы передавать сообщения в систему журналирования.
- **Обработчики**: получатели системы журналирования. Обработчики присоединяются к регистраторам и должны передавать информацию нужному получателю (обычно за пределами приложения). У одного регистратора может быть несколько обработчиков.
- **Фильтры**: позволяют регистраторам или обработчикам отклонять отдельные сообщения в зависимости от их содержимого. Фильтры можно присоединять как к регистраторам, так и к обработчикам.
- **Форматировщики**: преобразуют низкоуровневые сообщения журнала в нужный формат. Форматировщики присоединяются к обработчикам сообщений

и могут создавать сообщения, предназначенные для чтения как человеком, так и машиной.

Фактически сообщения журнала Python перемещаются в одном направлении — от приложения к получателю, через регистраторы и обработчики (рис. 12.1). И регистраторы, и фильтры могут останавливать передачу сообщения либо с помощью фильтров, либо установкой флага `enabled=False`.

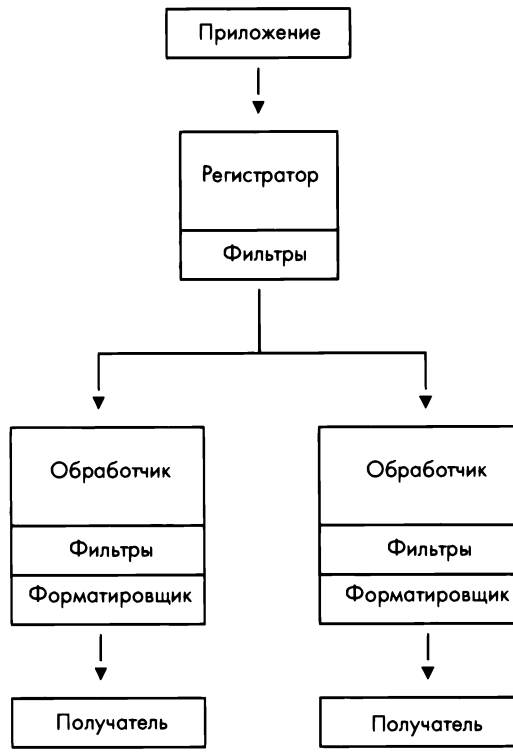

Рис. 12.1. Топология системы журналирования Python

Базовая конфигурация системы журналирования, которая порождается пустым вызовом `logging.basicConfig()`, создает следующую иерархию компонентов:

- **Корневой регистратор с уровнем журналирования logging.WARNING.** Все сообщения с уровнем ниже `logging.WARNING` по умолчанию игнорируются.
- **Один консольный обработчик, присоединенный к корневому регистратору.** Передает сообщения журнала в стандартный поток ошибок.

- **Простой форматировщик со стилем** `"%(levelname)s:%(name)s:%(message)s"`. Каждое сообщение, которое передается в заданный вывод, содержит имя уровня журналирования, имя регистратора и простой текст сообщения; разделителем служит двоеточие.

Этот набор настроек по умолчанию — хорошая отправная точка. Простой формат легко разобрать, а стандартный консольный обработчик будет работать с любым консольным приложением. Однако некоторые приложения требуют более структурированного представления сообщений и/или обработки нестандартных получателей сообщений. Для такой настройки можно переопределить обработчики и форматировщики.

В стандартной библиотеке журналирования есть три встроенных обработчика:

- `NullHandler`: не делает ничего. Может использоваться по аналогии с устройством `/dev/null` в системах POSIX, чтобы игнорировать все сообщения от регистратора. Часто является обработчиком по умолчанию в библиотеках, которые используют систему журналирования Python. В таких случаях пользователь библиотеки отвечает за то, чтобы переопределить обработчик `NullHandler` с использованием собственной конфигурации журналирования.
- `StreamHandler`: передает сообщения в заданный поток данных (объект, подобный файлу). Если поток не указан, `StreamHandler` по умолчанию передает сообщения в поток `sys.stderr`. После каждого сообщения объект потока данных сбрасывается, если он поддерживает метод `flush()`.
- `FileHandler`: подкласс `StreamHandler`, который передает сообщения в файл, заданный аргументом `filename`. Он берет на себя операции открытия и закрытия файла. По умолчанию для записи в файл используется режим дополнения и стандартная системная кодировка.

Это довольно скромный набор обработчиков, но модуль `logging.handlers` предлагает более десятка других продвинутых обработчиков. Среди них можно выделить следующие:

- `RotatingFileHandler`: файловый обработчик, который умеет переключать файлы журнала, когда текущий файл достигает определенного предельного размера. Это хорошее расширение базового `FileHandler`, особенно полезное для «разговорчивых» приложений, которые способны генерировать множество сообщений журнала за короткий промежуток времени. `RotatingFileHandler` можно настроить так, чтобы он хранил только заданное количество старых резервных файлов журнала, не позволяя заполнить весь диск журналами.
- `TimedRotatingFileHandler`: похож на `RotatingFileHandler`, но переключает файлы журнала через заданные промежутки времени (вместо отслеживания

размера). Может помочь предотвратить переполнение диска в приложениях, которые генерируют сообщения журнала в постоянном и предсказуемом темпе. Преимущество `TimedRotatingFileHandler` заключается в том, что старые файлы журналов можно легко просматривать по дате создания. Интервал по умолчанию для `TimedRotatingFileHandler` составляет 1 час.

- `SysLogHandler`: передает сообщения журнала серверу syslog. Syslog — популярный стандарт журналирования, и во многих дистрибутивах Linux сервер syslog по умолчанию запущен локально. Многие приложения и службы поддерживают журналирование с помощью syslog, так что обработчик `SysLogHandler` можно использовать, чтобы объединять журналы разных программ, которые работают на одном хосте. Кроме того, syslog позволяет передать ответственность за ротацию и сжатие файлов журналов единому системному средству журналирования.
- `SMTPHandler`: этот обработчик отправляет одно сообщение электронной почты по протоколу SMTP для каждого сообщения в журнале. Он обычно используется с уровнем `logging.ERROR` и передает информацию об ошибках и исключениях по заданному адресу электронной почты. Получатель может просмотреть эту информацию в своих входящих сообщениях. `SMTPHandler` обеспечивает один из простейших способов отслеживания ошибок и позволяет оповещать программистов о возникших проблемах, даже если исполнительная среда приложения для них сейчас недоступна.
- `SocketHandler`, `DatagramHandler` и `HTTPHandler`: простые сетевые обработчики, которые позволяют передавать сообщения журнала по сети. `SocketHandler` передает сообщения через сокет, `DatagramHandler` отправляет их в виде датаграмм UDP, а `HTTPHandler` — в виде запросов HTTP. С помощью этих обработчиков можно выстроить собственный распределенный механизм доставки журналов, хотя он будет хорошо работать только для небольших объемов данных, которые не требуют гарантий успешной доставки. Для крупных журналов рекомендуется использовать специализированный механизм доставки — syslog или другую современную распределенную систему журналирования.

Мы рассмотрим пример современной распределенной системы журналирования в разделе «Распределенные системы журналирования».

Чтобы посмотреть, как функционирует один из этих специализированных обработчиков журналов, предположим, что мы разрабатываем небольшое приложение для настольного компьютера. Будем считать, что программа выполняется с графическим интерфейсом и пользователь обычно не запускает ее из оболочки. Мы будем регистрировать информацию о предупреждениях и ошибках, обнаруженных во время выполнения программы. Если возникла проблема, пользователь сможет найти файл журнала и отправить нам ссылку на него.

Мы не знаем, сколько сообщений будет производить приложение. Чтобы предотвратить перенолнение диска, мы будем использовать ротационный обработчик файлов. Пускай ротация выполняется ежедневно, а история хранится за последние 30 дней. Конфигурация нашего обработчика журналирования может быть очень простой:

```
import logging.handlers
from datetime import timedelta, datetime

root_logger = logging.getLogger()
root_logger.setLevel(logging.INFO)
root_logger.addHandler(
    logging.handlers.TimedRotatingFileHandler(
        filename="application.log",
        when="D",
        backupCount=30,
    )
)
```

Вызов `logging.getLogger()` (без конкретного имени регистратора) позволяет нолучить специальный корневой регистратор. Его задача — предоставить конфигурацию по умолчанию и обработчики для других регистраторов, к которым не присоединены конкретные обработчики. Если у регистратора нет собственных обработчиков, то его сообщения будут автоматически раснространяться к родительскому диснетчеру журналирования.

Иерархия регистраторов и распространение сообщений журнала рассматриваются в разделе «Конфигурация журналирования».

Имея достун к корневому регистратору, можно нредоставить конфигурацию по умолчанию. Вызов `root_logger.setLevel(logging.INFO)` гарантирует, что регнстратор будет передавать только сообщения с уровнем журналирования, большим или равным `logging.INFO`. При такой настройке в журнале окажется большой объем информации. Если вы не используете нестандартные уровни сообщений, то более нодробным будет только режим `logging.DEBUG`.

Мы не настраивали систему журналирования с помощью функции `logging.basicConfig()`, так что у корневого регистратора нет обработчика по умолчанию. Мы добавили экземпляр `TimedRotatingFileHandler()` к корневому регистратору с номощью метода `addHandler()`. Аргумент `when="D"` задает ежедневную стратегию переключения, а `backupCount` определяет количество резервных файлов журналов, которые должны храниться на диске.

Вероятно, вам не захочется ждать целый месяц только для того, чтобы увидеть, как нереключаемые файлы журналов накапливаются на диске. Для ускорения

процесса мы применим ловкий трюк: с помощью пакета `freezegun` создадим у Python впечатление, что течение времени ускорилось.

Пакет `freezegun` использовался в главе 10 «Автоматизация тестирования и контроля качества» для тестирования кода, зависящего от времени.

Следующий код моделирует приложение, которое выдает одно сообщение в час, однако этот процесс ускорен в 36 000 раз:

```
from datetime import timedelta, datetime
import time
import logging
import freezegun

logger = logging.getLogger()

def main():
    with freezegun.freeze_time() as frozen:
        while True:
            frozen.tick(timedelta(hours=1))
            time.sleep(0.1)
            logger.info(f"Что-то произошло в момент времени {datetime.
now()}")

if __name__ == "__main__":
    main()
```

Запустив приложение в оболочке, вы не увидите никакого результата. Но если вы просмотрите список всех файлов в текущем каталоге, то обнаружите, что через несколько секунд начинают появляться новые файлы журнала:

```
$ ls -al
total 264
drwxr-xr-x  35 swistakm  staff  1120  8 kwi 00:22 .
drwxr-xr-x   4 swistakm  staff   128  7 kwi 23:00 ..
-rw-r--r--   1 swistakm  staff   583  8 kwi 00:22 application.log
-rw-r--r--   1 swistakm  staff  2491  8 kwi 00:21 application.log.2021-
04-07
-rw-r--r--   1 swistakm  staff  1272  7 kwi 23:59 application.log.2021-
04-08
```

Файл application.log — текущий журнал приложения, а файлы с датой в конце имени — резервные копии. Если дать программе еще немного поработать, вы убедитесь, что количество резервных файлов никогда не превышает 30. Через какое-то время `TimedRotatingFileHandler` начинает заменять старые резервные копии новыми.

В этом примере мы настроили журналирование без форматировщиков. В таких случаях обработчик передает сообщения в исходном виде, без какой-либо дополнительной информации. Взгляните на фрагмент последнего файла журнала:

```
Что-то произошло в момент времени 2021-04-08 17:31:54.085117
Что-то произошло в момент времени 2021-04-07 23:32:04.347385
Что-то произошло в момент времени 2021-04-08 00:32:04.347385
Что-то произошло в момент времени 2021-04-08 01:32:04.347385
```

В этих данных не хватает важного контекста. Мы видим дату сообщения только потому, что сами включили ее в сообщения, но не получаем информации о регистраторе, от которого поступают сообщения, и о критичности сообщения. Чтобы настроить вывод обработчика журнала, нужно присоединить к нему специализированный форматировщик сообщений.

Для этого используется метод `setFormatter()` объекта `handler`. Форматировщик должен быть экземпляром `logging.Formatter()`, который получает четыре аргумента инициализации:

- `fmt`: строковый шаблон форматирования выходного сообщения. В шаблон можно включать ссылки на любые атрибуты класса `logging.LogRecord`. По умолчанию используется значение `None` — простые текстовые сообщения без какого-либо форматирования.
- `datefmt`: шаблон форматирования даты для временной метки сообщения. Поддерживаются те же директивы форматирования, как у функции `time.strftime()`. По умолчанию используется значение `None`, которое соответствует формату, сходному с ISO8601.
- `style`: определяет стиль форматирования строки для аргумента `fmt`. Может принимать значения `'%'` (интерполяция строк с оператором `%`), `'{'` (форматирование `str.format()`) и `'$'` (форматирование `string.Template`). По умолчанию используется значение `'%'`.
- `validate`: указывает, нужно ли проверять аргумент `fmt` по аргументу `style`. По умолчанию используется значение `True`.

Например, можно настроить нестандартный форматировщик, который включает в сообщение время, уровень журналирования, имя регистратора и номер строки в коде. Конфигурация форматировщика будет выглядеть так:

```
root_logger = logging.getLogger()
root_logger.setLevel(logging.INFO)
formatter = logging.Formatter(
    fmt=(
        "%(asctime)s | %(levelname)s | "
        "%(name)s | %(filename)s:%(lineno)d | "
```

```
            "%(message)s"
        )
    )
    handler = logging.handlers.TimedRotatingFileHandler(
        filename="application.log",
        when="D",
        backupCount=30,
    )
    handler.setFormatter(formatter)
    root_logger.addHandler(handler)
```

За дополнительной информацией об атрибутах LogRecord, которые можно использовать в нестандартных форматировщиках, обращайтесь по адресу https://docs.python.org/3/library/logging.html#logrecord-attributes.

При такой конфигурации сообщение в файле журнала будет выглядеть следующим образом:

```
2021-04-08 02:03:50,780 | INFO | __main__ | logging_handlers.py:34 |
Что-то произошло в момент времени 2021-04-08 00:03:50.780432
```

Решение о том, стоит ли использовать специализированный регистратор, обычно зависит от многих факторов, включая целевую операционную систему, существующую в вашей организации системную инфраструктуру журналирования, предполагаемый объем журналов и методологию развертывания. Журналирование в автономных приложениях, системных службах или контейнерах Docker обычно устроено по-разному, и это необходимо учитывать. Например, полезно иметь старые файлы журналов, однако может оказаться затратно управлять наборами из тысяч файлов в больших распределенных системах, состоящих из десятков и сотен хостов. С другой стороны, не стоит ожидать, что пользователи автономных приложений будут запускать распределенную систему журналирования на своих персональных компьютерах только для того, чтобы воспользоваться вашим приложением.

То же можно сказать о форматировании сообщений. Некоторые системы сбора и обработки данных журналов используют преимущества структурированных форматов сообщений (таких, как JSON, msgpack или avro). Другие способны разбирать и извлекать семантические элементы сообщений по собственным правилам. Сообщения в формате простого текста легче читать людям, но сложнее обрабатывать в специализированных программах анализа журналов. Структурированные сообщения лучше подходят для машинной обработки, но их труднее читать невооруженным глазом.

Независимо от ваших текущих потребностей, можете быть уверены, что ваши предпочтения относительно журналирования изменятся со временем. Поэтому

конфигурация журналирования, в которой обработчики и форматировщики создаются вручную, редко оказывается удобной.

Вы уже знаете о функции `logging.basicConfig()`. Она создает приемлемую конфигурацию по умолчанию, однако также позволяет задать формат сообщения или указать обработчик по умолчанию. К сожалению, она работает только на уровне корневого регистратора и не позволяет выбирать другие регистраторы. В следующем разделе вы узнаете об альтернативных методах конфигурации, которые позволяют определять сложные правила журналирования для приложений любых размеров.

Конфигурация журналирования

Использовать корневой регистратор удобно, чтобы задать конфигурацию верхнего уровня системы журналирования. К сожалению, быстро выясняется, что некоторые библиотеки часто используют модуль Python `logging`, чтобы передавать информацию о важных событиях, которые происходят внутри них. В какой-то момент вам понадобится перенастроить журналирование и этих библиотек.

Обычно библиотеки используют собственные имена регистраторов. В распространенной схеме именования применяется специальный атрибут `__name__`, который содержит полное имя модуля:

```
import logging

logger = logging.getLogger(__name__)
```

Например, атрибут `__name__` в подмодуле `utils` пакета `acme` содержит значение `acme.utils`. Если `acme.utils` определяет свой регистратор вызовом `logging.getLogger(__name__)`, то ему будет присвоено имя `'acme.utils'`.

Если вам известно имя регистратора, вы можете в любой момент обратиться к нему, чтобы настроить нестандартную конфигурацию. Вот типичные сценарии такой настройки:

- **Подавление вывода**. Иногда вас вообще не интересуют сообщения, которые поступают от библиотеки. В главном файле приложения можно найти регистратор и отключить его вывод с помощью атрибута `disabled`, как в следующем примере:

```
acme_logger = logging.getLogger("acme.utils")
acme_logger.disabled = True
```

- **Переопределение обработчиков**: библиотеки не должны определять собственные обработчики; это задача пользователя библиотеки. Тем не менее

не каждый программист знаком с хорошими практиками журналирования, и иногда в полезном стороннем пакете оказывается регистратор с присоединенным обработчиком: фактически он игнорирует вашу конфигурацию корневого диспетчера журналирования.

Для такого регистратора можно нереопределить обработчики, как показано в этом примере:

```
acme_logger = logging.getLogger("acme.utils")
acme_logger.handlers.clear()
```

- **Изменение детализации вывода**: может оказаться, что некоторые регистраторы передают слишком много информации. Иногда предупреждения, которые передает библиотека, недостаточно серьезны для того, чтобы включать их в главный журнал приложения. Можно найти регистратор и переопределить для него уровень критичности сообщений:

```
acme_logger = logging.getLogger("acme.utils")
acme_logger.setLevel(logging.CRITICAL)
```

Регистраторы на уровне модулей могут пригодиться не только в библиотеках. Хорошая практика состоит в том, чтобы использовать такие регистраторы в больших приложениях, которые состоят из нескольких подпакетов или подмодулей. Это позволит вам легко настроить степень детализации и управлять обработчиками разных регистраторов. А чтобы целостно управлять конфигурацией журналирования Python, чрезвычайно важно понимать иерархические отношения между регистраторами.

Когда вы создаете регистратор с именем, содержащим точки, модуль `logging` помещает его в иерархическую структуру. Например, если обратиться к регистратору с помощью вызова `logging.getLogger("acme.lib.utils")`, то модуль `logging` выполняет следующие действия:

1. Сначала он ищет регистратор с именем `"acme.lib.utils"` потоково-безопасным способом. Если такого регистратора не существует, модуль создает новый экземпляр и регистрирует его под именем `"acme.lib.utils"`.
2. Если создан новый регистратор, модуль последовательно удаляет последние сегменты из его имени и ищет полученное имя, пока не найдет существующий регистратор. Если регистратора с таким именем нет, модуль `logging` создает специальный объект-заполнитель. Для `"acme.lib.utils"` он сначала будет искать `"acme.lib"`, а затем `"acme"`. Первый регистратор, не являющийся заполнителем, станет родительским по отношению к `"acme.lib.utils"`.
3. Если у вновь созданного регистратора нет ни одного предка, который не является заполнителем, его родителем становится корневой регистратор.

Кроме того, модуль `logging` следит за тем, чтобы существующие заполнители заменялись настоящими регистраторами, когда к ним впервые явно обращаются с помощью функции `logging.getLogger()`. При этом иерархические связи обновляются ретроспективно. Благодаря этому конкретные регистраторы можно настраивать в любом порядке и независимо от их места в иерархии.

От родительско-дочерних отношений существенно зависит, какие обработчики активизируются и в каком норядке. Когда вы передаете новое сообщение через конкретный регистратор, обработчики вызываются по следующим правилам:

1. Если у регистратора есть свои обработчики, то сообщение передается каждому из них:
   - Если атрибуту регистратора `propagate` присвоено значение `True` (значение по умолчанию), то сообщение раснространяется к родительскому регистратору.
   - Если атрибуту `propagate` присвоено значение `False`, обработка останавливается.
2. Если у регистратора нет своих обработчиков, сообщение передается родительскому регистратору.

Лучше всего определять обработчики журналов только для корневого регистратора, иначе будет трудно отслеживать все правила распространения и следить за тем, чтобы каждое сообщение регистрировалось только один раз. Но иногда имеет смысл определять обработчики для регистраторов более пизкого уровня (на уровне модулей), чтобы по-особому обрабатывать ту или иную категорию ошибок. Нанример, приложение может по умолчапию нонолнять журнал с помощью стандартного консольного обработчика, но к критически важному модулю будет присоединен `SMTPHandler`. Благодаря этому все сообщения от этого модуля будут дополнительно приходить вам на электронную ночту по SMTP.

Иерархия также помогает унравлять детализацией вывода целых групп регистраторов. Например, если у пакета `acme` есть несколько регистраторовпотомков и ни к одному из них не присоединен обработчик, можно заблокировать регистратор `"acme"`, чтобы нодавить вывод всех регистраторов нижних уровней.

Сложные иерархии могут отпугивать, особепно если вам приходится тонко пастраивать регистраторы, которые находятся в нескольких модулях. Чтобы создавать отдельные регистраторы, обработчики и форматировщики на Python, иногда требуется много шаблопного кода. Поэтому в модуле `logging.config` есть две функции, которые позволяют настроить всю систему журналирования в Python на более декларативном уровне:

- `fileConfig()`: принимает путь к конфигурационному файлу в стиле INI. Синтаксис этого файла такой же, как у конфигурационных файлов, с которыми работает встроенный модуль `configparser`.
- `dictConfig()`: принимает словарь с данными конфигурации.

За дополнительной информацией о журналировании в Python обращайтесь по адресу https://docs.python.org/3/library/logging.config.html.

В обоих способах конфигурации используются похожие разделы и параметры, различается только формат. Вот пример конфигурационного файла для журналирования с ротацией файлов по времени:

```
[formatters]
keys=default

[loggers]
keys=root

[handlers]
keys=logfile

[logger_root]
handlers=logfile
level=INFO

[formatter_default]
format=%(asctime)s | %(levelname)s | %(name)s | %(filename)s:%(lineno)d
| %(message)s

[handler_logfile]
class=logging.handlers.TimedRotatingFileHandler
formatter=default
kwargs={"filename": "application.log", "when": "D", "backupCount": 30}
```

А вот как можно определить ту же самую конфигурацию с помощью функции `dictConfig()`:

```
logging.config.dictConfig({
    "version": 1,
    "formatters": {
        "default": {
            "format": (
                "%(asctime)s | %(levelname)s | "
                "%(name)s | %(filename)s:%(lineno)d | "
                "%(message)s"
            )
        },
    },
```

```
    "handlers": {
        "logfile": {
            "class": "logging.handlers.TimedRotatingFileHandler",
            "formatter": "default",
            "filename": "application.log",
            "when": "D",
            "backupCount": 30,
        }
    },
    "root": {
        "handlers": ["logfile"],
        "level": "INFO",
    }
})
```

С таким разнообразием возможностей форматирования и иерархической структурой регистраторов работать с журналами в Python может быть довольно сложно. Однако часто можно уменьшить эти сложности, если применять полезные приемы из следующего раздела.

Хорошие практики журналирования

Python предоставляет гибкую и мощную систему журналирования. При таком количестве параметров конфигурации нетрудно запутаться в лишних сложностях. Однако на практике журналирование должно быть максимально простым. В конце концов, это первостепенный инструмент для понимания того, как работает ваше приложение. Вам будет некогда заниматься своим кодом, если вы будете тратить долгие часы, нытаясь разобраться, что же не так с журналом.

Для качественного журналирования чрезвычайно важно применять хорошие практики. Вот общепринятые правила эффективного и простого журналирования в Python:

- **Используйте регистраторы на уровне модулей.** Если создавать новые регистраторы с помощью `logging.getLogger(__name__)`, вам будет проще контролировать поведение журналирования для целых деревьев модулей. Этот подход особенно полезен для библиотек, потому что позволяет разработчикам настраивать журналы библиотек из конфигурации журналирования в основном приложении.
- **Используйте одно событие на строку.** Если вы используете текстовый вывод (потоки данных `stdout/stderr` или файлы), желательно, чтобы каждое сообщение журнала умещалось в одной строке. Это позволяет избежать многих проблем буферизации и смешения сообщений, если они поступают от нескольких потоков или процессов, которые направляют данные журнала в один вывод. Если вам очень нужно включить в сообщение текст из нескольких

строк (например, трассировку стека ошибки), это можно сделать с помощью таких структурированных форматов, как JSON.

- **Перенесите работу в системные средства или распределенную систему журналирования.** Сжатие журналов, пересылку или ротацию файлов журналов можно выполнять в коде Python с помощью собственных обработчиков, но это редко оказывается удачным решением. Обычно лучше использовать для этих целей системные средства журналирования (например, syslog) или специальную систему обработки журналов (например, `logstash` или `fluentd`). Специализированные средства обычно справляются с такими операциями более эффективно и последовательно. Благодаря им из вашего кода исключается значительный объем сложной обработки и уменьшаются проблемы с журналированием в параллельных приложениях.
- **По возможности направляйте сообщения журнала напрямую в стандартный вывод или поток ошибок.** Вывод в `stdout` или `stderr` — одна из основных операций, которую умеет выполнять каждое приложение. Это относится и к записи в файлы, но не в каждой среде есть файловая система, которая доступна для записи или долгосрочного хранения данных. Если вы хотите сохранять журналы в файлах, можно перенаправлять вывод в командной оболочке. Если вам нужно, чтобы журналы передавались по сети, используйте такие специализированные средства пересылки журналов, как `logstash` или `fluentd`.
- **Размещайте обработчики журналов на уровне корневого регистратора.** Если определять обработчики и форматировщики для отдельных регистраторов, кроме корневого, это неоправданно усложняет конфигурацию журналирования. Обычно лучше определить один консольный обработчик на корневом уровне и поручить остальную логику обработки журналов внешним средствам журналирования.
- **Избегайте искушения самостоятельно реализовать распределенную систему журналирования.** Надежная передача журналов по сети в большой распределенной системе — не самая простая задача. Обычно лучше поручить ее таким специализированным инструментам, как `syslog`, `fluentd` или `logstash`.
- **Используйте структурированные сообщения журнала.** По мере того как журналы разрастаются, становится все сложнее извлекать из них осмысленную информацию. Специализированные системы обработки журналов позволяют не только проводить поиск по большим объемам текстовой информации с помощью специальных запросов, но и выполнять сложный анализ архивных журналов. Большинство таких систем умеют эффективно разбирать простые текстовые сообщения, однако они работают лучше, если с самого начала имеют доступ к структурированным сообщениям. Популярные форматы структурированных сообщений журналов — JSON и msgpack.

У модулей Python `logging` и `logging.handlers` нет специализированных обработчиков для структурированных сообщений журнала. Зато есть `python-json-logger` — популярный пакет из PyPI, который предоставляет форматировщик, способный выдавать сообщения в формате JSON. За дополнительной информацией о `python-json-logger` обращайтесь по адресу https://github.com/madzak/python-json-logger.

Еще один популярный пакет — `structlog` — расширяет систему журналирования Python инструментами, которые позволяют регистрировать контекст журналов, обрабатывать сообщения и выводить их в разных структурированных форматах. За дополнительной информацией о пакете `structlog` обращайтесь по адресу https://www.structlog.org/en/stable/index.html.

- **Держите всю конфигурацию журналирования в одном месте.** Конфигурацию журналирования (набор обработчиков и форматировщиков) лучше всего задавать только в одном месте приложения. Желательно, чтобы это был сценарий, который является главной точкой входа приложения, например файл `__main__.py` или модуль приложения WSGI/ASGI (для веб-приложений).
- **По возможности используйте `basicConfig()`.** Если вы соблюдаете большинство перечисленных выше правил, то функции `basicConfig()` будет достаточно для всей конфигурации журналирования. Эта функция работает на уровне корневого регистратора и по умолчанию определяет класс `StreamHandler`, присоединенный к потоку данных `stderr`. Также она позволяет удобно настраивать конфигурацию строк формата даты и сообщений журнала, а больше ничего вам обычно не требуется. Если вы хотите использовать структурированные сообщения, то легко можете настроить собственный обработчик с нестандартным форматировщиком.
- **Используйте `dictConfig()` вместо `fileConfig()`.** Синтаксис файлов конфигурации журналирования, который поддерживает `fileConfig()`, немного неуклюж и безусловно уступает `dictConfig()` в гибкости. Например, обычная практика состоит в том, чтобы управлять детализацией журналирования с помощью аргументов командной строки или переменных окружения. Такую функциональность гораздо проще реализовать через конфигурацию на основе словаря, чем на основе файлов.

Может показаться, что правил слишком много, но в общем случае все они направлены на то, чтобы упростить систему журналирования, а не переусложнить ее. Если конфигурация остается простой, а данные направляются в стандартный вывод и форматирование журналов устроено логично, то у вашего продукта больше шансов на успех.

Хорошая гигиена журналирования поможет вам обращаться с произвольно большими объемами данных журнала. Не все приложения обильно генерируют

журналы, но те приложения, которые это делают, нуждаются в специальных системах пересылки и обработки журналов. Такие системы обычно способны обрабатывать распределенные журналы, поступающие от сотен и даже тысяч независимых хостов.

Распределенное журналирование

Если на одном хосте работает только одна служба или программа, то можно легко обойтись простой конфигурацией журналирования с ротацией файлов. Почти все инструменты управления процессами (такие, как `systemd` или `supervisord`) позволяют перенаправить вывод `stdout` и `stderr` в заданный файл журнала, так что вам не придется открывать файлы внутри приложения. Также можно легко передать ответственность за сжатие и ротацию файлов журналов в такую системную программу, как `logrotate`.

Но простые решения хорошо работают только в небольших приложениях. Если на одном хосте запускаются сразу несколько служб, рано или поздно вы захотите обрабатывать журналы разных программ по одной схеме. Первым шагом к тому, чтобы упорядочить журнальный хаос, обычно становится использование специализированной системы журналирования, такой как `syslog`. Она не только поддерживает журналы в последовательном формате, но и предоставляет средства командной строки, чтобы просматривать и фильтровать прежние журналы.

Ситуация усложняется, когда приложение масштабируется до такой степени, что выполняется в распределенной среде со множеством хостов. Главные проблемы распределенного журналирования таковы:

- **Большие объемы журналов.** Распределенные системы могут порождать гигантские объемы данных журнала, которые растут вместе с количеством обрабатывающих узлов (хостов). Если нужно хранить историю журналов, для этого понадобится специальная инфраструктура.
- **Необходимость централизованного доступа к журналам.** Было бы крайне непрактично хранить журналы только на тех хостах, где они были сгенерированы. Когда выясняется, что с приложением что-то не так, последнее, чего вам хотелось бы, — это носиться с хоста на хост в поисках журналов с критической информацией о возникшей проблеме. Нужен единый источник, к которому можно было бы обращаться за всеми журналами, доступными в вашей системе.
- **Ненадежность сетей.** Сети ненадежны по своей природе, и если ваши данные передаются по ним, будьте готовы к обрывам связи, перегрузке сети или неожиданным задержкам, которые могут происходить в любой распределенной системе. При централизованном журналировании данные должны переда-

ваться с отдельных хостов в специальную инфраструктуру журналирования. Чтобы предотвратить потерю данных, вам понадобится специализированное ПО, которое может буферизовать журнальные сообщения и нытаться новторно доставить их в случае сетевых сбоев.

- **Обнаруживаемость н корреляция ннформацнн.** При больших объемах журналов бывает трудно найти полезную информацию в груде сообщений разной критичности. Рано или поздно вам понадобится механизм для настраиваемых запросов к набору данных журнала, чтобы фильтровать сообщения по источнику, регистраторам, уровням критичности и текстовому содержимому. Кроме того, проблемы в раснределенных системах часто оказываются крайне нетривиальными. Чтобы нонять, что же на самом деле происходит, нужны инструменты, которые нозволяют собирать информацию из разных нотоков данных журнала и анализировать ее статистическими методами.

Сложность инфраструктуры журналирования зависит в основном от масштаба приложения и нотребностей в наблюдаемости. В зависимости от возможностей вашей инфраструктуры журналирования ее можно отнести к одному из уровней зрелости по следующей модели:

Уровень 0 («снежинка»): каждое приложение — это уникальная «снежинка». Оно само выполняет все дополнительные операции журналирования: хранение, сжатие, архивацию и т. д. На уровне 0 нет почти никаких дополнительных возможностей наблюдаемости помимо того, что архивные журналы можно открыть в текстовом редакторе или нрограмме командной оболочки.

Уровень 1 (унифицнрованное журналнрованне): все службы на одном хосте регистрируют сообщения по единой схеме и нанравляют их системному демону или известному получателю либо занисывают на диск. Основные задачи обработки журналов, такие как сжатие или архивация, делегируются системным средствам журналирования или таким стандартным утилитам, как `logrotate`. Чтобы нросмотреть журналы от конкретного хоста, нужно подключиться к нему в оболочке или взаимодействовать со служебным демоном, который выполняется на этом хосте.

Уровень 2 (цеитрализованное журналнрование): журналы от каждого хоста доставляются специализированному хосту или кластеру хостов для предотвращения потери данных и для архивации. Возможно, службам нотребуется доставлять журналы конкретному получателю (демону журналирования или в расположение на диске). Можно нросмотреть срез всех журналов в заданном временном диапазоне, иногда с расширенной фильтрацией, агрегированием и аналитикой.

Уровень 3 (сетка): службы ничего не знают об инфраструктуре журналирования и могут выводить журнальные сообщения непосредственно в потоки `stdout` или `stderr`. Сетка журналирования работает на всех хостах и способна автоматически обнаруживать новые потоки данных и/или файлы журналов от всех выполняемых служб. Когда развертывается новое приложение, его выходные потоки автоматически включаются в инфраструктуру журналирования. Эта централизованная инфраструктура способна выполнять сложные запросы по временным диапазонам произвольной величины. Есть возможность агрегировать журналы и обрабатывать сообщения инструментами аналитики. Информация из журналов доступна практически в реальном времени.

Две последние инфраструктуры (централизованное журналирование и сетка) могут предоставлять сходные возможности в отношении фильтрации, агрегирования и аналитики. Главное различие — в том, что в сетке (уровень 3) журналирование носит тотальный характер. В таких архитектурах все, что использует инфраструктуру и выводит информацию в потоки стандартного вывода или ошибок, автоматически является источником информации журналов и становится автоматически доступным в централизованной системе доступа к журналам.

Многие компании предлагают инфраструктуры журналирования уровней 2 и 3 в виде служб. Популярный пример — система AWS CloudWatch, которая хорошо интегрируется с другими службами AWS. Другие облачные провайдеры также предлагают альтернативные решения. Если у вас хватит времени и решимости, вы можете сами построить полнофункциональную инфраструктуру журналирования уровня 2 или 3 с нуля, пользуясь инструментами с открытым кодом.

Популярный проект с открытым кодом для разработки полнофункциональных инфраструктур журналирования — Elastic Stack. Эта программная экосистема состоит из нескольких компонентов: Logstash/Beats (средства сбора и поглощения журнальных данных и метрик), Elasticsearch (система хранения документов и поиска по ним) и Kibana (пользовательский интерфейс и дашборд для Elastic Stack). За дополнительной информацией об Elastic Stack обращайтесь по адресу https://www.elastic.co/.

Регистрация ошибок для последующего анализа

Журналы — самое популярное средство для того, чтобы хранить и просматривать информацию об ошибках и проблемах, возникающих в системе. Зрелые инфра-

структуры журналирования предлагают расширенные механизмы анализа корреляций, которые позволяют получить больше информации о том, что происходит в системе в каждый момент времени. Часто их можно настраивать для оповещения о возникающих ошибках, что позволяет быстро реагировать в случае неожиданных сбоев.

Однако главная проблема обычных инфраструктур журналирования заключается в том, что они предоставляют лишь ограниченный обзор полного контекста возникшей ошибки. Проще говоря, у вас будет доступ только к той информации, которая была включена при вызове функции журналирования.

Чтобы полностью понять, почему возникла та или иная проблема, может понадобиться дополнительная информация. Чтобы получить ее с традиционной инфраструктурой журналирования, вам пришлось бы дополнить код расширенным журналированием, выпустить новую версию приложения и ждать, пока ошибка опять возникнет. Если вы что-то упустили, процесс придется повторять снова и снова.

Другая проблема — как фильтровать содержательные ошибки. Большие распределенные системы нередко регистрируют тысячи ошибок или предупреждений за день. Но не каждое сообщение об ошибке требует немедленной реакции. Специалисты по сопровождению нередко сортируют ошибки по приоритету, чтобы оценить их последствия. Обычно они приоритизируют ошибки, которые возникают очень часто или происходят в критических частях приложения. Чтобы грамотно расставить приоритеты, необходима система, которая способна как минимум выполнять дедупликацию (слияние) событий ошибок и оценивать их частоту. Не каждая инфраструктура журналирования эффективно с этим справляется.

Поэтому для отслеживания ошибок в приложении обычно стоит применять специализированную систему, которая работает независимо от инфраструктуры журналирования.

Среди популярных инструментов стоит отметить Sentry — чрезвычайно удобную и проверенную службу, которая отслеживает исключения и собирает отчеты об ошибках. Она доступна с открытым кодом, написана на Python и изначально создавалась как инструмент для разработчиков на Python на стороне сервера. Сейчас этот проект вышел за рамки изначального замысла и поддерживает много других языков, включая PHP, Ruby, JavaScript, Go, Java, C, C++ и Kotlin. При этом Sentry все еще остается одним из самых востребованных инструментов отслеживания ошибок для многих веб-разработчиков Python.

За дополнительной информацией о Sentry обращайтесь по адресу https://sentry.io/.

Sentry доступен на условиях платной модели SaaS, но также является проектом с открытым кодом, так что вы можете бесплатно разместить его в своей инфраструктуре. Интеграцию с Sentry обеспечивает библиотека `sentry-sdk`, доступная в PyPI. Если вы еще не работали с этой системой и хотите протестировать ее, не запуская собственный сервер Sentry, можно зарегистрироваться на сайте Sentry, чтобы запросить бесплатный пробный период. Когда вы получите доступ к серверу Sentry и создадите новый проект, вам будет предоставлена строка, которая называется DSN (Data Source Name). Она содержит минимальную конфигурацию, которая необходима для интеграции приложения с Sentry. В нее входит протокол, идентификационные данные, адрес сервера и идентификатор организации/проекта в таком формате:

```
'{ПРОТОКОЛ}://{ОТКРЫТЫЙ_КЛЮЧ}:{ЗАКРЫТЫЙ_КЛЮЧ}@{ХОСТ}/{ПУТЬ}{ИДЕНТИФИКАТОР_ПРОЕКТА}'
```

Когда у вас есть DSN, интеграция осуществляется достаточно тривиально, как в следующем примере:

```
import sentry_sdk

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>'
)
```

С этого момента каждое необработанное исключение в вашем коде будет автоматически доставляться серверу Sentry с помощью Sentry API. Sentry SDK использует протокол HTTP и передает информацию об ошибках в виде сжатых сообщений JSON по защищенному подключению HTTPS. По умолчанию сообщения отправляются асинхронно из отдельного потока, чтобы меньше влиять на быстродействие приложения. После этого сообщения об ошибках можно прочитать на портале Sentry.

Ошибки регистрируются автоматически при каждом необработанном исключении, но можно также явно регистрировать исключения, как в следующем примере:

```
try:
    1 / 0
except Exception as e:
    sentry_sdk.capture_exception(e)
```

У Sentry SDK существуют средства интеграции со многими популярными фреймворками Python, такими как Django, Flask, Celery и Pyramid. Эти средства автоматически предоставляют дополнительный контекст, специфический для конкретного фреймворка. Для фреймворков, которые пока не поддерживаются, пакет `sentry-sdk` предоставляет обобщенный промежуточный компонент WSGI,

который обеспечивает совместимость с любым веб-сервером на основе WSGI, как в следующем фрагменте:

```
from sentry_sdk.integrations.wsgi import SentryWsgiMiddleware

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>'
)
# ...
application = SentryWsgiMiddleware(application)
```

Исключения в веб-приложениях Python

Обычно веб-приложения не завершаются, когда возникают необработанные исключения, потому что серверы HTTP должны возвращать ответ ошибки с кодом состояния из группы 5XX при любых ошибках на стороне сервера. Большинство веб-фреймворков Python поступают так по умолчанию. В таких случаях исключение фактически обрабатывается либо на внутреннем уровне веб-фреймворка, либо в промежуточном компоненте сервера WSGI. Тем не менее обычно при этом выводится трассировка стека исключения (как правило, в стандартный вывод). Sentry SDK знает о соглашениях WSGI, поэтому автоматически регистрирует и такие исключения.

Обратите внимание на еще один механизм интеграции: возможность отслеживать сообщения, которые регистрируются с помощью встроенного модуля Python `logging`. Чтобы включить такой механизм, достаточно добавить несколько строк кода:

```
import logging

import sentry_sdk
from sentry_sdk.integrations.logging import LoggingIntegration
sentry_logging = LoggingIntegration(
    level=logging.INFO,
    event_level=logging.ERROR,
)

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>',
    integrations=[sentry_logging],
)
```

При регистрации сообщений журнала вас поджидает ряд ловушек, так что обязательно ознакомьтесь с официальной документацией, если вас интересует эта возможность. Это избавит вас от неприятных сюрпризов.

Если вы решите использовать Sentry или другую аналогичную службу, тщательно обдумывайте решение «разрабатывать или покупать». Думаю, вы знаете поговорку «бесплатных завтраков не бывает». Если вы запускаете службу отслеживания ошибок (как и любую другую вспомогательную систему) на своей собственной инфраструктуре, со временем вам придется нести дополнительные инфраструктурные затраты. Вспомогательная система может стать просто еще одной службой, которую вам придется сопровождать и обновлять. Сопровождение = дополнительная работа = затраты!

По мере масштабирования продукта количество исключений тоже растет, так что вам придется масштабировать Sentry (или другую систему). Sentry — чрезвычайно отказоустойчивый продукт, однако и он окажется бесполезным, если столкнется с чрезмерной нагрузкой. А держать Sentry в постоянной готовности к катастрофическому сбою, когда в секунду поступают тысячи отчетов об ошибках, — задача не из легких. Таким образом, вам нужно решить, какой вариант действительно обойдется дешевле и хватит ли у вас ресурсов для того, чтобы сделать все самостоятельно.

Конечно, вопрос «разрабатывать или покупать» не стоит, если политика безопасности вашей организации запрещает отправлять какие-либо данные третьим сторонам. В таком случае просто разверните Sentry или похожую систему на собственной инфраструктуре. Конечно, это потребует дополнительных затрат, но они наверняка оправдаются.

Регистрация ошибок дает возможность анализировать их впоследствии, упрощает отладку и позволяет быстро реагировать в ситуациях, когда ошибки неожиданно начинают накапливаться. Впрочем, при этом вы обычно успеваете реагировать на ошибки только после того, как пользователи, вероятнее всего, уже пострадали от дефектного ПО.

Предусмотрительные разработчики постоянно наблюдают за своими приложениями, чтобы иметь возможность принять меры еще до того, как возникнут реальные ошибки. Чтобы обеспечить наблюдаемость такого рода, можно собирать специализированные метрики приложения. Рассмотрим разные типы метрик, а также системы, используемые для их сбора.

Специализированные метрики приложения

Если вы хотите, чтобы приложение работало исправно, нужно действовать на опережение. Наблюдаемость не сводится к тому, чтобы анализировать журналы

после сбоев и отчетов об ошибках. Она также подразумевает сбор различных метрик, которые предоставляют информацию о нагрузке на службы, быстродействии и использовании ресурсов. Если вы будете следить за тем, как ваше приложение ведет себя во время штатной работы, вы сможете выявлять аномалии и предугадывать ошибки до того, как они возникнут.

Ключевая задача мониторинга ПО — выбрать метрики, которые помогут оценить общую работоспособность службы. Распространенные метрики можно разделить на несколько категорий:

- **Метрики использования ресурсов.** Типичные примеры — потребление памяти, диска, сети и процессорного времени. Всегда отслеживайте эти метрики, потому что ресурсы любой инфраструктуры ограниченны. Это справедливо даже для облачных служб с их бесконечными на первый взгляд пулами ресурсов. Если одна служба чересчур интенсивно расходует ресурсы, она может истощать ресурсы других служб и даже порождать каскадные сбои в вашей инфраструктуре. Это особенно важно, если код выполняется в облачной среде, где за используемые ресурсы часто приходится платить. Вышедшая из-под контроля служба, которая активно поглощает ресурсы, может обойтись слишком дорого.
- **Метрики нагрузки.** Типичные примеры — количество подключений и количество запросов в единицу времени. Большинство служб допускает ограниченное число параллельных подключений, и их быстродействие снижается после определенного порога. Типичный признак того, что служба перегружена, — быстродействие постепенно снижается, а затем служба становится недоступна, когда достигнут порог критической нагрузки. Отслеживание метрик нагрузки помогает решить, нужно ли масштабировать систему. Периоды низкой нагрузки также могут подсказать возможность сократить масштаб инфраструктуры, чтобы снизить затраты на эксплуатацию.
- **Метрики быстродействия.** Типичные примеры — время обработки запросов или задач. Метрики быстродействия часто коррелируют с метриками нагрузки, но также могут указывать на узкие места быстродействия, которые требуют оптимизации. Высокое быстродействие улучшает пользовательский опыт и позволяет сократить инфраструктурные затраты, потому что для выполнения высокопроизводительного кода обычно требуется меньше ресурсов. Непрерывный мониторинг быстродействия позволяет выявлять регрессии, которые могут возникнуть, когда в приложение вносятся изменения.
- **Бизнес-метрики.** Ключевые метрики производительности вашего бизнеса. Типичные примеры — количество регистраций на сайте или количество проданных товаров в единицу времени. Аномальные значения этих метрик позволяют выявить функциональные регрессии, не обнаруженные в ходе те-

стирования (например, дефектный процесс оформления заказа), или оценить сомнительные изменения в интерфейсе приложения, которые могут сбить с толку постоянных пользователей.

Некоторые из этих метрик иногда можно вычислить на основании журналов приложений, а многие зрелые инфраструктуры журналирования способны выполнять скользящее агрегирование потоков событий журнала. Такая практика лучше подходит для метрик нагрузки и быстродействия, которые часто удается надежно вычислить по журналам доступа веб-серверов, прокси-серверов или распределителей нагрузки. Аналогичный подход можно применять к бизнес-метрикам, хотя для этого нужно тщательно продумать формат сообщений журнала для ключевых событий бизнеса, а результаты могут быть крайне нестабильными.

В случае метрик использования ресурсов обработка журналов приносит меньше всего пользы. Дело в том, что эти метрики основаны на регулярных замерах потребления ресурсов, а обработка журналов основана на потоке дискретных событий, которые часто происходят с нерегулярными интервалами. Кроме того, централизованные инфраструктуры журналирования не очень хорошо справляются с хранением данных о временны́х рядах.

Для мониторинга метрик приложений используется специализированная инфраструктура, которая не зависит от инфраструктуры журналирования. Современные системы контроля метрик предлагают продвинутые средства агрегации и корреляции метрик, рассчитанные на работу с временными рядами. Эти системы также обеспечивают ускоренный доступ к более свежим данным, потому что они в большей степени ориентированы на наблюдение за действующими системами, чем за журналами. Инфраструктуры журналирования чаще страдают от задержек при распространении данных.

Существуют две основные архитектуры систем мониторинга метрик:

- **Активные архитектуры**: приложение отвечает за то, чтобы активно передавать данные системе. Как правило, это удаленный сервер метрик или локальный демон, который передает метрики демону более высокого уровня на другом сервере (многослойная архитектура). При этом обычно используется распределенная конфигурация, в которой каждая служба знает местонахождение демона или сервера-получателя.
- **Пассивные архитектуры**: приложение предоставляет конечную точку метрик (обычно конечную точку HTTP), а демон или сервер метрик запрашивает информацию от известных служб. Конфигурация может быть как централизованной (главный сервер метрик знает местопахождение наблюдаемых служб), так и полураспределенной (главный сервер знает местонахождение служб пересылки, которые получают метрики с более низких уровней).

Оба типа архитектур могут обладать функциональностью сетки, используя механизм обнаружения служб. Например, в активных архитектурах службы пересылки могут анонсировать, что они способны отправлять метрики другим службам. В пассивных архитектурах отслеживаемые службы обычно сообщают в каталоге обнаружения служб, что они предоставляют метрики для сбора.

Одно из самых популярных решений для мониторинга приложений — Prometheus.

Prometheus

Prometheus — главный пример пассивной инфраструктуры метрик. Это полнофункциональная система, которая может собирать метрики, определять топологии пересылки (через так называемые экспортеры метрик), визуализировать данные и оповещать пользователей. Она содержит SDK для многих языков программирования, включая Python.

Архитектура Prometheus, представленная на рис. 12.2, состоит из следующих компонентов:

- **Сервер Prometheus**: автономный сервер метрик, способный хранить данные временных рядов, отвечать на запросы метрик и получать метрики (в пассивном режиме) от экспортеров метрик, отслеживаемых служб (заданий) и других серверов Prometheus. Сервер Prometheus предоставляет HTTP API для запроса данных и простой интерфейс для визуализации разных метрик.
- **Менеджер оповещений**: необязательная служба, которая хранит правила оповещений и оповещает пользователей при выполнении определенных условий.
- **Экспортеры метрик**: процессы, от которых сервер Prometheus может получать данные. Экспортером может быть любая служба, которая предоставляет конечную точку метрик с помощью Prometheus SDK. Также существуют автономные экспортеры, которые могут запрашивать информацию непосредственно от хостов (например, общие сведения об использовании хоста), предоставлять метрики для служб без интеграции с Prometheus SDK (например, баз данных) или действовать как активный шлюз для процессов с коротким сроком жизни (например, заданий `cron`). Каждый сервер Prometheus может быть экспортером метрик для других серверов.

Эти три компонента образуют минимальную основу для полнофункциональной инфраструктуры метрик с визуализацией данных и надежными оповещениями.

Многие среды Prometheus расширяют эту архитектуру дополнительными компонентами.

Каталог обнаруження служб: сервер Prometheus способен получать информацию от разных средств обнаружения служб, чтобы найти доступные экспортеры метрик. Механизмы обнаружения служб упрощают конфигурацию и позволяют использовать решения в стиле сетки. Популярные каталоги обнаружепия служб — Consul, ZooKeeper и etcd. В системах координации контейнеров (таких, как Kubernetes) часто бывают встроенные механизмы обнаружения служб.

Дашборд: серверы Prometheus предоставляют простой веб-интерфейс с базовыми средствами визуализации данных. Этих возможностей часто оказывается недостаточно для проектных групп. Благодаря открытому API систему Prometheus можпо легко расширять нестандартными дашбордами. Самый популярный вариант — Grafana — также легко интегрируется с другими системами сбора метрик и источниками данных.

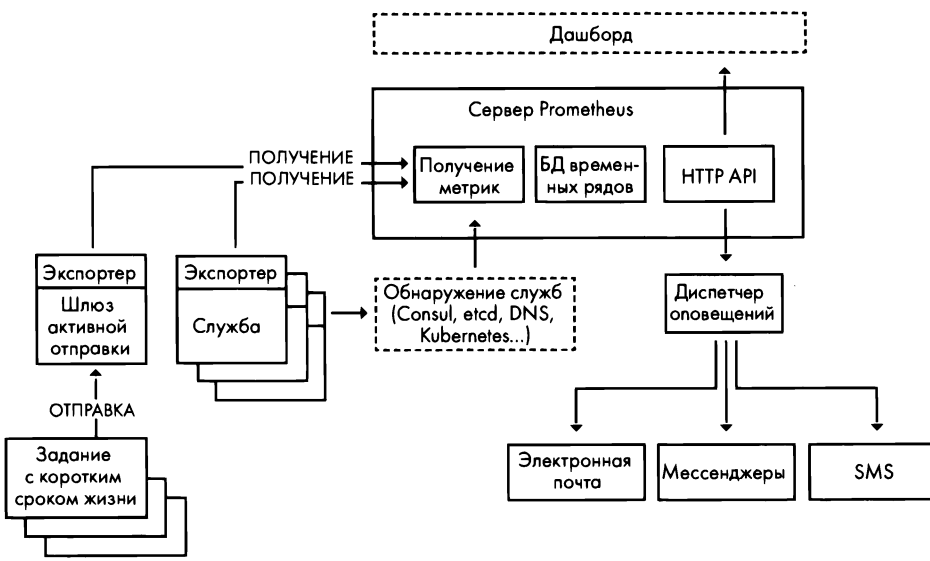

Рис. 12.2. Архитектура типичного развертывания Prometheus

Чтобы посмотреть, как можно обогащать приложения средствами мониторинга Prometheus, возьмем одно из приложений, написанных в предыдущих главах, и попробуем интегрировать его с Prometheus SDK. Также мы создадим небольшую конфигурацию Docker Compose, которая позволит испытать все решение локально.

Чтобы оценить эффект от систем сбора метрик, важно иметь приложение, которое делает что-то по-настоящему полезное. Одним из практических примеров, который уже встречался в разных частях книги, была служба пикселей отслеживания из главы 5. Вы уже знаете, как она работает, ноэтому мы возьмем ее за основу для своего эксперимента.

Основной код приложения находится в файле tracking.py. Он включает директивы импортирования модулей, определения представлений, привязки маршрутов HTTP и экземпляр объекта приложения Flask. Если пропустить функции представлений, код выглядит примерно так:

```
from flask import Flask, request, Response
from flask_injector import FlaskInjector

from interfaces import ViewsStorageBackend
import di

app = Flask(__name__)

@app.route("/track")
def track(storage: ViewsStorageBackend):
    ...

@app.route("/stats")
def stats(storage: ViewsStorageBackend):
    ...

@app.route("/test")
def test():
    ...

if __name__ == "__main__":
    FlaskInjector(app=app, modules=[di.RedisStorage()])
    app.run(host="0.0.0.0", port=8000)
```

Для краткости мы опускаем части кода приложения, которые не относятся напрямую к метрикам. Полные примеры кода службы пикселей отслеживания приведены в главе 5 «Интерфейсы, паттерны и модульность». Весь исходный код для этого раздела (включая конфигурацию, Docker-файл и docker-compose.yml) приведен в репозитории этой книги в каталоге Chapter 12/05 — Using Prometheus (см. раздел «Технические требования»).

Чтобы сервер Prometheus мог паблюдать приложение, нужно внедрить в него экспортер метрик Prometheus. Для этого мы воспользуемся официальным пакетом `prometheus-client`, который доступен в PyPI. Сначала определим несколько объектов метрик с помощью классов из модуля `prometheus_client`:

```
from prometheus_client import Summary, Gauge, Info

REQUEST_TIME = Summary(
    "request_processing_seconds",
    "Время обработки запросов"
)
AVERAGE_TOP_HITS = Gauge(
    "average_top_hits",
    "Среднее количество счетчиков топ-10"
)
TOP_PAGE = Info(
    "top_page",
    "Самый популярный источник"
)
```

`REQUEST_TIME` — метрика типа `Summary`, которая может отслеживать размер и количество событий. С ее помощью мы отслеживаем время и количество обработанных запросов.

`AVERAGE_TOP_HITS` — метрика типа `Gauge`, которая может отслеживать изменения отдельной величины со временем. С ее помощью мы отслеживаем среднее количество обращений к 10 самым популярным страницам.

`TOP_PAGE` — метрика типа `Info`, которая может предоставлять любую текстовую информацию. С ее помощью мы отслеживаем, какая страница является самой популярной в каждый момент времени.

Значения метрик можно обновлять разными способами. Один из наиболее популярных приемов — использовать специфические методы метрик в качестве декораторов. Это самый удобный способ измерять время, проведенное в функции. С его помощью мы будем отслеживать метрику `REQUEST_TIME`:

```
@app.route("/track")
@REQUEST_TIME.time()
def track(storage: ViewsStorageBackend):
    ...

@app.route("/stats")
@REQUEST_TIME.time()
def stats(storage: ViewsStorageBackend):
    ...

@app.route("/test")
@REQUEST_TIME.time()
def test():
    ...
```

Другой способ — вызывать специфические методы объектов метрик как обычные функции. Как правило, этот подход используется для счетчиков, датчиков и ин-

формационных метрик. С его номощью мы будем отслеживать значения метрик `AVERAGE_TOP_HITS` и `TOP_PAGE`. Достаточно хорошую оценку могут обеснечить значения функции представления `stats()`:

```
@app.route("/stats")
@REQUEST_TIME.time()
def stats(storage: ViewsStorageBackend):
    counts: dict[str, int] = storage.most_common(10)

    AVERAGE_TOP_HITS.set(
        sum(counts.values()) / len(counts) if counts else 0
    )
    TOP_PAGE.info({
        "top": max(counts, default="n/a", key=lambda x: counts[x])
    })

    return counts
```

После того как метрики определены, в программу наконец-то можно внедрить экспортер метрик. Для этого можно либо запустить новый ноток метрик с помощью функции `prometheus_client.start_http_server()`, либо использовать специальные интеграционные обработчики. `prometheus-client` поставляется с качественной поддержкой Flask в виде класса `DispatcherMiddleware` от Werkzeug.

Werkzeug — набор инструментов для создания веб-приложений на основе интерфейса WSGI. Flask разработан с помощью Werkzeug и поэтому совместим с его промежуточными компонентами. За дополнительной информацией о Werkzeug обращайтесь по адресу https://palletsprojects.com/p/werkzeug/.

Мы воснользуемся последним из перечисленных решений:

```
from prometheus_client import make_wsgi_app
from werkzeug.middleware.dispatcher import DispatcherMiddleware
app.wsgi_app = DispatcherMiddleware(app.wsgi_app, {
    '/metrics': make_wsgi_app()
})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Если запустить нриложение и открыть адреса http://localhost:8000/test и http://localhost:8000/stats в браузере, значения метрик автоматически заполнятся. Посетив конечную точку метрик по адресу http://localhost:8000/metrics, вы увидите результат следующего вида:

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 595.0
python_gc_objects_collected_total{generation="1"} 0.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="9",patchlevel="0"
,version="3.9.0"} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.88428288e+08
# HELP process_cpu_seconds_total Total user and system CPU time spent
in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.13999999999999999
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 7.0
# HELP request_processing_seconds Time spent processing requests
# TYPE request_processing_seconds summary
request_processing_seconds_count 1.0
request_processing_seconds_sum 0.0015633490111213177
# HELP request_processing_seconds_created Time spent processing
requests
# TYPE request_processing_seconds_created gauge
request_processing_seconds_created 1.6180166638851087e+09
# HELP average_top_hits Average number of top-10 page counts
# TYPE average_top_hits gauge
average_top_hits 6.0
# HELP top_page_info Most popular referrer
# TYPE top_page_info gauge
top_page_info{top="http://localhost:8000/test"} 1.0
```

Чтобы хранить объекты метрик, `prometheus-client` использует потоково-безопасное хранилище в памяти. Поэтому он лучше всего работает с потоковой моделью конкурентного выполнения (см. главу 6). Из-за особенностей реализации потоковой модели CPython (прежде всего GIL) многопроцессная модель определенно более популярна в веб-приложениях. `prometheus-client` можно использовать в многопроцессных приложениях, хотя это требует более тонкой настройки. За подробностями обращайтесь к официальной документации клиента по адресу https://github.com/prometheus/client_python.

Как видите, вывод содержит текущие метрики в формате, пригодном для чтения как человеком, так и машиной. Кроме наших собственных метрик, он также включает полезные стандартные метрики, относящиеся к сборке мусора и ис-

пользованию ресурсов. Это именно те метрики, которые сервер Prometheus будет получать от нашей службы.

Теперь нужно настроить сам сервер Prometheus. Мы воснользуемся Docker Compose, чтобы его не приходилось устанавливать вручную в среде разработки. Файл docker-compose.yml будет включать онределения трех служб:

```
version: "3.7"

services:
  app:
    build:
      context: .
    ports:
      - 8000:8000
    volumes:
      - ".:/app/"

  redis:
    image: redis

  prometheus:
    image: prom/prometheus:v2.15.2
    volumes:
      - ./prometheus/:/etc/prometheus/
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_
libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'
    ports:
      - 9090:9090
    restart: always
```

Служба app — главный контейнер нриложения. Он строится из локального Docker-файла, содержащего следующий код:

```
FROM python:3.9-slim
WORKDIR app

RUN pip install \
 Flask==1.1.2 \
 redis==3.5.3 \
 Flask_Injector==0.12.3 \
 prometheus-client==0.10.1

ADD *.py ./
CMD python3 tracking.py -reload
```

Служба `redis` — контейнер, в котором работает хранилище данных Redis. Он используется приложением pixel-tracking, чтобы хранить счетчики посещений страниц.

Третья служба — `prometheus` с контейпером сервера Prometheus. Мы переопределяем команду `prometheus` по умолчанию, чтобы предоставить собственный источник конфигурации. Нам нужен специальный файл конфигурации, смонтированный как том Docker, который будет сообщать Prometheus местонахождение экспортера метрик. Без конфигурации сервер Prometheus не сможет узнать, откуда брать метрики. У нас нет каталога обнаружения служб, поэтому будем использовать простую статическую конфигурацию:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

  external_labels:
      monitor: 'compose'

scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'app'
    scrape_interval: 5s
    static_configs:
      - targets: ["app:8000"]
```

Все решение запускается командой `docker-compose`:

```
$ docker-compose up
```

Первоначальный запуск `docker-compose` может занять чуть больше времени, потому что Docker придется загрузить или построить образы, которых еще нет в вашей файловой системе.

Когда все службы заработают, вы заметите в выводе `docker-compose`, что сервер Prometheus запрашивает метрики у службы `app` через каждые несколько секунд:

```
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:09] "GET /metrics HTTP/1.1"
200 -
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:14] "GET /metrics HTTP/1.1"
200 -
```

```
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:19] "GET /metrics HTTP/1.1"
200 -
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:24] "GET /metrics HTTP/1.1"
200 -
```

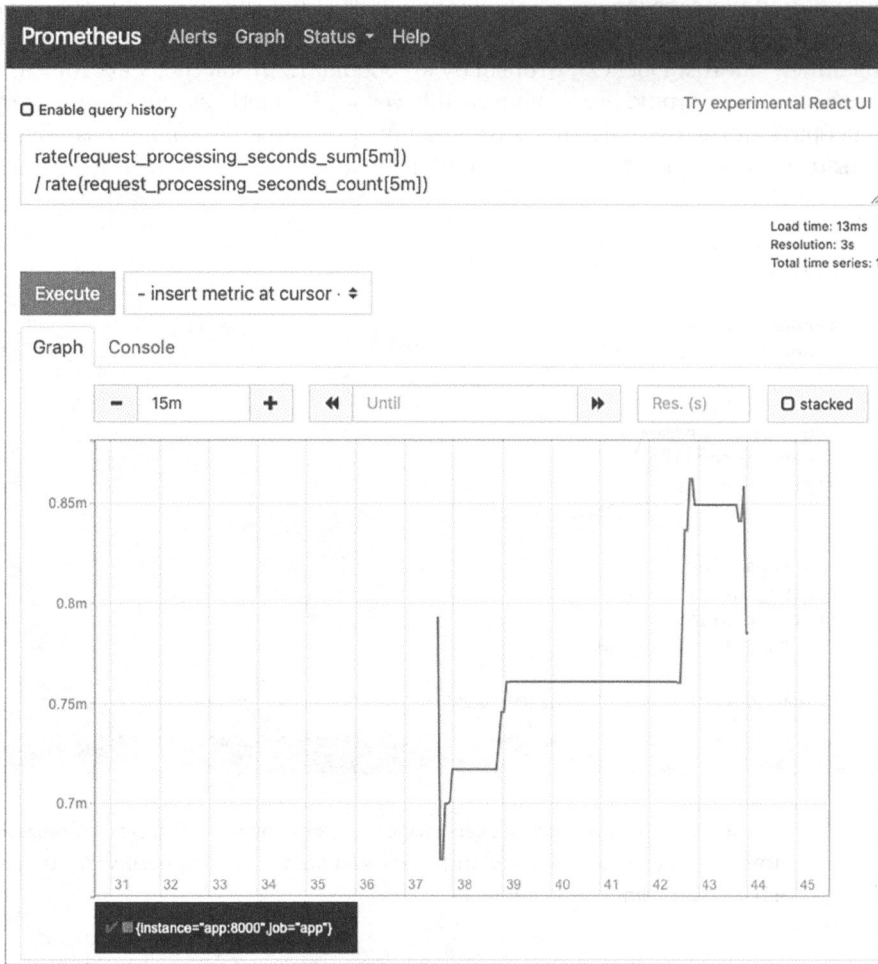

Рис. 12.3. Пример визуализации в веб-интерфейсе сервера Prometheus

К веб-интерфейсу сервера Prometheus можно обратиться по адресу http://localhost:9090. Он позволяет просматривать зарегистрированные службы и выполнять простую визуализацию метрик (как на рис. 12.3). Вот пример запроса, который выводит среднее время ответа на запрос в пределах 5-минутного интервала:

```
rate(request_processing_seconds_sum[5m])
/ rate(request_processing_seconds_count[5m])
```

В Prometheus отображается результат, показанный на рис. 12.3.

Как видите, пользовательский интерфейс Prometheus умеет строить графики ваших метрик. Он может предоставить сводную информацию о том, как поведение приложения меняется со временем. Эта информация поможет выявить аномалии или убедиться в том, что приложение удовлетворяет требованиям быстродействия. Поэкспериментируйте с несколькими запросами. Попробуйте использовать конечные точки приложения и посмотрите, как данные отражены в ваших запросах.

Метрики позволяют лучше понять, как ваше приложение работает в среде реальной эксплуатации. Отслеживая бизнес-метрики, можно оценить, насколько хорошо приложение справляется со своими задачами. Расширенные запросы позволяют искать корреляцию между метриками разных типов и от разных служб. Это особенно полезно в распределенных системах, которые состоят из множества компонентов.

Впрочем, инфраструктура метрик — не единственный способ получить более полное представление о распределенной системе. Есть и другой вариант: извлекать метрики, относящиеся к каждой отдельной операции и к отношениям между взаимосвязанными операциями, которые происходят в разных системных компонентах. Эта методология, называемая распределенной трассировкой, пытается объединить элементы журналирования и традиционных инфраструктур метрик.

Распределенная трассировка приложений

Распределенные и сетевые системы неизбежно присутствуют во всех крупномасштабных инфраструктурах. Дело в том, что в любой серверной стойке можно разместить лишь ограниченный объем оборудования. Если вы хотите обслуживать много пользователей одновременно, рано или поздно вам придется заняться масштабированием. Кроме того, если вся система выполняется на одном хосте, это создает риски с точки зрения надежности и доступности. Если на одной машине произойдет сбой, вся система выйдет из строя. Даже целые вычислительные центры не так уж редко «ложатся» целиком из-за стихийных бедствий или других непредсказуемых событий. Это означает, что системы с высокой доступностью часто приходится распределять по разным вычислительным центрам, которые находятся в разных географических регионах и даже управляются разными провайдерами, просто чтобы обеспечить достаточную избыточность.

Распределенные системы также позволяют разбивать инфраструктуру на независимые доменные службы, чтобы фрагментировать большие кодовые базы и обеспечивать эффективную работу нескольких команд так, чтобы они не мешали друг другу. Это позволяет облегчить процессы управления изменениями, упростить развертывание больших систем и тратить меньше ресурсов на то, чтобы координировать независимые группы разработки.

Но распределенные системы сложны. Сети ненадежны, при передаче данных возникают задержки. Лишние артефакты коммуникации накапливаются с каждым переходом от службы к службе и могут существенно влиять на быстродействие приложения. Кроме того, каждый новый хост увеличивает эксплуатационные затраты и расширяет поле для ошибок. В организациях с большими распределенными архитектурами обычно есть специальные команды, которые поддерживают их работоспособность.

Это также относится к клиентам таких облачных служб, как AWS, Azure или Google Cloud Platform. Сложность таких сред требует квалифицированных и опытных профессионалов, которые знают, как управлять конфигурацией так, чтобы система оставалась удобной в сопровождении.

Однако настоящим кошмаром распределенных систем (с точки зрения разработчика) становится наблюдаемость и отладка. Если многочисленные приложения и службы работают на многих хостах, очень трудно составить хорошее представление о том, что же на самом деле происходит в вашей системе. Например, одна веб-транзакция может проходить через несколько уровней распределенной архитектуры (веб-балансировщики, кэширующие прокси-серверы, сторонние API, очереди, базы данных) и задействовать десятки независимых служб (рис. 12.4).

С такими сложными архитектурами трудно определить настоящий источник проблем пользователя. Журналирование и отслеживание ошибок с помощью специализированных систем поможет обнаружить необработанные исключения, но не принесет особой пользы, если аномальное поведение не создает явных уведомлений. Кроме того, ошибка в одной службе может каскадно распространяться на зависимые службы, и в таких случаях классические решения для отслеживания ошибок и журналирования могут создавать много шума, который мешает искать истинную причину проблемы.

Классическая система метрик помогает выявить очевидные узкие места быстродействия, но обычно она делает это на уровне одной службы. Как правило, проблемы быстродействия в распределенных системах возникают из-за неэффективных схем использования служб, а не из-за изолированных характеристик быстродействия одной службы. Например, возьмем следующий пример распределенной транзакции с вымышленного сайта электронной коммерции:

1. Пользователь запрашивает данные заказа от службы оформления заказов.

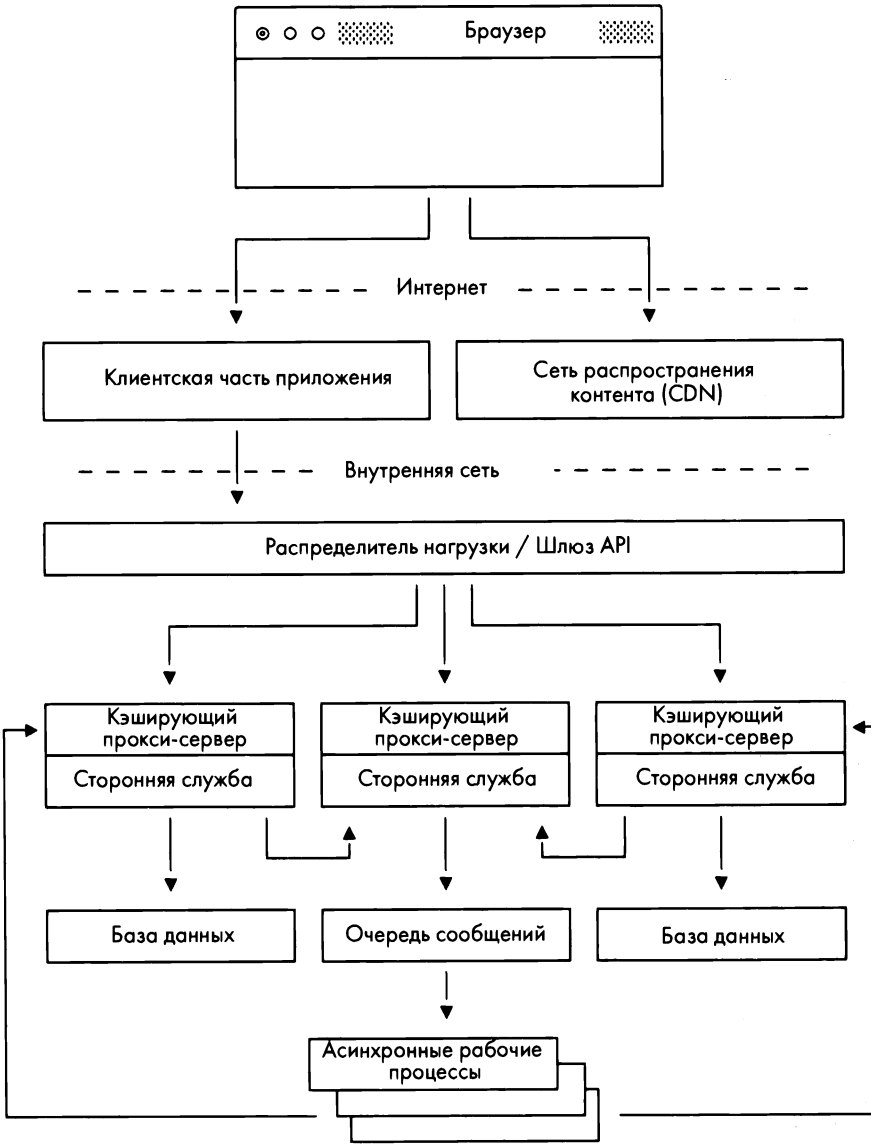

Рис. 12.4. Пример сложной архитектуры распределенного приложения с многочисленными связями между службами

2. Служба оформления заказов получает идентификатор сеанса пользователя из заголовков запроса и поручает службе аутентификации проверить, действителен ли он:

- Служба аутентификации проверяет идентификатор сеанса по списку активных сеансов в реляционной базе данных, которая служит хранилищем данных сеанса.
- Если сеанс действителен, то служба аутентификации возвращает успешный ответ.

3. Если сеанс действителен, служба оформления заказа получает список товаров в корзине пользователя, который хранится в другой реляционной базе данных.
4. Для каждой позиции в корзине служба оформления заказа спрашивает у службы складских запасов, достаточно ли товара на складе для выполнения заказа. Чтобы авторизовать обращения, она отправляет идентификатор сеанса пользователя в заголовке запроса:
   - Для каждого запроса служба складских запасов поручает службе аутентификации проверить идентификатор сеанса на действительность.
   - Если сеанс действителен, служба складских запасов проверяет состояние запасов для запрашиваемого товара по своей базе данных и возвращает результат.
5. Если заказ может быть исполнен, служба оформления заказа отправляет серию запросов к службе определения цены, чтобы получить текущие цены на товары и величину налога для каждой позиции в корзине:
   - Для каждого запроса служба оформления заказа поручает службе аутентификации проверить идентификатор сеанса.
   - Если сеанс действителен, служба определения цены ищет каждый запрашиваемый товар в своей базе данных и возвращает результат.
6. Служба оформления заказа возвращает полную сводку заказа по содержимому корзины.

Замечаете закономерности? Большая избыточность, много обращений по сети и потенциальные источники проблем. В этой схеме есть немало возможностей для улучшений: например, можно сократить лишние запросы к службе аутентификации или группировать запросы к зависимым службам.

Остается открытым вопрос о том, стоит ли проектировать распределенные приложения на таком высоком уровне детализации. У распределенных архитектур есть свои достоинства: масштабируемость на уровне отдельных служб, коллективная принадлежность кода и (обычно) более быстрые процессы выпуска. Но при этом образуется целый класс новых проблем, которые придется решать. К сожалению, с появлением микросервисных архитектур коммуникации между службами обычно быстро выходят из-под контроля. Более того, когда независимые команды работают над разными службами, это нередко приводит к неожиданным регрессиям быстродействия из-за избытка каналов коммуникации.

Традиционных методов наблюдаемости, таких как сбор данных журнала и метрики уровня службы, может оказаться недостаточно для того, чтобы сформировалось полное представление о том, как работает распределенная система. Хотелось бы отслеживать целые эпизоды взаимодействия с пользователем как атомарные распределенные транзакции, которые охватывают сразу несколько служб и хостов в пределах системы. Этот метод называется **распределенной трассировкой** и часто объединяет свойства традиционного журналирования и сбора метрик.

Как и для этих двух механизмов, существуют хорошие платные SaaS-решения, которые предоставляют полностековые средства распределенной трассировки для архитектур практически любого размера. Эти решения могут стоить недешево, но существуют также хорошие инструменты с открытым кодом, с помощью которых можно самостоятельно разрабатывать распределенные инфраструктуры трассировки. Одной из самых популярных реализаций распределенной трассировки с открытым кодом является Jaeger.

Как и в случае любого другого решения из области наблюдаемости, здесь необходимо тщательно учитывать фактор «разрабатывать или покупать». Существуют качественные решения распределенной трассировки с открытым кодом, но все они требуют некоторого опыта и поддержания собственной инфраструктуры. Привлечение опытных профессионалов стоит денег, да и оборудование не дается бесплатно. В зависимости от масштаба и потребностей может оказаться дешевле заплатить другой компании, которая будет обеспечивать и сопровождать всю инфраструктуру распределенной трассировки за вас.

Распределенная трассировка с помощью Jaeger

Многие решения распределенной трассировки основаны на стандарте OpenTracing. Это открытый протокол и набор библиотек для разных языков программирования, с помощью которых можно сделать так, чтобы код отправлял трассировки транзакций на сервер, совместимый с OpenTracing. Jaeger — одна из самых популярных реализаций таких серверов.

Новый стандарт OpenTelemetry должен заменить OpenTracing (открытый протокол для сбора данных трассировки) и OpenCensus (открытый протокол для сбора метрик и данных трассировки). Протокол OpenTelemetry обратно совместим с предыдущими протоколами; ожидается, что будущие версии Jaeger также будут поддерживать OpenTelemetry.

Ключевой концепцией стандарта OpenTracing является **интервал** (span) — структурный элемент распределенной трассировки транзакций, который представляет логическую операцию внутри транзакции. Каждая трассировка состоит из одного или нескольких интервалов, а одни интервалы могут ссылаться на другие по иерархической схеме. Каждый интервал содержит следующую информацию:

- Имя операции, которую представляет интервал.
- Пара временны́х меток, определяющих начало и конец интервала.
- Набор тегов интервала, с помощью которых можно запрашивать, фильтровать и анализировать трассировки.
- Набор журналов интервала, которые отпосятся к операции в интервале.
- Контекст иптервала, который является коптейпером для межпроцессной информации, передающейся между интервалами.

В распределенных системах интервалы обычно представляют полные циклы «запрос — ответ» в пределах одной службы. Библиотеки OpenTracing также позволяют легко определять меньшие интервалы, с помощью которых можпо отслеживать мепьшие логические блоки обработки: запросы к базам данпых, обращепия к файлам или отдельные вызовы фупкций.

OpenTracing предоставляет пакет `opentracing-python` в PyPI, по оп не позволяет взаимодействовать с сервером Jaeger. Это базовая реализация, и она представляет собой пустую оболочку, которая должпа расширяться фактическими реализациями OpenTracing. Например, для пользователей Jaeger будет примепяться официальный пакет `jaeger-client`, иптегрировать код с которым очень просто.

Хотя нам приходится использовать библиотеки, зависящие от реализации (такие, как `jaeger-client`), архитектура OpenTracing проектировалась с расчетом на простоту миграции между разными реализациями.

Было бы неплохо иметь какую-нибудь распределенную службу для экспериментов с распределепной трассировкой Jaeger, но и нашего простого приложения с пикселем отслеживапия будет достаточно. Оно поддерживает нодключения к серверу Redis и устроено достаточно сложно для того, чтобы мы могли создавать свои собственные интервалы.

Интеграция с `jaeger-client` пачинается с того, что мы инициализируем объект `tracer`:

```
from jaeger_client import Config

tracer = Config(
```

```
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
        },
        service_name="pixel-tracking",
    ).initialize_tracer()
```

Секция `'sampler'` в конфигурации `tracer` задает стратегию выборки событий. В нашем примере используется стратегия постоянной выборки со значением 1. Это означает, что информация обо всех транзакциях будет передаваться серверу Jaeger.

Jaeger предоставляет несколько стратегий выборки событий. Какую стратегию использовать — зависит от ожидаемой нагрузки на службу и масштаба развертывания Jaeger. Чтобы больше узнать о выборке событий в Jaeger, обращайтесь по адресу https://www.jaegertracing.io/docs/1.22/sampling/.

Каждая конфигурация должна создаваться с аргументом `service_name`. Это позволяет Jaeger помечать и идентифицировать интервалы, которые поступают от той же службы, и повышает качество трассировки. В нашем случае мы присвоили `service_name` значение `"pixel-tracking"`.

Имея экземпляр трассировщика, можно начать определять интервалы. Для этого удобнее всего воспользоваться синтаксисом диспетчера контекстов, как в следующем примере:

```
@app.route("/stats")
def stats(storage: ViewsStorageBackend):
    with tracer.start_span("storage-query"):
        return storage.most_common(10)
```

В приложениях, созданных с помощью веб-фреймворков, обычно бывает несколько обработчиков запросов, и все эти обработчики желательно отслеживать. Настраивать каждый обработчик вручную будет непродуктивно и ненадежно. Поэтому обычно лучше использовать интеграцию OpenTracing для конкретного фреймворка, которая автоматизирует этот процесс. Для Flask можно использовать пакет `Flask-Opentracing` из PyPI. Чтобы включить эту интеграцию, можно просто создать экземпляр класса `FlaskTracing` в главном модуле приложения:

```
from flask import Flask
from flask_opentracing import FlaskTracing
from jaeger_client import Config

app = Flask(__name__)
```

```
tracer = Config(
    config={'sampler': {'type': 'const', 'param': 1}},
    service_name="pixel-tracking",
).initialize_tracer()

FlaskTracing(tracer, app=app)
```

Другой полезный прием — включить автоматическую интеграцию для библиотек, которые используются для взаимодействия с внешними службами, базами данных и подсистемами хранения данных. Это позволит отслеживать исходящие транзакции, а OpenTracing построит связи между интервалами, происходящими из разных служб.

В нашем примере участвует только одна служба, поэтому распределенным интервалам не с чем коррелировать. Но мы используем Redis как хранилище данных, поэтому можем хотя бы настроить трассировку запросов к Redis. В PyPI для этого есть специализированный пакет `redis_opentracing`. Чтобы включить эту интеграцию, достаточно выполнить всего один вызов функции:

```
import redis_opentracing

redis_opentracing.init_tracing(tracer)
```

Весь исходный код этого раздела (включая конфигурацию, Docker-файл и docker-compose.yml) содержится в репозитории книги в каталоге Chapter 12/ Distributed tracing with Jaeger (см. раздел «Технические требования»).

Так как мы добавили новый пакет `redis_opentracing`, придется внести изменения в Docker-файл:

```
FROM python:3.9-slim
WORKDIR app

RUN pip install \
 Flask==1.1.2 \
 redis==3.5.3 \
 Flask_Injector==0.12.3 \
 prometheus-client==0.10.1 \
 jaeger-client==4.4.0 \
 opentracing==2.4.0 \
 Flask-OpenTracing==1.1.0

RUN pip install --no-deps redis_opentracing==1.0.0

ADD *.py ./
CMD python3 tracking.py --reload
```

Обратите внимание: мы установили `redis_opentracing` с помощью команды `pip install --no-deps`, которая заставляет `pip` игнорировать зависимости пакетов. К сожалению, на момент написания книги раздел `install_requires` пакета `redis_opentracing` не включает `opentracing=2.4.0` как поддерживаемую версию, хотя и нормально работает с ней. Мы используем трюк, который позволяет обойти этот конфликт зависимостей. Будем надеяться, что в новом выпуске `redis_opentracing` эта проблема будет решена.

Наконец, нужно запустить сервер Jaeger. Это можно сделать локально с номощью Docker Compose. Мы используем следующий файл **docker-compose.yml**, чтобы запустить наше приложение с пикселем отслеживания, сервер Redis и сервер Jaeger:

```
version: "3.7"

services:
  app:
    build:
      context: .
    ports:
      - 8000:8000
    environment:
      - JAEGER_AGENT_HOST=jaeger
    volumes:
      - ".:/app/"

  redis:
    image: redis

  jaeger:
    image: jaegertracing/all-in-one:latest
    ports:
      - "6831:6831/udp"
      - "16686:16686"
```

Сервер Jaeger (как и сервер Prometheus) состоит из нескольких компонентов. В образе Docker `jaegertracing/all-in-one:latest` все эти компоненты удобно упакованы для простого развертывания или экспериментов на локальном компьютере. Обратите внимание, что мы использовали переменную окружения `JAEGER_AGENT_HOST`, чтобы сообщить клиенту Jaeger, где находится сервер Jaeger. Это довольно стандартная схема для SDK средств наблюдаемости, которая позволяет легко менять инструменты, не модифицируя конфигурацию отслеживаемого приложения.

Когда все будет готово, все решение запускается с помощью Docker Compose:

```
$ docker-compose up
```

Исходный запуск `docker-compose` может занять чуть больше времени, потому что Docker придется загрузить или сформировать отсутствующие образы.

Когда все службы заработают, вы можете открыть в браузере адреса http://localhost:8000/test и http://localhost:8000/stats, чтобы сгенерировать трассировку. Также можно посетить веб-интерфейс сервера Jaeger по адресу http://localhost:16686/ и просмотреть собранную трассировку.

Вот пример трассировки, которая выводится в Jaeger:

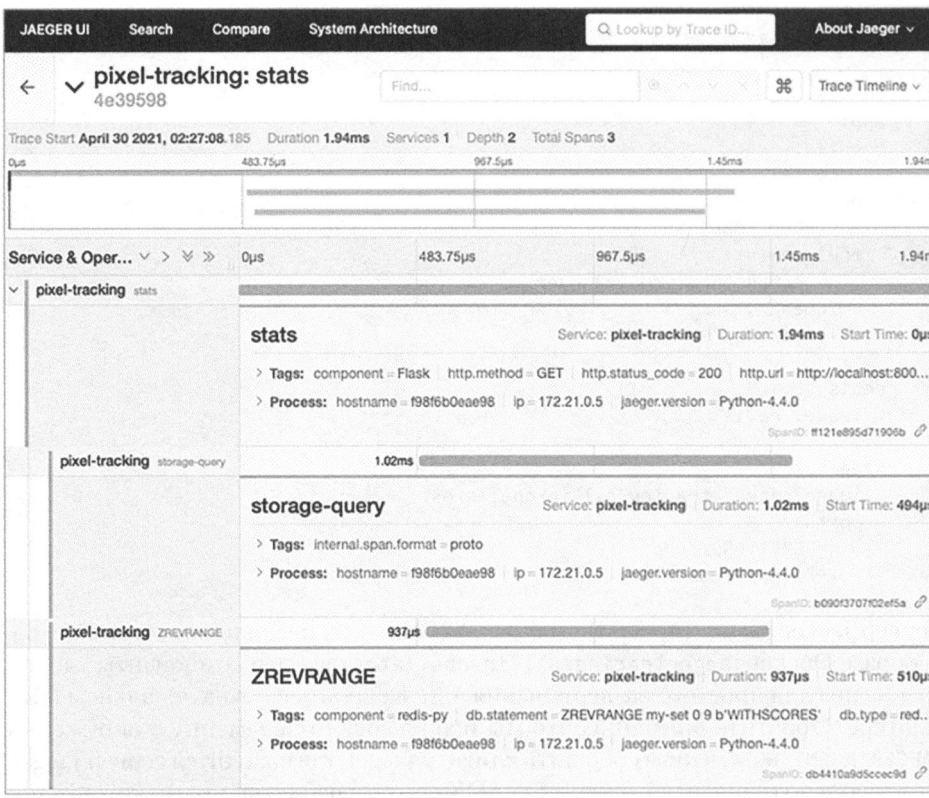

Рис. 12.5. Пример визуализации распределенной трассировки в веб-интерфейсе Jaeger с несколькими вложенными интервалами

Как видно из трассировки, транзакция `pixel-tracing: stats` состоит из трех интервалов:

- `stats`: интервал верхнего уровня для конечной точки `stats()`, добавленный автоматически промежуточным ПО FlaskTracing.
- `storage-query`: интервал, добавленный вручную с помощью диспетчера контекста `tracer.start_span("storage-query")`.
- `ZVREVRANGE`: интервал, добавленный автоматически интеграцией `redis_opentracing`.

Каждая трассировка включает точный хронометраж и дополнительные теги: IP-адрес, версию библиотеки или данные, специфические для того или иного интервала (инструкция базы данных, код ответа HTTP и т. д.). Эти элементы крайне полезны для того, чтобы выявлять проблемы с быстродействием и понимать закономерности взаимодействия между компонентами распределенной системы.

Итоги

Мы рассмотрели три основных механизма наблюдаемости современных приложений: журналирование, сбор метрик и распределенную трассировку. У каждого механизма есть свои достоинства, хотя журналирование, безусловно, самый важный способ сбора информации от вашего приложения: оно устроено просто, не требует специальной инфраструктуры (хотя иметь ее полезно) и реже всего подводит вас.

Тем не менее у журналирования есть свои ограничения. Когда сообщения журнала не структурированы, становится сложнее извлекать из журналов полезную информацию. Кроме того, журналы не подходят, чтобы периодически запрашивать информацию об использовании ресурсов и быстродействии. Они хороши для целей аудита и анализа после серьезных сбоев, но редко помогают отслеживать текущую информацию и реагировать на впезанные события.

Поэтому системы сбора метрик становятся естественным и ценным расширением инфраструктур журналирования. Они позволяют собирать информацию в реальном времени, создавать нестандартные метрики и строить дашборды, которыми будут активно пользоваться специалисты по эксплуатации. Также в определенной степени системы сбора метрик можно применять для обзора бизнес-метрик, хотя большие организации обычно предпочитают специализированное аналитическое ПО. Системы сбора метрик незаменимы для отслеживания быстродействия приложений.

Наконец, инфраструктуры распределенной трассировки отлично работают в сложных системах, где много компонентов взаимодействуют друг с другом. Такие инфраструктуры незаменимы в ситуациях, когда проблемы возникают

на стыках разных взаимодействующих служб. Они предоставляют исключительно полезные сведения о потоке информации в системе и упрощают поиск аномалий при взаимодействиях.

Решения наблюдаемости каждого вида дают полезное представление о том, что происходит в системе, но какой-либо один вариант не решит всех ваших проблем. Лучше всего было бы иметь их все, но, к сожалению, не каждая организация может позволить себе разработать или приобрести все эти решения. Если ваши ресурсы ограниченны и необходимо выбрать что-то одно, стоит начать с журналирования, потому что оно обычно окупает затраты эффективнее всего.

Когда у вас появится эффективное решение из области наблюдаемости, вы начнете обнаруживать нетривиальные проблемы и дефекты производительности. Некоторые из них вас по-настоящему удивят. В конце концов, любой продукт лучше всего оценивать, запуская его в целевой среде под реальной нагрузкой. Обычно вы будете замечать, что большинство проблем быстродействия сосредоточено в изолированных «горячих точках». После того как эти проблемы будут выявлены, наступит время подробного анализа быстродействия и оптимизации — темы следующей главы.

13

Оптимизация кода

Оптимизация кода — это деятельность, которая заставляет приложение работать более эффективно, но обычно не влияет на его функциональность. Чаще всего оптимизируется скорость выполнения (то есть процессорное время), но оптимизация может также минимизировать потребление различных ресурсов: памяти, дискового пространства, пропускной способности сети и т. д.

В предыдущей главе вы узнали о различных методах наблюдаемости, с помощью которых можно выявлять узкие места в быстродействии приложения. Журналирование, отслеживание метрик и трассировку можно использовать, чтобы сформировать общую картину быстродействия и расставить приоритеты для оптимизации. Кроме того, специалисты по эксплуатации часто используют нестандартные уведомления для метрик быстродействия и использования ресурсов или регистрируют сообщения об истекшем времени ожидания, чтобы выявить компоненты, которые требуют немедленной реакции.

Но даже лучшие системы журналирования, сбора метрик и трассировки дают лишь отдаленное представление о проблеме быстродействия. Чтобы исправить ее, вам придется провести тщательное профилирование, которое выявит детализированные закономерности использования ресурсов.

Есть много разных приемов оптимизации. Одни из них направлены на алгоритмическую сложность и использование оптимизированных структур данных. Другие жертвуют точностью или целостностью результатов ради того, чтобы повысить быстродействие быстро и просто. Только когда вы поймете, как работает ваше приложение и как оно использует ресурсы, можно будет применить подходящий метод оптимизации.

В этой главе рассматриваются следующие темы:

- Типичные причины плохого быстродействия.
- Профилирование кода.
- Снижение сложности за счет выбора подходящих структур данных.
- Архитектурные компромиссы.

В Python есть встроенные средства профилирования и полезные возможности оптимизации, так что теоретически можно не пользоваться ничем другим, кроме Python и его стандартной библиотеки. Однако дополнительные инструменты можно считать своего рода метаоптимизацией: они помогают вам более эффективно находить и решать проблемы быстродействия, поэтому мы будем их интенсивно использовать.

Технические требования

Ниже перечислены пакеты Python, используемые в этой главе, которые можно загрузить из PyPI:

- `gprof2dot`
- `objgraph`
- `pymemcache`

Информация о том, как устанавливать пакеты, приведена в главе 2 «Современные среды разработки для Python». Возможно, вам также стоит установить пару дополнительных приложений:

- Graphviz: система визуализации диаграмм с открытым кодом, доступная по адресу https://graphviz.org.
- Memcached: служба кэширования с открытым кодом, доступная по адресу https://memcached.org.

Файлы с кодом этой главы доступны по адресу https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2013.

Типичные причины плохого быстродействия

Прежде чем углубляться в тонкости профилирования, обсудим типичные причины плохого быстродействия в приложениях.

На практике для многих систем характерны одни и те же проблемы с быстродействием. Важно знать их потенциальные причины, чтобы выбрать правильную

стратегию профилирования. Кроме того, некоторые специфические причины проявляются в узнаваемых закономерностях. Если вы знаете, что искать, то сможете исправить очевидные проблемы быстродействия без подробного анализа и сэкономите много времени на последующей оптимизации.

Наиболее распространенные причины плохого быстродействия приложений таковы:

- избыточная сложность;
- избыточное выделение ресурсов и утечка ресурсов;
- избыточный ввод/вывод и блокирующие операции.

Больше всего проблем среди перечисленных факторов создает избыточная сложность, поэтому начнем именно с нее.

Сложность кодо

Первое и самое очевидное, на что нужно обращать внимание при попытке улучшить быстродействие приложения, — сложность кода. Есть много определений того, что же такое сложность, и ее можно выразить многими способами. Некоторые измеряемые метрики сложности могут предоставить объективную информацию о поведении кода, и ее часто можно экстраполировать на ожидаемое быстродействие. Опытные программисты даже могут обоснованно предположить, какая из двух разных реализаций лучше поведет себя на практике, опираясь на их сложность и контекст выполнения.

Назовем две самые популярные характеристики сложности приложений:

- **Цикломатическая сложность**, которая часто бывает тесно связана с быстродействием приложения.
- **Нотация «О-большое»**, также называемая записью Ландау, — способ классификации алгоритмов, который помогает объективно оценить быстродействие кода.

Таким образом, оптимизацию иногда можно понимать как уменьшение сложности. В следующих разделах мы поближе познакомимся с этими двумя разновидностями сложности кода.

Цикломатическая сложность

Цикломатическая сложность — метрика, которую разработал Томас Дж. Маккейб в 1976 году; по фамилии автора она также иногда называется **сложностью Маккейба**. Цикломатическая сложность равна количеству линейно независи-

мых маршрутов через фрагмент кода. В двух словах, все точки ветвления (команды `if`) и циклы (команды `for` и `while`) повышают сложность кода.

В зависимости от значения цикломатической сложности код относится к тому или иному классу сложности. Часто встречается такая шкала классов:

Цикломатическая сложность	Класс сложности
от 1 до 10	Несложный
от 11 до 20	Умеренно сложный
от 21 до 50	Очень сложный
свыше 50	Слишком сложный

Цикломатическая сложность обычно обратно пропорциональна быстродействию приложения (чем выше сложность, тем ниже быстродействие). Тем не менее это скорее оценка качества кода, нежели метрика быстродействия. Она не отменяет необходимости профилировать код, выявляя узкие места быстродействия. В коде с высокой цикломатической сложностью часто используются довольно замысловатые алгоритмы, которые медленно работают при большом объеме входных данных.

Хотя цикломатическая сложность не позволяет надежно судить о быстродействии приложения, у нее есть одно важное достоинство: это метрика исходного кода, которую можно вычислить с помощью соответствующих инструментов. Этого нельзя сказать о других канонических характеристиках сложности, включая запись «О-большое». Будучи объективно измеримой, цикломатическая сложность может стать полезным дополнением к профилированию, потому что она предоставляет полезную информацию о проблемных частях вашего ПО. Сложные части кода — первое, что следует проанализировать, если вы планируете радикально переработать архитектуру кода.

Измерить цикломатическую сложность в Python относительно просто, потому что она выводится из абстрактного синтаксического дерева. Конечно, вам не придется делать это вручную. Популярный пакет `mccabe` из PyPI способен автоматически анализировать сложность исходного кода на Python. Он также доступен в виде плагина Pytest под именем `pytest-mccabe`, поэтому его легко включить в автоматизированные процессы контроля качества и тестирования (см. главу 10).

О пакете `mccabe` можно больше узнать по адресу https://pypi.org/project/mccabe/.

Пакет `pytest-mccabe` доступен по адресу https://pypi.org/project/pytest-mccabe/.

Нотация «О-большое»

Каноническим способом определения сложности функций считается нотация «О-большое». Эта метрика оценивает, как ведет себя алгоритм нри увеличении размера ввода. Например, увеличивается время вынолнения алгоритма в линейной или квадратичной зависимости от размера ввода?

Вручную оценивать «О-большое» для алгоритма лучше всего тогда, когда вы пытаетесь в общих чертах понять, как быстродействие связано с размером входпых данных. Зная сложность компонентов приложения, легче обнаруживать аспекты, которые существенно замедляют код, и сосредоточиваться на них.

Когда измеряется «О-большое», из зависимости исключаются все константы и составляющие пизших порядков, чтобы установить, к чему она стремится, если объем входных данпых становится очень большим. Идея заключается в том, чтобы отнести алгоритм к одному из известных классов сложпости (нусть даже приближепно). Ниже перечислены самые распространенные классы сложности, где n — количество входных элементов задачи:

Обозпачеиие	Тип
O(1)	Постоянная; не зависит от размера входных данных
O(n)	Линейная; растет прямо пропорционально n
O($n \log n$)	Квазилинейная
O(n^2)	Квадратичная
O(n^3)	Кубическая
O($n!$)	Факториальная

Понять классы сложности будет проще, если сравнить основные операции доступа к элементам некоторых стандартных коллекций Python:

- Поиск в списке Python по индексу имеет сложность O(1). Списки Python — это массивы переменного размера с нредварительно выделепной памятью, сходные с контейпером `Vector` из стандартной библиотеки C++. Позиция элемепта списка в памяти известна заранее, ноэтому время обращения к любому элементу будет одним и тем же.
- Определение индекса элемента по его значепию в списке имеет сложность O(n). При использовании метода `list.index(value)` Python неребирает элементы списка, пока не найдет элемент, совпадающий со значением выражения `value`. В худшем случае придется перебрать все элементы списка; следовательно, будут выполпены n операций, где n — длина списка. Однако

в среднем поиск случайного элемента потребует $n/2$ операций, так что сложность равна $O(n)$.

- Поиск в словаре по ключу имеет сложность $O(1)$. В Python любое неизменяемое значение можно использовать в качестве ключа в словаре. Значения ключей не преобразуются напрямую в адреса памяти, но в Python используется эффективное хеширование, которое гарантирует, что операция поиска по ключу в среднем выполняется с постоянной сложностью.

Следующий пример поможет понять, как определить сложность функции:

```
def function(n):
    for i in range(n):
        print(i)
```

В этом фрагменте функция `print()` будет выполнена `n` раз. Длина цикла находится в линейной зависимости от n, так что сложность всей функции составит $O(n)$.

Если в функции есть условия, то сложность определяется по худшему случаю. Рассмотрим следующий пример:

```
def function(n, print_count=False):
    if print_count:
        print(f'count: {n}')
    else:
        for i in range(n):
            print(i)
```

Здесь сложность функции может быть $O(1)$ или $O(n)$ в зависимости от значения аргумента `print_count`. Худшим случаем является $O(n)$, так что это и есть искомая сложность.

Оценивая сложность, не всегда приходится учитывать худший случай. Быстродействие многих алгоритмов изменяется в зависимости от статистических характеристик входных данных. Иногда затраты на обработку худших случаев удается амортизировать с помощью разных трюков. Поэтому во многих ситуациях лучше рассматривать среднюю или амортизированную сложность реализации.

Для примера рассмотрим операцию присоединения одного элемента к экземпляру типа `list` в Python. Мы знаем, что `list` в CPython использует не связанные списки, а массив с опережающим выделением памяти. Если массив уже заполнен, то для присоединения нового элемента потребуется выделить новый массив и скопировать все существующие элементы (ссылки) в новую область памяти. Если взглянуть на это с точки зрения сложности худшего случая, становится ясно, что метод `list.append()` имеет сложность $O(n)$, относительно высокую по сравнению с типичной реализацией структуры свя-

занного списка. Однако мы также знаем, что реализация типа `list` в CPython использует опережающее выделение памяти (выделяется больше памяти, чем требуется в текущий момент), чтобы умепьшить сложность периодических выделений памяти. Оценивая сложность по серии операций, можно заметить, что средняя сложность `list.append()` равна O(1), и это отличный результат.

Всегда держите в голове характеристику «О-большое», но избегайте лишпего догматизма. Нотация «О-большое» является асимптотической; это означает, что она предназначена для анализа предельного поведения функции, когда объем входных данных стремится к бесконечности. Следовательно, она не обязательно обеспечивает надежную приближенную оценку быстродействия для реальных данных. Асимптотическая запись прекрасно подходит для того, чтобы определить скорость роста функции, но она не дает прямого ответа на простой вопрос: какая реализация займет меньше времени? Сложность худшего случая игнорирует все нюансы продолжительности отдельных операций, чтобы показать, как программа ведет себя асимптотически. Этот тип сложности актуален для очень больших входных данпых, которые не всегда нужно учитывать при анализе.

Допустим, в вашей задаче нужно обработать n независимых элементов. У вас есть две программы — A и B. Вы знаете, что программа A решает задачу за $100n^2$ операций, а программа B — за $5n^3$ операций. Какую реализацию вы выберете?

При очень больших входных данных программа A подходит лучше, потому что она лучше ведет себя асимптотически. Она обладает сложностью $O(n^2)$ по сравнению со сложностью $O(n^3)$ программы B. Однако решение простого неравенства $100n^2 > 5n^3$ показывает, что при $n < 20$ программа B выполняет меньше операций. Таким образом, если чуть больше знать об ограничениях ввода, это позволит принимать более обоснованные решения. Кроме того, нельзя предполагать, что отдельные операции в обеих программах занимают одинаковое время. Если вы хотите знать, какая программа будет выполняться быстрее для входных данных заданного объема, придется профилировать оба приложения.

Избыточное выделение ресурсов и утечки

С увеличением сложности часто возникает проблема избыточного выделения ресурсов. Сложный (алгоритмически или асимптотически) код с большей вероятностью будет включать неэффективные структуры данных или выделять слишком много ресурсов, не освобождая их впоследствии.

Хотя неэффективное использование ресурсов часто идет рука об руку со сложностью, избыточное выделение ресурсов следует рассматривать как отдельную проблему быстродействия. Это объясняется двумя причинами:

- **Выделеине (и освобождеиие) ресурсов заннмает время.** Это относится и к оперативной памяти, которая считается быстрой (по сравнению с дисками). Иногда лучше поддерживать пул ресурсов, вместо того чтобы постоянно выделять и освобождать их.
- **Маловероятно, что ваше приложение будет работать в одиночестве.** Если оно выделяет слишком много ресурсов, это может обездолить другие программы на том же хосте. В крайних случаях избыточное использование ресурсов одним приложением может привести к тому, что вся система окажется недоступной или программа неожиданно завершится.

Если приложение захватывает все доступные ресурсы (будь то память, дисковое пространство, пропускная способность сети или любой другой ресурс), это отрицательно скажется на других программах, которые выполняются в той же среде.

Кроме того, многие операционные системы или среды позволяют запрашивать больше ресурсов, чем доступно физически. Типичный случай — перевыделение памяти, которое позволяет операционной системе выделять процессам больше памяти, чем установлено на хосте. Для этого используется виртуальная память. При нехватке физической памяти операционная система обычно временно выгружает неиспользуемые страницы памяти на диск, чтобы освободить место.

При нормальной нагрузке эти манипуляции могут остаться незаметными. Но если виртуальная память используется чересчур интенсивно, нормальная подкачка может перейти в «дергание», когда операционная система постоянно выгружает и загружает страницы с диска. Это может радикально снизить быстродействие.

Причиной избыточного использования ресурсов могут быть неэффективные структуры данных, выделение слишком большого объема памяти из пула ресурсов или непреднамеренные **утечки ресурсов**. Последние возникают тогда, когда приложение захватило какие-либо ресурсы, но не освобождает их, даже когда они больше не нужны. Утечки чаще всего связаны с выделением памяти, но могут происходить и с другими ресурсами (например, дескрипторами файлов или открытыми подключениями).

Избыточный ввод/вывод и блокирующие операции

При написании приложений мы часто забываем, что многие операции требуют записи и чтения данных с диска или из сетевого подключения, а на это нужно время. Операции ввода/вывода всегда ощутимо влияют на быстродействие

приложений, но многие программные архитекторы не учитывают их, когда проектируют сложные сетевые системы.

С появлением быстрых SSD-дисков дисковый ввод/вывод стал быстрее, чем когда-либо, но по скорости он все еще уступает оперативной памяти (а оперативная память уступает кэшу процессора). Кроме того, некоторые диски обеспечивают хорошую пропускную способность для больших файлов, но показывают неожиданно низкие результаты в режиме произвольного доступа.

Все сказанное относится и к сетевым подключениям. Скорость передачи данных не может превышать скорость света. Если процессы выполняются на двух хостах, которые находятся далеко друг от друга, каждое взаимодействие добавляет круговую задержку. Это традиционная головная боль распределенных систем, которые часто состоят из большого количества взаимодействующих сетевых служб.

Избыточные операции ввода/вывода тоже влияют на способность ПО одновременно обслуживать нескольких пользователей. Вы уже узнали о разных моделях конкурентного выполнения в главе 6 «Конкурентное выполнение». Даже при асинхронной модели, которая теоретически должна лучше всего работать с вводом/выводом, можно случайно задействовать синхронные операции ввода/вывода, которые сведут на нет все преимущества конкурентности. В такой ситуации снижение быстродействия может наблюдаться как падение максимальной пропускной способности обработки.

Как видите, у проблем с быстродействием может быть много причин. Чтобы исправить проблему, сначала необходимо выявить причину. Обычно для этого применяют процесс, который называется **профилированием**.

Профилирование кода

Зная, что может пойти не так, можно выдвигать гипотезы о том, почему происходят проблемы с быстродействием и как их исправить. Однако проверить эти гипотезы можно только профилированием. Как правило, лучше не пытаться оптимизировать приложение без предварительного профилирования.

Практический опыт полезен, так что нет ничего плохого в том, чтобы перед профилированием провести ревизию кода и немного поэкспериментировать. Кроме того, некоторые методы профилирования требуют включать дополнительный код или писать тесты быстродействия. Это означает, что часто код все равно приходится внимательно перечитывать. Если при этом проводить небольшие эксперименты (например, в форме сеансов отладки), можно выявить очевидные проблемы.

И все же не стоит полагаться на легкие пути. Удачное соотношение между свободными экспериментами и классическим профилированием составляет около 1 : 9. Я рекомендую выстраивать процесс профилирования и оптимизации по таким принципам:

1. **Решите, сколько времени вы готовы потратить.** Не каждая оптимизация возможна, и не каждую проблему можно исправить. Будьте готовы к тому, что первая попытка может оказаться неудачной. Заранее решите, когда остановиться. Лучше отступить и повторить попытку позднее, чем проиграть долгое сражение.
2. **Разбейте это время на сеансы.** Оптимизация похожа на отладку: она тоже требует сосредоточенности, порядка и ясности ума. Выберите продолжительность сеанса, которая позволит вам выполнить весь цикл: выдвижение гипотезы, профилирование и экспериментирование. Сеанс не должен превышать нескольких часов.
3. **Составьте графнк.** Проводите сеансы в то время, когда вас с наименьшей вероятностью могут прервать, а работа будет наиболее эффективной. Если задача велика, лучше планировать работу на несколько дней вперед, чтобы каждый день можно было начинать со свежими идеями.
4. **Не планнруйте другую разработку.** Если первый сеанс не приведет к успеху, вы, вероятно, будете постоянно думать о своей проблеме и вам будет трудно сосредоточиться на других серьезных задачах. Если вы трудитесь в команде, четко объяспите, что не сможете в полной мере участвовать в работе, пока не закроете тему. Заготовьте несколько небольших задач, которые помогут вам отвлечься между сеансами.

Вероятно, оптимизация потребует модифицировать код вашего приложения. Чтобы было меньше шансов что-нибудь испортить, убедитесь, что код хорошо покрыт тестами. Стандартные методы тестирования описаны в главе 10 «Автоматизация тестирования и контроля качества».

Главная составляющая каждого сеанса профилирования — гипотеза о том, что могло пойти не так. Она позволяет решить, какие методы профилирования использовать и какие инструменты будут лучше работать. Превосходным источником гипотез служат системы сбора метрик и распределенной трассировки, которые мы подробно рассмотрели в главе 12 «Наблюдение за поведением и быстродействием приложений».

Если вы понятия не имеете, в чем проблема, лучше всего начать с традиционного профилирования процессора, потому что оно с наибольшей вероятностью приведет вас к следующей реалистичпой гипотезе. Дело в том, что многие анти-

паттерны использования ресурсов и сети также могут проявиться в профильных данных процессора.

Профилирование загруженности процессора

Как правило, проблемы быстродействия возникают на небольшом участке кода, который существенно влияет на быстродействие приложения в целом. Такие участки называются **узкими местами**. Оптимизация заключается в том, чтобы выявить и исправить эти узкие места.

Главный источник узких мест — ваш код. Стандартная библиотека предоставляет все инструменты, которые необходимы для профилирования кода. Они работают на детерминированном принципе. **Детермиированный ирофилнровщик** измеряет время, проведенное в каждой функции; для этого он внедряет таймер на самом низком уровне. Это приводит к значительным дополнительным затратам ресурсов, но дает хорошее представление о том, где теряется время. С другой стороны, **статистический нрофилировщик** отслеживает использование указателя инструкций, не внедряя ничего в код. Этот вариант менее точен, зато позволяет выполнять целевую программу на полной скорости.

Существуют два способа профилирования кода:

- **Макропрофилировапие**: профилирует всю программу во время ее использования и генерирует статистику.
- **Микронрофилированне**: выполняет измерения в конкретной части программы, для чего она вручную снабжается инструментами профилирования.

Если неизвестно, какой именно фрагмент кода выполняется медленно, мы обычно начинаем с макронрофилирования всего приложения, чтобы получить общий профиль быстродействия, а затем онределяем, какие компоненты оказались узкими местами.

Макропрофилирование

Для макропрофилирования приложение запускается в особом режиме, где интерпретатор оснащается специальными инструментами для сбора статистики, которая относится к использованию кода. Стандартная библиотека Python предоставляет ряд инструментов для этой цели, включая следующие модули:

- `profile`: чистая реализация на Python.
- `cProfile`: реализация на C с таким же интерфейсом, как у `profile`, но с меньшими накладными расходами.

В большинстве случаев программистам на Python рекомендуется использовать `cProfile`, который расходует меньше ресурсов. Но если вам нужно каким-то образом расширить профилировщик, то лучше подойдет `profile`: он не использует расширения на C и поэтому проще расширяется.

У обеих систем похожие интерфейсы и принципы использования, поэтому здесь мы рассмотрим только одну из них. Ниже приведен модуль myapp.py с функцией `main`, которая будет профилироваться модулем `cProfile`:

```
import time

def medium():
    time.sleep(0.01)

def light():
    time.sleep(0.001)

def heavy():
    for i in range(100):
        light()
        medium()
        medium()
    time.sleep(2)

def main():
    for i in range(2):
        heavy()

if __name__ == '__main__':
    main()
```

Этот модуль можно вызвать напрямую из командной строки, чтобы получить сводную информацию:

```
$ python3 -m cProfile myapp.py
```

Результаты профилирования для сценария myapp.py могут выглядеть так:

```
         1208 function calls in 8.243 seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.001    0.000    8.243    4.121 myapp.py:13(heavy)
        1    0.000    0.000    8.243    8.243 myapp.py:2(<module>)
        1    0.000    0.000    8.243    8.243 myapp.py:21(main)
      400    0.001    0.000    4.026    0.010 myapp.py:5(medium)
      200    0.000    0.000    0.212    0.001 myapp.py:9(light)
        1    0.000    0.000    8.243    8.243 {built-in method exec}
      602    8.241    0.014    8.241    0.014 {built-in method sleep}
```

Столбцы в этом выводе имеют такой смысл:

- `ncalls`: общее количество вызовов.
- `tottime`: суммарное время, проведенное в функции, за исключением времени, которое проведено в вызовах подфункций.
- `cumtime`: суммарное время, проведенное в функции, включая время, которое проведено в вызовах подфункций.

Значение в столбце `percall` справа от `tottime` равно `tottime / ncalls`, а в столбце `percall` справа от `cumtime` — `cumtime / ncalls`.

Эти метрики — текстовое представление объекта статистики, созданного профилировщиком. Этот объект также можно создать и просмотреть в интерактивном сеансе Python:

```
>>> import cProfile
>>> from myapp import main
>>> profiler = cProfile.Profile()
>>> profiler.runcall(main)
>>> profiler.print_stats()
         1206 function calls in 8.243 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall file:lineno(function)
        2    0.001    0.000    8.243    4.121 myapp.py:13(heavy)
        1    0.000    0.000    8.243    8.243 myapp.py:21(main)
      400    0.001    0.000    4.026    0.010 myapp.py:5(medium)
      200    0.000    0.000    0.212    0.001 myapp.py:9(light)
      602    8.241    0.014    8.241    0.014 {built-in method sleep}
```

Статистику можно сохранить в файле и затем прочитать модулем `pstats`. Этот модуль предоставляет класс, который умеет обрабатывать файлы профилей, а также предоставляет ряд вспомогательных инструментов, позволяющих с большим удобством просматривать результаты профилирования. Следующий фрагмент показывает, как узнать общее количество вызовов и вывести первые три вызова, отсортированные по метрике `time`:

```
>>> import pstats
>>> import cProfile
>>> from myapp import main
>>> cProfile.run('main()', 'myapp.stats')
>>> stats = pstats.Stats('myapp.stats')
>>> stats.total_calls
1208
>>> stats.sort_stats('time').print_stats(3)
Mon Apr  4 21:44:36 2016    myapp.stats
```

```
         1208 function calls in 8.243 seconds

   Ordered by: internal time
   List reduced from 8 to 3 due to restriction <3>

   ncalls  tottime  percall  cumtime  percall file:lineno(function)
      602    8.241    0.014    8.241    0.014 {built-in method sleep}
      400    0.001    0.000    4.025    0.010 myapp.py:5(medium)
        2    0.001    0.000    8.243    4.121 myapp.py:13(heavy)
```

После этого можно обойти код, выводя вызывающую и вызываемую сторону для каждой функции:

```
>>> stats.print_callees('medium')
   Ordered by: internal time
   List reduced from 8 to 1 due to restriction <'medium'>

Function           called...
                   ncalls  tottime  cumtime
myapp.py:5(medium) ->  400    4.025    4.025  {built-in method sleep}

>>> stats.print_callees('light')
   Ordered by: internal time
   List reduced from 8 to 1 due to restriction <'light'>

Function           called...
                    ncalls  tottime  cumtime
myapp.py:9(light)  ->  200    0.212    0.212  {built-in method sleep}
```

Возможность сортировать вывод позволяет работать с разными представлениями, чтобы находить узкие места. Для примера рассмотрим следующие сценарии:

- Если количество небольших вызовов (низкое значение `percall` для столбца `tottime`) велико (высокое значение `ncalls`) и занимает бóльшую часть глобального времени, то функция или метод, вероятно, выполняется в очень длинном цикле. Часто оптимизация может заключаться в том, чтобы переместить этот вызов в другую область видимости и таким образом сократить число операций.
- Если один вызов функции занимает очень много времени, нонробуйте кэшировапие (если оно возможно).

Узкие места также можно наглядно представить в виде диаграмм (рис. 13.1). Чтобы преобразовать данные профилировщика в точечный график, можно воспользоваться сценарием gprof2dot.py:

```
$ gprof2dot.py -f pstats myapp.stats | dot -Tpng -o output.png
```

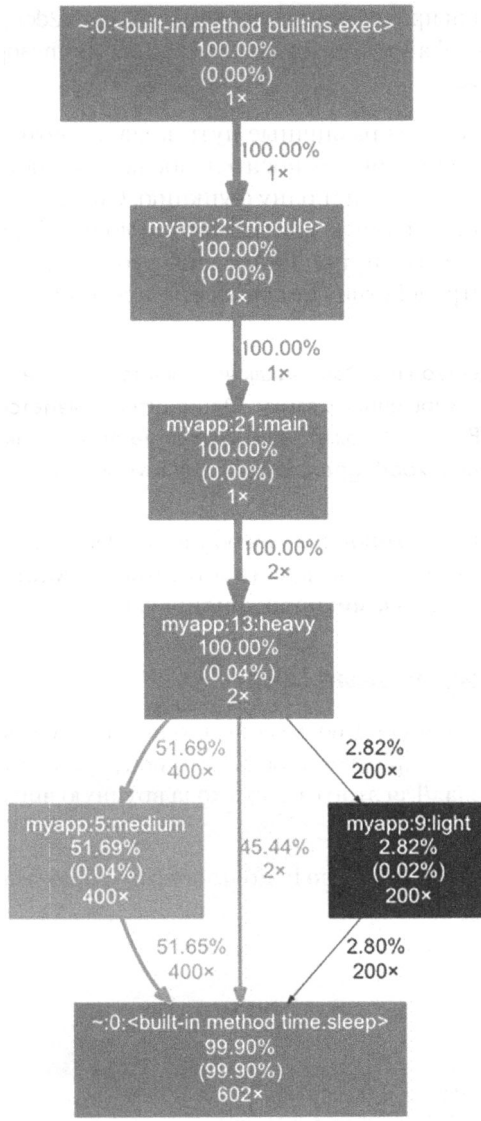

Рис. 13.1. Пример диаграммы со сводкой профилирования, сгенерированной сценарием gprof2dot

Сценарий gprof2dot.py — часть пакета `gprof2dot`, доступного в PyPI. Его можно загрузить с помощью `pip`. Для работы с этим пакетом необходимо установить программу Graphviz, которую можно бесплатно загрузить по адресу http://www.graphviz.org/.

Вот пример вывода для приведенного выше вызова gprof2dot.py в оболочке Linux, который преобразует файл данных профилирования myapp.stats в диаграмму в формате PNG:

На диаграмме изображены различные нути кода, по которым проходит программа во время выполнения, и относительное время, проведенное на каждом пути. Каждый блок представляет одну функцию. Связывая функции, вы получаете количество выполнений заданного пути и долю общего времени выполнения, проведенную па этих путях. Диаграммы — отличный способ исследовать закономерности быстродействия больших приложений.

Преимущество gprof2dot заключается в том, что он пытается быть независимым по отношению к языку. Он не ограничивается выводом profile или cProfile в Python, а может читать и другие профильные данные, например perf для Linux, xperf, gprof, HPROF для Java и многие другие.

Макропрофилирование — хороший способ узнать, в какой функции происходит проблема (или хотя бы очертить окрестность этой функции). Когда вы ее обнаружите, можно переходить к микропрофилированию.

Микропрофилирование

Обнаружив медленную функцию, мы обычно нереключаемся па микропрофилирование, чтобы сгенерировать профиль, ориентированный на наименьший возможный объем кода. Для этого в часть кода вручную внедряется специально созданный тест скорости.

Например, модуль `cProfile` можно использовать в форме декоратора, как в следующем нримере:

```
import time
import tempfile
import cProfile
import pstats

def profile(column='time', list=3):
    def parametrized_decorator(function):
        def decorated(*args, **kw):
            s = tempfile.mktemp()

            profiler = cProfile.Profile()
            profiler.runcall(function, *args, **kw)
            profiler.dump_stats(s)

            p = pstats.Stats(s)
            print("=" * 5, f"{function.__name__}() profile", "=" * 5)
```

```
            p.sort_stats(column).print_stats(list)
        return decorated
    return parametrized_decorator

def medium():
    time.sleep(0.01)

@profile('time')
def heavy():
    for i in range(100):
        medium()
        medium()
    time.sleep(2)

def main():
    for i in range(2):
        heavy()

if __name__ == '__main__':
    main()
```

Этот подход позволяет тестировать отдельные части приложения (в данном случае функцию `heavy()`) и получать более актуальную статистику. Вы сможете собрать много изолированных, точно направленных профильных данных при одном запуске приложения. Вот как может выглядеть вывод, если запустить приведенный выше код в интерпретаторе Python:

```
===== heavy() profile =====
Wed Apr 10 03:11:53 2019 /var/folders/jy/wy13kx0s7sb1dx2rfsqdvzdw0000gq
/T/tmpyi2wejm5

         403 function calls in 4.330 seconds

 Ordered by: internal time
 List reduced from 4 to 3 due to restriction <3>

 ncalls tottime percall cumtime percall filename:lineno(function)
    201 4.327   0.022   4.327   0.022   {built-in method time.sleep}
    200 0.002   0.000   2.326   0.012   cprofile_decorator.
py:24(medium)
      1 0.001   0.001   4.330   4.330   cprofile_decorator.py:28(heavy)

===== heavy() profile =====
Wed Apr 10 03:11:57 2019 /var/folders/jy/wy13kx0s7sb1dx2rfsqdvzdw0000gq
/T/tmp8mubgwjw
         403 function calls in 4.328 seconds

 Ordered by: internal time
 List reduced from 4 to 3 due to restriction <3>
```

```
  ncalls tottime percall cumtime percall filename:lineno(function)
     201 4.324   0.022   4.324   0.022   {built-in method time.sleep}
     200 0.002   0.000   2.325   0.012   cprofile_decorator.
py:24(medium)
       1 0.001   0.001   4.328   4.328.  cprofile_decorator.py:28(heavy)
```

Из этих результатов видно, что функция `heavy()` вызывалась ровно два раза, и оба раза получались очень похожие профили. В списке вызовов отражен 201 вызов функции `time.sleep()`, выполнение которых в сумме заняло 4.3 секунды.

Иногда на этой стадии одного лишь профиля со списком вызываемых функций недостаточно, чтобы понять проблему. Распространенный подход состоит в том, чтобы создавать альтернативные реализации конкретных частей кода и измерять продолжительность их выполнения. Если бы функция `heavy()` делала что-то полезное, то можно было бы, например, написать для этой же задачи код с более низкой сложностью.

`timeit` — полезный модуль, который предоставляет простой способ измерять время выполнения небольшого фрагмента кода, используя лучший таймер, который предоставляется системой (объект `time.perf_counter()`). Рассмотрим пример:

```
>>> from myapp import medium
>>> import timeit
>>> timeit.timeit(light, number=1000)
1.2880568829999675
```

Функция `timeit.timeit()` выполняет заданную функцию 1000 раз (количество задается параметром `number`) и возвращает суммарное время, затраченное на все выполнения. Если вы ожидаете значительного разброса результатов, используйте `timeit.repeat()`, чтобы повторить тест заданное количество раз:

```
>>> timeit.repeat(light, repeat=5, number=1000)
[1.283251813999982, 1.2764048459999913, 1.2787090969999326,
1.279601415000002, 1.2796783549999873]
```

Модуль `timeit` позволяет повторить вызов многократно, и с его помощью можно легко измерить время выполнения изолированных фрагментов кода. Это очень удобно за пределами контекста приложения (например, в командной строке), но для существующего приложения такое решение не идеально.

С помощью модуля `timeit` можно создавать тесты быстродействия в вашем фреймворке автоматизированного тестирования, но это требует осторожности. Результаты хронометража могут изменяться от случая к случаю. Точность результатов можно повысить, если повторять один тест несколько раз и вычислять

среднее значение. Более того, некоторые процессоры могут изменять тактовую частоту при разной нагрузке или температуре. Таким образом, вы будете получать разные результаты в зависимости от того, занят компьютер или простаивает. Другие параллельно выполняемые программы тоже могут повлиять на общую картину хронометража. В результате для небольших фрагментов кода имеет смысл многократно повторять тест. Поэтому, измеряя время в процессах автоматизированного тестирования, мы обычно стараемся выявлять закономерности, а не устанавливать конкретные пороговые значения времени как утверждения.

При профилировании процессора часто удается обнаружить различные закономерности, связанные с захватом и освобождением ресурсов, потому что эти операции нередко осуществляются вызовом функций. Традиционное профилирование Python с модулями `profile` и `cProfile` может отразить общую картину закономерностей использования ресурсов, однако если речь идет о памяти, мы предпочитаем специализированные решения для ее профилирования.

Профилирование использования памяти

Прежде чем браться за выявление проблем с памятью в Python, следует знать, что утечки намяти в Python имеют особую природу. В некоторых компилируемых языках (таких, как C и C++) утечки намяти почти всегда возникают из-за выделенных блоков памяти, на которые не ссылается ни один указатель. Если у вас нет ссылки на намять, ее нельзя освободить. Именно эта ситуация называется **утечкой памяти**.

В Python низкоуровневое управление памятью недоступно для пользователя, зато возникает проблема ссылок на объекты, которые уже не используются в программе, но не были удалены. Интерпретатор не может освободить такие ресурсы, но ситуация отличается от классических утечек памяти в языке C.

Всегда существует исключительный случай утечки памяти через указатели в расширениях Python на C, но это совершенно другие ошибки, для диагностики которых нужны совершенно другие инструменты. Их не удастся легко проанализировать из кода Python. Стандартный инструмент для анализа утечек памяти в C — Valgrind. За дополнительной информацией о Valgrind обращайтесь по адресу https://valgrind.org/.

Итак, утечки памяти в Python в основном возникают из-за непредвиденных или незапланированных схем захвата ресурсов. Они крайне редко оказываются следствием реальных ошибок, связанных с некорректными операциями выделения и освобождения памяти. Такие операции доступны для разработчиков

только в CPython при написании расширений на C с Python/C API. Вам практически никогда не придется иметь с ними дела. А значит, большинство утечек памяти в Python возникает из-за чрезмерной сложности кода и неочевидного взаимодействия между компонентами, которое трудно отследить. Чтобы обнаруживать подобные дефекты в вашем коде, необходимо знать, как на самом деле используется память в программах.

Оказывается, не очень просто получить информацию о количестве объектов, которые находятся под управлением интерпретатора Python, и проанализировать их размер. Например, чтобы узнать, сколько памяти в байтах занимает конкретный объект, необходимо обойти все его атрибуты, распутать перекрестные ссылки, а затем все просуммировать. Это нетривиальная задача, если учесть, как объекты ссылаются друг на друга. Во встроенном модуле `gc`, который предоставляет интерфейс к сборщику мусора Python, нет высокоуровневых функций для этой цели. Чтобы получить полную информацию, код Python нужно было бы компилировать в отладочном режиме.

Часто программисты просто запрашивают у системы данные о том, сколько памяти приложение использует до и после некоторой операции. Однако эта характеристика получается приближенной и сильно зависит от управления памятью на уровне системы. Например, команда `top` в Linux или Диспетчер задач в Windows позволяют выявлять самые очевидные проблемы с памятью. Однако этот подход весьма трудоемок и с ним бывает трудпо разобраться, какой блок кода вызывает проблемы.

К счастью, существуют инструменты, которые позволяют создавать снимки памяти и вычислять количество и размер загруженных объектов. Однако следует помнить, что Python неохотно освобождает память и предпочитает удерживать ее на случай, если она понадобится снова.

Когда-то одним из самых нопулярных инструментов для отладки проблем с памятью в Python был пакет Guppy-PE и его компонент `Heapy`. К сожалению, оп больше не сопровождается и не поддерживает Python 3. Однако есть альтернативы, которые до определепной степени совместимы с Python 3:

- Memprof (http://jmdana.github.io/memprof/): по заявлениям разработчиков, программа должна работать в некоторых POSIX-совместимых системах (macOS и Linux). Последнее обновление вышло в декабре 2019 года.
- memory_profiler (https://pypi.org/project/memory-profiler).
- Pympler (https://pypi.org/project/Pympler/).
- objgraph (https://mg.pov.lt/objgraph/).

Последние три программы позиционируются как не зависящие от ОС и в настоящее время активно сопровождаются.

Учтите, что приведенная информация о совместимости основана исключительно на классификаторах, которые используются новейшими выпусками упомянутых пакетов, заявлениях в документации и анализе определений сборки проектов на момент написания книги. Ситуация может измениться к тому моменту, когда вы будете ее читать.

Как видите, разработчикам на Python доступно множество инструментов профилирования памяти, и у каждого из них есть свои ограничения. В этой главе мы сосредоточимся на профилировщике, который заведомо работает с последней версией Python (то есть Python 3.9) в разных операционных системах. Он называется `objgraph`.

API профилировщика `objgraph` может показаться немного неуклюжим, и его функциональность крайне ограниченна. Но он работает, очень хорошо делает то, что должен делать, и предельно прост в использовании. Измерения затрат памяти обычно не включаются в окончательную версию кода, так что инструмент не обязан быть красивым.

Модуль objgraph

`objgraph` — простой модуль для создания диаграмм ссылок на объекты в приложениях. С помощью таких диаграмм чрезвычайно удобно искать утечки памяти в Python.

Модуль `objgraph` доступен в PyPI, но не является полностью автономным. Чтобы строить диаграммы использования памяти, ему необходима программа Graphviz, которую мы установили в разделе «Макропрофилирование».

`objgraph` предоставляет ряд средств для вывода различной статистики, которая относится к использованию памяти и счетчикам объектов. Пример использования `objgraph` приведен в следующем фрагменте сеанса интерпретатора:

```
>>> import objgraph
>>> objgraph.show_most_common_types()
function                   1910
dict                       1003
wrapper_descriptor         989
tuple                      837
weakref                    742
method_descriptor          683
builtin_function_or_method 666
getset_descriptor          338
set                        323
```

```
member_descriptor          305
>>> objgraph.count('list')
266
>>> objgraph.typestats(objgraph.get_leaking_objects())
{'Gt': 1, 'AugLoad': 1, 'GtE': 1, 'Pow': 1, 'tuple': 2, 'AugStore':
1, 'Store': 1, 'Or': 1, 'IsNot': 1, 'RecursionError': 1, 'Div': 1,
'LShift': 1, 'Mod': 1, 'Add': 1, 'Invert': 1, 'weakref': 1, 'Not': 1,
'Sub': 1, 'In': 1, 'NotIn': 1, 'Load': 1, 'NotEq': 1, 'BitAnd': 1,
'FloorDiv': 1, 'Is': 1, 'RShift': 1, 'MatMult': 1, 'Eq': 1, 'Lt': 1,
'dict': 341, 'list': 7, 'Param': 1, 'USub': 1, 'BitOr': 1, 'BitXor': 1,
'And': 1, 'Del': 1, 'UAdd': 1, 'Mult': 1, 'LtE': 1}
```

Обратите внимание: количество созданных объектов, которое выводит `objgraph`, уже велико из-за того, что многие встроенные функции и типы Python — это обычные объекты Python, которые существуют в той же намяти процесса. Кроме того, сам модуль `objgraph` создает некоторые объекты, которые включены в эту сводку.

Как упоминалось ранее, `objgraph` позволяет создавать диаграммы закономерностей иснользования памяти и перекрестных ссылок, которые связывают все объекты в заданном нространстве имен. Самые нолезные функции модуля `objgraph` — `objgraph.show_refs()` и `objgraph.show_backrefs()`. Обе функции получают ссылку на анализируемый объект и сохраняют изображение диаграммы в файле с помощью пакета Graphviz. Примеры таких диаграмм изображены на рис. 13.2 и 13.3. Для их создания иснользовался следующий код:

```
from collections import Counter
import objgraph

def graph_references(*objects):
    objgraph.show_refs(
        objects,
        filename='show_refs.png',
        refcounts=True,
        # Дополнительная фильтрация ради краткости
        too_many=5,
        filter=lambda x: not isinstance(x, dict),
    )
    objgraph.show_backrefs(
        objects,
        filename='show_backrefs.png',
        refcounts=True
    )

if __name__ == "__main__":
    quote = """
    People who think they know everything are a
    great annoyance to those of us who do.
    """
```

```
words = quote.lower().strip().split()
counts = Counter(words)
graph_references(words, quote, counts)
```

Если пакет Graphviz не установлен, `objgraph` будет выводить диаграммы в формате DOT — на специальном языке описания графов.

На следующей диаграмме нредставлены все ссылки, которые хранятся в объектах `words`, `quote` и `counts` (рис. 13.2).

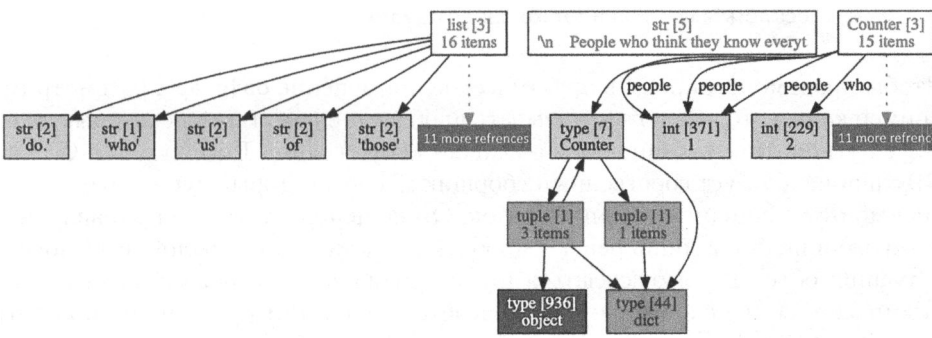

Рис. 13.2. Результат вызова show_refs() из функции graph_references()

Как видите, объект `words` (обозначенный как `list [3]`) хранит ссылки на 16 объектов, а объект `counts` (обозначенный как `Counter [3]`) — на 15 объектов. В нем на один объект меньше, чем в `words`, потому что слово `"who"` встречается в нем дважды. Объект `quote` (обозначенный как `str [5]`) содержит простую строку, поэтому он не хранит никаких донолнительных ссылок.

На следующей диаграмме показаны обратные ссылки, которые ведут к объектам `words`, `quote` и `counts` (рис. 13.3).

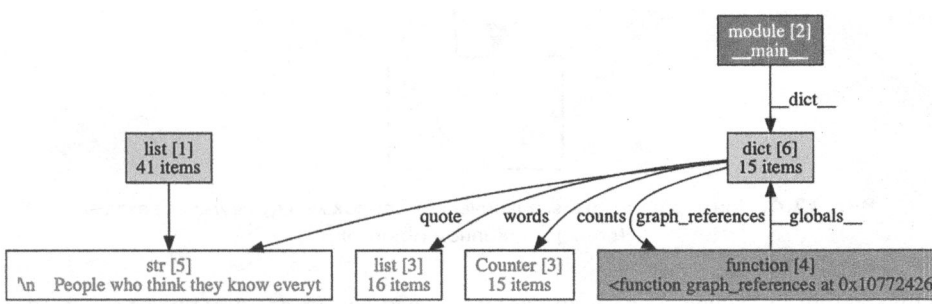

Рис. 13.3. Результат вызова show_backrefs() из функции graph_references()

Из диаграммы видно, что объекты `quote`, `words` и `counts` являются ссылками в словаре (обозначенном как `dict [6]`) глобальных переменных модуля `__main__` с именем `__globals__`. Кроме того, ссылка на объект `quote` хранится в специальном объекте списка (обозначенном как `list [1]`), что связано с механизмом интернирования строк CPython.

Интернирование строк — механизм оптимизации памяти, используемый в CPython. Когда загружается модуль, для большинства строковых литералов заранее выделяется память. Строки в Python являются неизменяемыми, так что повторные вхождения одного и того же строкового литерала будут ссылаться на один и тот же адрес памяти.

Чтобы продемонстрировать практическое применение `objgraph`, рассмотрим нример кода, который может вызывать проблемы с памятью в некоторых версиях Python. Как упоминалось в главе 9 «Интеграция Python с C и C++», CPython использует собственный сборщик мусора, который существует независимо от механизма счетчика ссылок. Он не используется для управления памятью в целом, а лишь решает проблему циклических ссылок. Во многих ситуациях объекты могут ссылаться друг на друга так, что их нельзя удалить простыми методами, основанными на подсчете ссылок. Вот простейший пример:

```
x = []
y = [x]
x.append(y)
```

Ситуация наглядно представлена на рис. 13.4.

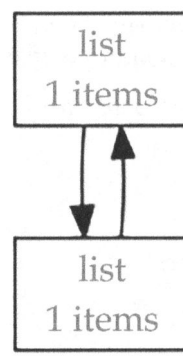

Рис. 13.4. Пример диаграммы циклических ссылок между двумя объектами (сгенерированной objgraph)

В предыдущем случае, даже если удалить все внешние ссылки на объекты `x` и `y` (например, при возврате из локальной области видимости функции), эти два

объекта нельзя будет утилизировать на основании счетчика ссылок, потому что всегда будут существовать две перекрестные ссылки, принадлежащие этим объектам. В этой ситуации в дело вступает сборщик мусора Python. Он может обнаруживать циклические ссылки на объекты и запускать освобождение их памяти, если нет других действительных ссылок на эти объекты за пределами цикла.

Настоящие проблемы начинаются тогда, когда хотя бы у одного из объектов в таком цикле определен нестандартный метод `__del__()`. Это пользовательский обработчик уничтожения объекта, который будет вызываться, когда счетчик ссылок объекта уменьшится до нуля. Он может выполнять произвольный код на Python, а следовательно, и создавать новые ссылки на объекты. Именно поэтому до Python 3.4 сборщик мусора не умел устранять циклические ссылки, если хотя бы один из объектов предоставлял нестандартную реализацию метода `__del__()`. Документ PEP 442 добавил в Python безопасную финализацию объектов и стал частью стандарта языка, начиная с Python 3.4. Однако описанная проблема может оставаться в пакетах, которые заботятся об обратной совместимости и ориентируются на широкий спектр версий интерпретатора Python. Следующий фрагмент кода демонстрирует, что циклический сборщик мусора ведет себя по-разному в разных версиях Python:

```
import gc
import platform
import objgraph

class WithDel(list):
    """ подкласс list с нестандартной реализацией __del__ """
    def __del__(self):
        pass


def main():
    x = WithDel()
    y = []
    z = []

    x.append(y)
    y.append(z)
    z.append(x)

    del x, y, z

    print("недостижимо до сбора мусора:    %s" % gc.collect())
    print("недостижимо после сбора мусора: %s" % len(gc.garbage))
    print("количество объектов WithDel:    %s" %
          objgraph.count('WithDel'))
```

```
if __name__ == "__main__":
    print("Версия Python: %s" % platform.python_version())
    print()
    main()
```

Следующий вывод этого фрагмента, выполняемого в Python 3.3, показывает, что циклический сборщик мусора в старых версиях Python не уничтожает объекты, для которых определен метод `__del__()`:

```
$ python3.3 with_del.py
Python version: 3.3.5

недостижимо до сбора мусора:    3
недостижимо после сбора мусора: 1
количество объектов WithDel:    1
```

В более новых версиях Python сборщик мусора надежно справляется с финализацией объектов, даже если они онределяют метод `__del__()`:

```
$ python3.3 with_del.py
Python version: 3.3.5

недостижимо до сбора мусора:    3
недостижимо после сбора мусора: 0
количество объектов WithDel:    0
```

Хотя в более поздних выпусках Python нестандартная финализация уже не угрожает утечкой памяти, она все еще может создать проблемы в приложениях, которые должны работать в разных средах. Как упоминалось ранее, функции `objgraph.show_refs()` и `objgraph.show_backrefs()` позволяют выявлять проблематичные объекты, которые участвуют в неразрывных циклических ссылках. Например, можно легко изменить функцию `main()`, чтобы она выводила все обратные ссылки на экземпляры `WithDel`, позволяя обнаружить утечку ресурсов:

```
def main():
    x = WithDel()
    y = []
    z = []

    x.append(y)
    y.append(z)
    z.append(x)

    del x, y, z

    print("недостижимы до сбора мусора:    %s" % gc.collect())
```

```
print("недостижимы после сбора мусора: %s" % len(gc.garbage))
print("количество объектов WithDel:    %s" %
      objgraph.count('WithDel'))

objgraph.show_backrefs(
    objgraph.by_type('WithDel'),
    filename='after-gc.png'
)
```

Если запустить этот пример в Python 3.3, мы получим диаграмму, изображенную на рис. 13.5. Она показывает, что `gc.collect()` не удалось удалить экземпляры `x`, `y` и `z`. Кроме того, `objgraph` выделяет красным цветом все объекты с нестандартным методом `__del__()`, чтобы упростить выявление подобных проблем:

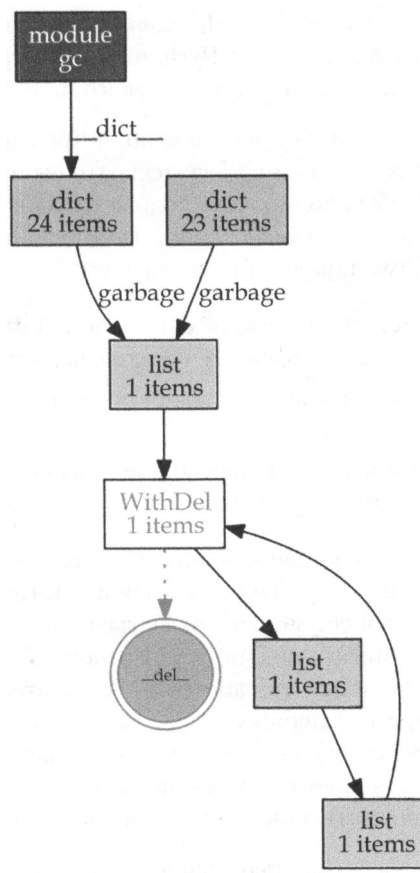

Рис. 13.5. Пример циклических ссылок, которые не может устранить сборщик мусора Python до версии 3.4

Если утечки памяти возникают в расширениях на C (например, в расширениях Python/C), их значительно сложнее диагностировать и профилировать. Однако сложно — не значит невозможно, как будет показано в следующем разделе.

Утечка памяти в коде C

Если код Python не вызывает подозрений, но затраты памяти продолжают расти при вызове изолированной функции в цикле, возможно, утечка на стороне C. Например, это происходит, когда в критической части импортированного расширения на C пропущен макрос `Py_DECREF`.

Код на C интерпретатора CPython весьма надежен и протестирован на отсутствие утечек памяти, так что это последнее место, где стоит искать проблемы с памятью. Но если вы используете пакеты с нестандартными расширениями на C, поиск следует начинать именно с них. Поскольку этот код работает на гораздо более низком уровне абстракции, чем Python, устранять утечки памяти приходится с помощью совершенно иных инструментов.

Отладка памяти в C — не самое простое дело, и прежде чем углубляться во внутреннее устройство расширений, убедитесь, что вы правильно определили источник проблемы. Чтобы изолировать подозрительные пакеты, очень часто применяется код, по своей природе похожий на модульные тесты. Для диагностики источника проблемы попробуйте сделать так:

1. Напишите отдельный тест для каждой единицы API или функциональности расширения, которое вы подозреваете в утечке памяти.
2. Запустите тест в цикле на продолжительное время в изоляции (одна тестовая функция на запуск).
3. Понаблюдайте снаружи, какой из тестируемых аспектов функциональности повышает расход памяти со временем.

Используя этот подход, вы в конце концов изолируете проблемную часть расширения и впоследствии потребуется меньше времени, чтобы проанализировать и исправить код. Этот процесс может показаться хлопотным, потому что он требует тратить время и писать код, но в долгосрочной перспективе хлопоты окупятся. Всегда можно упростить работу за счет повторного использования средств тестирования, представленных в главе 10 «Автоматизация тестирования и контроля качества». Вероятно, такие инструменты, как Pytest, не были предусмотрены специально для такого случая, но по крайней мере они экономят время на проведение множественных тестов в изолированных средах.

Если вы успешно изолировали часть расширения, в которой происходит утечка памяти, можно переходить собственно к отладке. Если вам повезет, простой ручной анализ изолированной части исходного кода может принести желаемый

результат. Во многих случаях, чтобы решить проблему, достаточно просто добавить пропущенный вызов `Py_DECREF`. Тем не менее чаще ваша работа будет не настолько простой, и придется задействовать более серьезные средства. Среди универсальных инструментов для борьбы с утечками памяти в скомпилированном коде стоит отметить Valgrind — целый фреймворк для построения средств динамического анализа, который должен входить в инструментарий каждого программиста. Изучить и освоить его на должном уровне не так просто, но вам определенно стоит ознакомиться хотя бы с основами использования Valgrind.

За дополнительной информацией о Valgrind обращайтесь по адресу https://valgrind.org.

После профилирования, когда вы знаете, что́ именно не в норядке с быстродействием вашего кода, наступает время оптимизации. В большинстве случаев плохое быстродействие объясняется сложностью кода. Очень часто сложность можно уменьшить, если просто применить подходящие структуры данных. Рассмотрим несколько примеров оптимизации со встроенными типами данных Python.

Уменьшение сложности за счет выбора структур данных

Чтобы уменьшить сложность кода, важно проанализировать, как хранятся ваши данные. Всегда внимательно выбирайте структуры данных. В следующем разделе приводятся примеры того, как улучшить быстродействие простых фрагментов кода, если использовать подходящие типы данных.

Поиск в списке

Из-за особенностей реализации типа `list` в Python поиск заданного значения в списке обходится относительно дорого. Метод `list.index()` имеет сложность $O(n)$, где n — количество элементов списка. Такая линейная сложность не создает проблем, если нужно найти два-три элемента, но может отрицательно сказаться на быстродействии на критических участках кода, особенно для очень больших списков.

Если вам приходится часто выполнять поиск в списке, нопробуйте модуль `bisect` из стандартной библиотеки Python. Функции этого модуля предназначены в основном для того, чтобы вставлять значения или находить индексы вставки для заданных значений, сохраняя порядок уже отсортированной носледовательности. С помощью `bisect` можно эффективно находить индексы элементов по алгорит-

му половинного деления. В следующем нримере из официальной документации модуля функция находит индекс элемента методом бинарного ноиска:

```
def index(a, x):
    'Находит крайнее левое значение, равное x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
```

Обратите внимание: чтобы функции из `bisect` работали, им нужна отсортированная носледовательность. Если список еще не отсортирован, то его сортировка будет операцией со сложностью в худшем случае $O(n \log n)$. Этот класс сложности хуже, чем $O(n)$, так что нерационально сортировать весь список для того, чтобы выполнить всего одип поиск. Однако если нужно выполнить несколько операций поиска по большому списку, который редко изменяется, одна процедура `sort()` для `bisect` может оказаться лучшим компромиссом.

Если у вас уже есть отсортированный список, `bisect` позволяет вставлять в него повые элемепты без пересортировки. Функции `bisect_left()` и `bisect_right()` возвращают точки вставки в списках, отсортированных слева направо и справа налево соответственно. Вот как с номощью функции `bisect_left()` можно вставить новое значение в список, отсортированный слева направо:

```
>>> from bisect import bisect_left
>>> items = [1, 5, 6, 19, 20]
>>> items.insert(bisect_left(items, 15), 15)
>>> items
[1, 5, 6, 15, 19, 20]
```

Функции `insort_left()` и `insort_right()` — сокращенные формы для вставки элементов в отсортироваппые сниски:

```
>>> from bisect import bisect_left
>>> items = [1, 5, 6, 19, 20]
>>> items.insert(bisect_left(items, 15), 15)
>>> items
[1, 5, 6, 15, 19, 20]
```

В следующем разделе мы покажем, как использовать множество вместо списка, когда элемепты должны быть уникальными.

Использование множеств

Когда вам нужпо построить последовательность уникальных значений по заданной последовательности, обычно первым в голову нриходит следующий алгоритм:

```
>>> sequence = ['a', 'a', 'b', 'c', 'c', 'd']
>>> result = []
>>> for element in sequence:
...     if element not in result:
...         result.append(element)
...
>>> result
['a', 'b', 'c', 'd']
```

В этом примере сложность появляется из-за поиска в списке `result`. Оператор `in` имеет временную сложность $O(n)$. Затем он используется в цикле, сложность которого — тоже $O(n)$. Таким образом, общая сложность будет квадратичной, то есть $O(n^2)$.

Работа пойдет быстрее, если применить тип `set`, потому что при этом для поиска хранимых значений используется хеширование (как и с типом `dict`). Тип `set` также гарантирует уникальность элементов, так что вам понадобится всего лишь создать новое множество на базе объекта последовательности. Иначе говоря, для каждого значения в последовательности программа будет за постоянное время проверять, есть ли этот элемент во множестве:

```
>>> sequence = ['a', 'a', 'b', 'c', 'c', 'd']
>>> unique = set(sequence)
>>> unique
set(['a', 'c', 'b', 'd'])
```

Сложность снижается до $O(n)$, то есть до сложности создания объекта множества. Когда вы используете тип `set`, чтобы обеспечить уникальность элементов, дополнительным преимуществом становится более короткий и выразительный код.

Иногда встроенных типов данных недостаточно, чтобы эффективно работать с вашими структурами данных. Python предоставляет большой набор высокопроизводительных типов данных в модуле `collections`.

Модуль collections

Модуль `collections` обеспечивает специализированные альтернативы для универсальных встроенных типов контейнеров. Вот основные типы из этого модуля, которые будут рассматриваться в этой главе:

- `deque`: аналог списка с расширенными возможностями.
- `defaultdict`: аналог словаря со встроенной фабричной функцией по умолчанию.
- `namedtuple`: аналог кортежа, в котором элементам назначаются ключи.

Другие типы из модуля collections уже рассматривались в предыдущих главах: ChainMap в главе 3 «Новые возможности Python», UserList и UserDict в главе 4 «Python в сравнении с другими языками» и Counter в главе 5 «Интерфейсы, паттерны и модульность».

В следующих разделах мы поговорим об этих типах подробнее.

deque

deque (дек) — альтернативная реализация снисков. Если встроенный тип `list` основан на обычных массивах, то deque основан на **двусвязном сннске**. Поэтому оп намного быстрее работает при вставке в начало или в конец, но нампого медленнее — при обращении по произвольному индексу.

Конечно, благодаря опережающему выделению памяти для внутрепнего массива в типе `list` не каждый вызов `list.append()` требует перераспределения памяти, и средняя сложность этого метода равна O(1). Ситуация кардинально меняется, когда требуется добавить элемент в начальную позицию списка. Так как все элементы справа от нового в массиве необходимо сдвипуть, сложность `list.insert()` составляет O(n). Если вы выполняете много операций pop, append и `insert`, то использование deque вместо `list` может существенно улучшить быстродействие.

Помните, что перед переходом с `list` на deque код всегда необходимо профилировать, потому что некоторые операции, которые быстро выполняются с массивами (например, обращение по произвольному индексу), весьма неэффективны в связанных списках.

Например, если с помощью `timeit` измерить время присоединения одного элемента и удаления его из последовательности, различия между `list` и deque могут быть незаметными.

Вот пример запуска `timeit` для тина `list`:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
```

А вот соответствующий пример для deque:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
```

Но если аналогичным образом сравнить случаи с добавлением и удалением первого элемента списка, различия в быстродействии впечатляют:

Вот пример запуска `timeit` для типа `list`:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.insert(0, 0); sequence.pop(0)'
1000000 loops, best of 3: 0.392 usec per loop
```

А вот соответствующий пример для deque:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.insert(0, 0); sequence.pop(0)'
1000000 loops, best of 3: 0.392 usec per loop
```

Как и следовало ожидать, разница растет с размером последовательности. Вот пример выполнения того же теста с `timeit` для списка из 10 000 элементов:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10));' \
  'sequence.insert(0, 0); sequence.pop(0)'
1000000 loops, best of 3: 0.392 usec per loop
```

Если сделать то же самое с deque, вы увидите, что время операции не изменилось:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.insert(0, 0); sequence.pop(0)'
1000000 loops, best of 3: 0.392 usec per loop
```

deque хорошо работает при реализации очередей, но в стандартной библиотеке Python также есть отдельный модуль queue, который предоставляет базовую реализацию очередей FIFO, LIFO (Last-In First-Out, «последним зашел — первым вышел») и очередей с приоритетом. Если вы хотите применять очереди как механизм межпроцессных коммуникаций, следует использовать классы модуля queue вместо `collections.deque`. Это связано с тем, что эти классы предоставляют всю необходимую семантику блокировки. Если вы не используете многопоточное выполнение и не применяете очереди как механизм взаимодействия, класса deque будет достаточно для базовой реализации очередей.

Благодаря эффективным методам append() и pop(), которые работают с одинаковой скоростью с обоих концов последовательности, deque идеально подходит

для реализации очередей. Например, очередь FIFO (First-In First-Out, «первым зашел — первым вышел») будет намного эффективнее, если реализовать ее на основе `deque` вместо `list`.

defaultdict

Тип `defaultdict` похож на `dict`, за исключением того, что он добавляет фабрику по умолчанию для новых ключей. Это избавляет от необходимости писать дополнительный код, чтобы инициализировать элементы отображения, а также повышает эффективность по сравнению с методом `dict.setdefault`.

Может показаться, что `defaultdict` — не более чем «синтаксический сахар» для `dict`, который позволяет писать более компактный код. Однако при неудачном поиске по ключу подстановка резервного значения выполняется немного быстрее, чем с методом `dict.setdefault()`.

Рассмотрим пример запуска `timeit` для метода `dict.setdefault()`:

```
$ python3 -m timeit \
  -s 'd = {}' \
  'd.setdefault("x", None)'
10000000 loops, best of 3: 0.153 usec per loop
```

А вот пример запуска `timeit` для эквивалентного использования `defaultdict`:

```
$ python3 -m timeit \
  -s 'd = {}' \
  'd.setdefault("x", None)'
10000000 loops, best of 3: 0.153 usec per loop
```

Различия в предыдущем примере могут показаться более существенными, но вычислительная сложность не изменилась. Метод `dict.setdefault()` состоит из двух шагов (поиск по ключу и присваивание), каждый из которых имеет сложность O(1). Классов сложности ниже O(1) не бывает, но иногда стоит поискать более быстрые альтернативы в том же классе O(1). При оптимизации критических секций кода пригодится даже незначительное улучшение скорости.

Тип `defaultdict` принимает в качестве параметра фабрику, благодаря чему его можно использовать со встроенными типами или классами, чьи конструкторы не принимают аргументов. Следующий фрагмент кода из официальной документации демонстрирует, как сделать счетчик с помощью `defaultdict`:

```
>>> from collections import defaultdict
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> list(d.items())
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

Для этого конкретного примера (счетчик уникальных элементов) в модуле `collections` тоже есть специальный класс `Counter`. Его можно использовать, чтобы эффективно получать набор элементов с наибольшим количеством вхождений.

namedtuple

`namedtuple` — фабрика классов, которая получает имя типа со списком атрибутов и создает класс на его основе. С помощью нового класса можно создавать объекты, похожие на кортежи, и пользоваться методами доступа к их элементам:

```
>>> from collections import defaultdict
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> list(d.items())
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

Как показано в предыдущем примере, с помощью `namedtuple` можно создавать структуры, которые удобнее записывать по сравнению с пользовательским классом, потому что он может требовать шаблонного кода для инициализации значений. С другой стороны, `namedtuple` создается на основе кортежа, так что обращение к его элементам по индексу — быстрая операция. Можно также создавать подклассы сгенерированного класса, чтобы добавлять новые операции.

Преимущества `namedtuple` перед другими типами данных могут быть неочевидны на первый взгляд. Главное преимущество — в том, что его проще использовать и интерпретировать, чем обычные кортежи. Индексы кортежей не передают никакой семантики, поэтому было бы удобно иметь возможность обращаться к элементам кортежа по атрибутам. Следует заметить, что тех же преимуществ можно было бы добиться с помощью словарей, у которых средняя сложность операций чтения и записи равна O(1).

Главное преимущество в отношении быстродействия заключается в том, что `namedtuple` все еще остается разновидностью кортежа. То есть это неизменяемая структура, для которой сразу выделяется необходимый объем памяти. А словари используют опережающее выделение памяти во внутренней хеш-таблице, чтобы сложность операций чтения и записи была ниже средней. Таким образом, `namedtuple` выигрывает по части эффективности памяти.

Тот факт, что `namedtuple` основан на кортеже, также способствует быстродействию. К элементам можно обращаться по целочисленному индексу, как и в других базовых объектах последовательностей — списках и кортежах. Это простая и быстрая операция. В случае `dict` или экземпляров пользовательских классов (которые хранят атрибуты с помощью словарей) обращение к элементу требует поиска по хеш-таблице. Словари тщательно оптимизированы, чтобы обеспечивать хорошее быстродействие независимо от размера коллекции, по как упоминалось ранее, O(1) считается всего лишь средней сложностью. Фактическая амортизированная сложность для операций чтения и записи в `dict` в худшем случае равна O(n). Реальный объем работы для выполнения такой операции зависит как от размера коллекции, так и от истории предыдущих операций. В разделах кода, критичных по быстродействию, имеет смысл использовать списки или кортежи вместо словарей, потому что они более предсказуемы. В таких ситуациях `namedtuple` — превосходный тип, который сочетает преимущества словарей и кортежей:

- В тех частях, где удобочитаемость важнее, можно обращаться к атрибутам.
- В частях, критичных по быстродействию, к элементам можно обращаться по индексам.

Именованные кортежи могут быть полезным средством оптимизации, но там, где важна удобочитаемость, классы данных лучше представляют данные, сходные со структурами. Классы данных и их преимущества рассматривались в главе 4 «Python в сравнении с другими языками».

Можно уменьшить сложность, если хранить данные в эффективной структуре, которая хорошо сочетается с тем, как алгоритм ее использует. Тем не менее если решение не очевидно, стоит задуматься о том, чтобы исключить и переписать проблемную часть кода, а не жертвовать удобочитаемостью ради быстродействия. Часто код Python можно сделать одновременно удобочитаемым и быстрым, поэтому старайтесь найти хороший способ выполнить работу, вместо того чтобы городить обходные решения из-за некачественного проектирования.

Но иногда у проблемы, с которой вы столкнулись, нет эффективного решения или у вас в распоряжении нет хорошей высокопроизводительной структуры. В таких ситуациях стоит подумать об определенных архитектурных компромиссах, примеры которых рассматриваются в следующем разделе.

Архитектурные компромиссы

Если ваш код не удается улучшить, сокращая сложность или выбирая подходящие структуры данных, то, возможно, стоит рассмотреть компромиссное решение. Если проанализировать проблемы пользователей и определить, что для них действительно важно, часто можно ослабить некоторые требования приложения. Нередко удается улучшить быстродействие, если:

- заменить точные алгоритмы решения эвристиками и аппроксимирующими алгоритмами;
- передать часть работы в очередь отложенных задач;
- использовать вероятностные структуры данных.

Давайте рассмотрим эти варианты улучшения.

Эвристики и аппроксимирующие алгоритмы

У некоторых алгоритмических задач просто нет хороших современных решений, которые можно выполнить за время, приемлемое для пользователей.

Например, рассмотрим программу, которая решает сложные задачи оптимизации, такие как задача коммивояжера или задача маршрутизации транспорта. И та и другая относятся к классу NP-сложных задач в комбинаторной оптимизации. Для них не известно точных алгоритмов с низкой сложностью, а это сильно ограничивает размер задачи, которую можно решить на практике. Скорее всего, для большого объема входных данных вам не удастся предоставить правильное решение за разумное время.

К счастью, пользователей чаще интересует не лучшее из возможных решений, а достаточно хорошее решение, которое можно найти своевременно. В таких случаях логично использовать эвристики или аппроксимирующие алгоритмы, которые обеспечивают приемлемый результат.

- **Эвристики** решают задачи, жертвуя оптимальностью, полнотой или точностью ради скорости. Качество таких решений бывает трудно оценить по сравнению с результатом точных алгоритмов.
- **Аппроксимирующие алгоритмы** используют примерно те же идеи, что и эвристики, но качество их решения обычно можно проверить, а время выполнения ограничено предсказуемыми рамками.

Известно много хороших эвристик и аппроксимирующих алгоритмов, которые позволяют решать исключительно большие задачи коммивояжера или маршрутизации транспорта за разумное время. Они также с большой вероятностью дают хорошие результаты, которые отличаются от оптимальных всего на 2–5 %.

У эвристик есть еще одно положительное качество: их не обязательно строить с нуля для каждой новой задачи. Их высокоуровневые версии, называемые **метаэвристиками**, предоставляют стратегии для решения математических задач оптимизации, которые не привязаны к конкретной задаче и, как следствие, могут применяться во многих ситуациях. Перечислим несколько популярных метаэвристических алгоритмов:

- **Имитация отжига** моделирует физические процессы, которые происходят при отжиге в металлургии (управляемый нагрев и охлаждение материалов).
- **Эволюционные вычисления** строятся по образцу биологических процессов и используют такие эволюционные механизмы, как мутации, воспроизводство, рекомбинация и отбор, для эффективного поиска по большой области решений в сложных задачах.
- **Генетические алгоритмы** — специализированная форма эволюционных вычислений, где возможные решения задачи представляются как наборы генотипов, а результаты получаются с помощью генетических преобразований (таких, как кроссинговер и мутации).
- **Поиск с запретами** — общий метод поиска решения, который вводит запрещенные пути поиска (табу), чтобы снизить вероятность того, что алгоритм сойдется на локальных оптимумах.
- **Алгоритм муравьиной колонии**, который имитирует поведение муравьев в колонии при поиске в возможном пространстве решений задач.

Эвристики и аппроксимирующие алгоритмы служат эффективными методами оптимизации, когда бóльшая часть работы проходит в одной алгоритмической задаче приложения. Но часто проблемы быстродействия обусловлены общей архитектурой системы и связями между разными ее компонентами.

Типичный архитектурный компромисс, который улучшает субъективно воспринимаемое быстродействие сложных приложений, основан на очередях задач и отложенной обработке.

Очереди задач и отложенная обработка

Иногда проблема не в том, чтобы сделать много, а в том, чтобы сделать вовремя. Типичный пример, который часто встречается в литературе, — отправка электронной почты из веб-приложения. В таком случае повышенное время отклика для запросов HTTP не обязательно связано с реализацией вашего кода. Время отклика может главным образом зависеть от сторонней службы, например от удаленного сервера электронной почты. Можно ли успешно оптимизировать приложение, если оно проводит бóльшую часть времени в ожидании ответа от других служб?

И да и нет. Если вы никак не контролируете службу, которая вносит решающий вклад в продолжительность обработки, и вам не удается найти более быстрое решение, то ускорить ничего не получится. Простой пример обработки запроса HTTP, который приводит к отправке электронной почты, изображен на рис. 13.6.

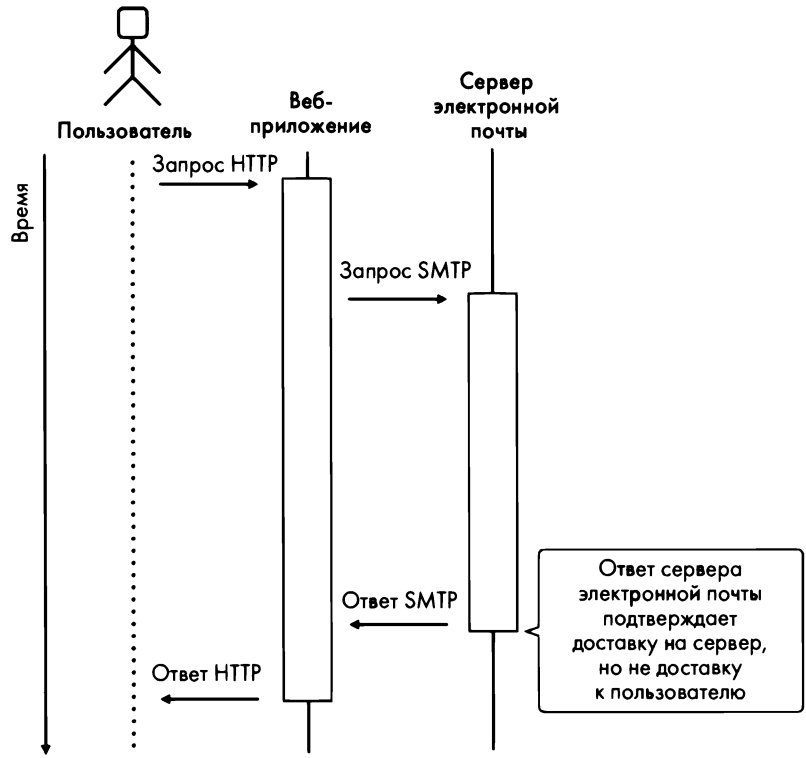

Рис. 13.6. Пример синхронной отправки электронной почты в веб-приложении

Если код зависит от сторонних служб, время ожидания часто не удается сократить. Однако можно повлиять на то, как пользователи его воспринимают.

Типичное решение основано на использовании очередей задач или сообщений (рис. 13.7). Когда операция занимает неопределенное время, добавьте ее в очередь невыполненной работы и немедленно сообщите пользователю, что его запрос был принят. Вот почему отправка электронной почты служит таким хорошим примером: в ней уже задействованы очереди задач! Когда вы отправляете новое сообщение на сервер электронной почты по протоколу SMTP, успешный ответ означает не то, что ваше сообщение доставлено адресату, а лишь то, что оно доставлено серверу электронной почты. Если ответ от сервера не

должен гарантировать, что почта была доставлена, вам не нужно ожидать доставки, чтобы сгенерировать ответ HTTP для пользователя:

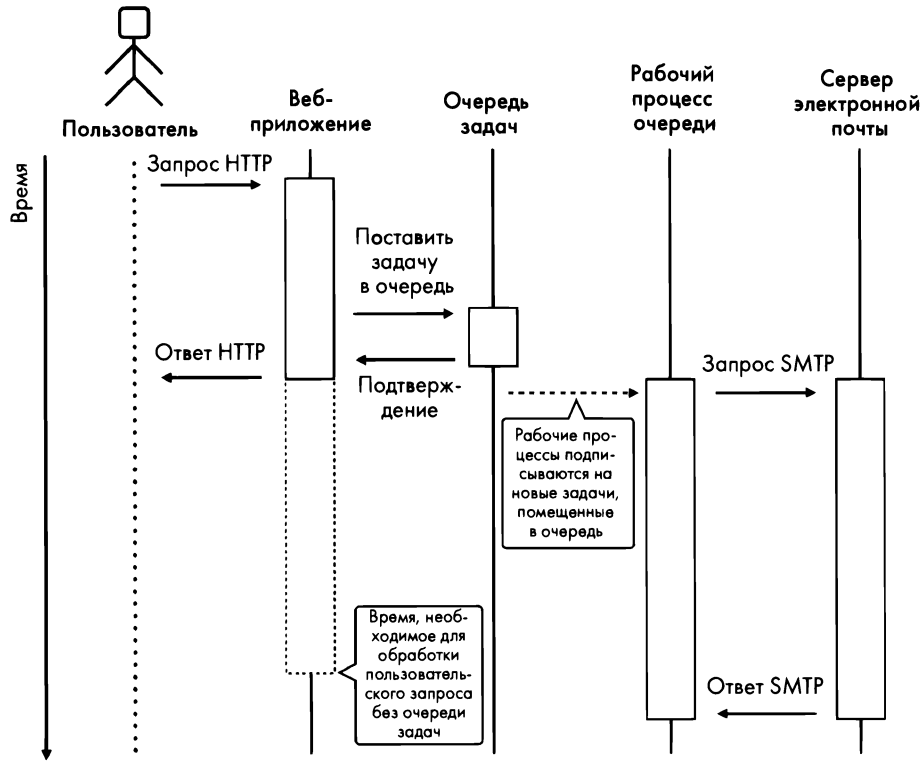

Рис. 13.7. Пример асинхронной отправки электронной почты в веб-приложении

Даже если ваш сервер электронной ночты отвечает молниеносно, вам все равно нонадобится время, чтобы сгеперировать отправляемые сообщения. Формируете ли вы ежегодные отчеты в формате XLS? Доставляете ли счета в виде файлов PDF? Если ваша система транспортировки электронной почты уже асинхронна, всю задачу генерации сообщений можно поместить в систему обработки сообщений. Если вы не можете гарантировать точное время доставки, вам не стоит беспокоиться о том, чтобы сообщения генерировались синхронно.

Правильное использование очередей сообщений и задач на критических участках приложения также может дать другие нреимущества:

- Рабочие объекты (Web workers), которые обслуживают запросы HTTP, освобождаются от донолнительной работы и могут быстрее обслуживать

запросы. Это позволяет обрабатывать больше запросов с теми же ресурсами, а следовательно, справляться с большей нагрузкой.

- Очереди сообщений обычно более устойчивы к временным сбоям внешних служб. Например, если у базы данных или сервера электронной почты периодически истекает время ожидания, всегда можно поместить текущую задачу обратно в очередь и повторить попытку позднее.
- С хорошей реализацией очереди сообщений можно легко распределить работу между несколькими машинами. Это может улучшить масштабируемость некоторых компонентов приложения.

Как видно из рис. 13.7, при добавлении асинхронного процесса в приложение неизбежно повышается сложность архитектуры всей системы. Вам придется подготовить сторонние службы (очередь сообщений — например, RabbitMQ) и создать рабочие процессы, которые способны обрабатывать асинхронные задания. К счастью, существует ряд популярных инструментов, с помощью которых можно строить распределенные очереди задач.

Celery — один из популярных инструментов Python для асинхронной обработки заданий. Celery — полнофункциональный фреймворк очередей задач, который поддерживает множественные брокеры сообщений и выполнение задач по расписанию. Он даже может заменять задания cron. О Celery можно больше узнать по адресу http://www.celeryproject.org/.

Если вам нужно что-то попроще, хорошей альтернативой может стать RQ. Этот инструмент намного проще Celery и использует хранилище данных «ключ — значение» Redis в качестве брокера сообщений Redis Queue (RQ). За дополнительной информацией о RQ обращайтесь по адресу http://python-rq.org/.

Несмотря на наличие хороших проверенных инструментов, всегда тщательно продумывайте свой подход к очередям задач. Не все разновидности задач стоит обрабатывать в очередях. Хотя очереди хорошо справляются со многими проблемами, они могут создать ряд других проблем:

- Системная архитектура становится сложнее.
- Одно и то же сообщение может быть обработано многократно.
- Нужно больше служб для сопровождения и контроля.
- Повышается задержка обработки.
- Усложняется журналирование.

Совершенно иной подход к архитектурным компромиссам основан на использовании недетерминированных (вероятностных) структур данных.

Вероятностные структуры данных

Вероятностные структуры данных хранят наборы значений таким способом, который позволяет отвечать на конкретные вопросы за ограниченный промежуток времени или с ограниченными ресурсами, если иначе сделать это невозможно. Типичный пример — эффективное хранение счетчиков уникальных просмотров на большой стриминговой платформе с миллиардами видеороликов и пользователей (например, YouTube). Наивная реализация с точной информацией о том, кто какой ролик просмотрел, потребовала бы гигантских объемов памяти, и скорее всего, не смогла бы работать эффективно. При таких колоссальных задачах стоит обратиться к вероятностным структурам данных.

Важнейшая особенность таких структур заключается в том, что ответы, которые они предоставляют, истинны только с некоторой долей вероятности; иначе говоря, они лишь аппроксимируют реальные значения. Впрочем, вероятность правильного ответа легко оценить. Хотя вероятностные структуры не всегда дают правильный ответ, они бывают полезны, если потенциальные ошибки до определенной степени допустимы.

Существует множество структур данных с такими вероятностными свойствами. Каждая из них предназначена для определенных задач, но из-за своей стохастической природы они годятся не для каждой ситуации. Для примера рассмотрим одну из самых популярных структур — HLL (HyperLogLog).

Алгоритм HLL аппроксимирует количество уникальных элементов в мультимножестве. Если вы захотите узнать количество уникальных элементов в обычном множестве, вам придется сохранить их все. Очевидно, такой подход непрактичен для очень больших наборов данных. HLL отличается от классического подхода, когда множества реализуются как программные структуры данных.

Не углубляясь в подробности реализации, скажем, что она обеспечивает только приближенную оценку количества элементов множества; реальные значения при этом не сохраняются. Их невозможно получить, перебрать или проверить на принадлежность множеству. HLL жертвует точностью ради того, чтобы уменьшить временну́ю сложность и затраты памяти. Например, реализация HLL для Redis занимает всего 12 Кбайт со стандартной ошибкой 0.81 %, а размер коллекции практически не ограничен.

Вероятностные структуры данных — интересный способ решения проблем производительности. В большинстве случаев все сводится к тому, что мы жертвуем точностью, чтобы ускорить обработку или повысить эффективность использования ресурсов. Впрочем, это нужно не всегда. Вероятностные структуры также часто применяются в системах хранения «ключ — значение», чтобы ускорить

поиск по ключу. Один из самых популярных методов, используемых в таких системах, называется AMQ (Approximate Member Query), и при этом часто применяется интересная вероятностная структура данных — **фильтр Блума**.

В следующем разделе мы поговорим о кэшировании.

Кэширование

Если некоторые функции вашего приложения выполняются слишком долго, стоит подумать о кэшировании. Оно сохраняет на будущее возвращаемые значения функций, запросов к базам данных, запросов HTTP и т. д. Результат функции или метода, выполнение которого обходится слишком затратно, можно кэшировать, когда выполняется одно из следующих условий:

- Функция является детерминированной и для одинаковых входных данных всегда возвращает одно и то же значение.
- Функция возвращает недетерминированное значение, но оно остается полезным и действительным в течение некоторого времени.

Иначе говоря, детерминированная функция всегда возвращает один и тот же результат для одного набора аргументов, а недетерминированная функция возвращает результаты, которые могут меняться со временем. Кэширование обоих видов результатов обычно сокращает время вычислений и позволяет экономить ресурсы.

Самое важное требование для любого решения из области кэширования — система хранения данных, из которой сохраненные значения читаются намного быстрее, чем они вычислялись бы. Вот несколько примеров удачных кандидатов для кэширования:

- Результаты вызываемых объектов, которые обращаются с запросами к базам данных.
- Результаты вызываемых объектов, которые генерируют статические значения: содержимое файлов, веб-запросы или документы в формате PDF.
- Результаты детерминированных вызываемых объектов, которые выполняют сложные вычисления.
- Глобальные отображения, которые отслеживают значения с ограниченным сроком жизни (например, объекты веб-сеансов).
- Результаты, к которым нужно обращаться быстро и часто.

Другой важный сценарий для кэширования — сохранение результатов сторонних API, которые получены от веб-приложений. Кэширование может значительно повысить быстродействие приложения, исключив сетевые задержки, но

оно также поможет сэкономить деньги, если оплата рассчитывается по количеству запросов к API.

В зависимости от архитектуры приложения кэширование можно реализовать по-разному и на разных уровнях сложности. В сложных приложениях могут использоваться разные подходы на разных уровнях стека архитектуры приложения. Иногда кэш может быть простым словарем или другой глобальной структурой данных, которая хранится в памяти процесса. В других ситуациях развертывают специальную службу кэширования, которая работает на специально настроенном оборудовании. В следующих разделах дан обзор самых популярных способов кэширования, а также рассматриваются некоторые стандартные сценарии его использования и типичные ловушки.

Итак, начнем с детерминированного кэширования.

Детерминированное кэширование

Детерминированные функции — самый простой и безопасный объект для кэширования. Такие функции всегда возвращают одно и то же значение для одних и тех же входных данных, так что в общем случае результаты кэширования можно хранить бесконечно. Такой подход ограничен только емкостью хранилища. В простейшем случае результаты кэширования хранятся в памяти процесса, потому что из памяти они обычно читаются быстрее всего. Этот прием часто называется **мемоизацией** (memoization).

Мемоизация чрезвычайно полезна, чтобы оптимизировать рекурсивные функции, которым иногда приходится многократно обрабатывать один и тот же ввод. Рекурсивные реализации уже обсуждались для чисел Фибоначчи в главе 9 «Интеграция Python с C и C++». Ранее в этой книге мы пытались улучшить быстродействие программы средствами C и C++, а теперь попробуем добиться той же цели более простым способом — с помощью кэширования. Но сначала вспомним, как выглядел код функции `fibonacci()`:

```
def fibonacci(n):
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Как видите, рекурсивная функция `fibonacci()` вызывает себя дважды, если входное значение больше 2. Из-за этого она крайне неэффективна. Сложность функции на стадии выполнения составляет $O(2^n)$, а при ее выполнении строится очень глубокое и необозримое дерево вызовов. При большом входном значении функция будет выполняться слишком долго. Также высока вероятность того, что она исчерпает максимальную глубину рекурсии для интерпретатора Python.

Если внимательно присмотреться к рис. 13.8, на котором изображено дерево вызовов функции `fibonacci()`, можно заметить, что многие промежуточные результаты вычисляются многократно. Можно сэкономить много времени и ресурсов, если повторно использовать некоторые из этих значений:

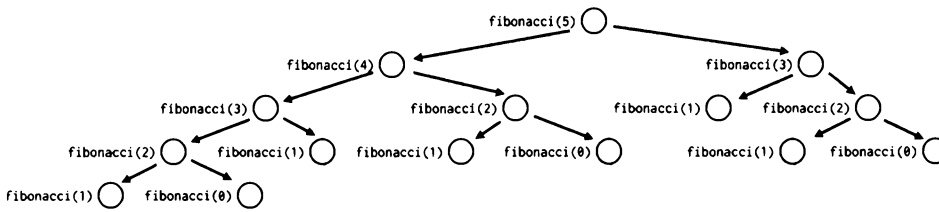

Рис. 13.8. Дерево вызовов для выполнения fibonacci(5)

В простейшем варианте мемоизации результаты предыдущих вызовов будут сохраняться в словаре и читаться по мере доступности.

Оба рекурсивных вызова в функции `fibonacci()` содержатся в одной строке кода:

```
return fibonacci(n - 1) + fibonacci(n - 2)
```

Мы знаем, что Python выполняет инструкции слева направо. Это означает, что в такой ситуации сначала будет вызвана функция с бо́льшим значением аргумента, а затем — с меньшим. Благодаря этому для мемоизации можно построить очень простой декоратор:

```
def memoize(function):
    call_cache = {}

    def memoized(argument):
        try:
            return call_cache[argument]
        except KeyError:
            return call_cache.setdefault(
                argument, function(argument)
            )

    return memoized
```

Затем этот декоратор применяется к функции `fibonacci()`:

```
@memoize
def fibonacci(n):
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Мы использовали словарь с замыканием декоратора `memoize()` как простой способ сохранять кэшированные значения. Сохранение и чтение значений в этой структуре данных имеет среднюю сложность O(1), а значит, общая сложность мемоизируемой функции значительно снижается. Каждый уникальный вызов функции будет обработан только один раз. Дерево вызовов такой обновленной функции представлено на рис. 13.9. Даже без математических доказательств можно наглядно увидеть, что сложность функции `fibonacci()` уменьшилась от очень высокой $O(2^n)$ до линейной $O(n)$:

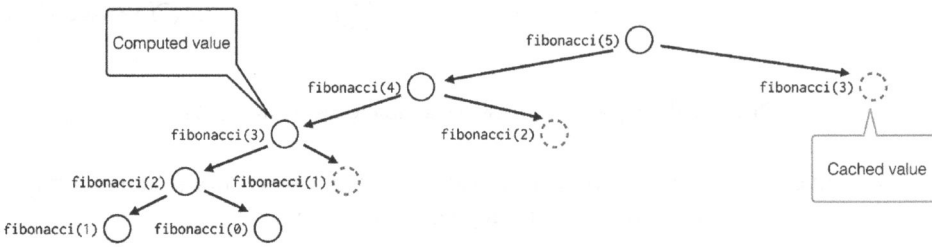

Рис. 13.9. Дерево вызовов для fibonacci(5) с мемоизацией

Конечно, эта реализация декоратора `memoize()` не идеальна. Она хорошо работала в приведенном примере, но этот код непригоден для повторного использования. Если нужно мемоизировать функции с несколькими аргументами или управлять размером кэша, потребуется более общее решение.

К счастью, в стандартной библиотеке Python есть очень простой и универсальный инструмент, который можно использовать в большинстве ситуаций детерминированного кэширования. Речь идет о декораторе `lru_cache()` из модуля `functools`. Имя происходит от сокращения LRU (Least Recently Used) — алгоритма, который удаляет давно не использовавшиеся данные. Дополнительные параметры позволяют точнее управлять мемоизацией:

- `maxsize`: задает максимальный размер кэша. Значение `None` означает, что ограничений нет.
- `typed`: определяет, должны ли значения разных типов отображаться на один и тот же результат, если при операции сравнения они считаются равными.

Использование `lru_cache()` в нашем примере с числами Фибоначчи могло бы выглядеть так:

```
from functools import lru_cache

@lru_cache(None)
def fibonacci(n):
```

```
if n < 2:
    return 1
else:
    return fibonacci(n - 1) + fibonacci(n - 2)
```

В следующем разделе мы перейдем к недетерминированному кэшированию.

Недетерминировонное кэширование

Кэширование недетерминированных функций сложнее мемоизации. Так как функция каждый раз может возвращать новый результат, обычно нельзя использовать предыдущие значения в течение произвольно долгого времени. Вместо этого приходится решать, как долго кэшированное значение может оставаться действительным. По истечении этого периода времени хранимые результаты считаются устаревшими, а кэш нужно обновить новым значением.

Другими словами, недетерминированное кэширование применяется, когда предварительно вычисленные результаты являются временными. Кэшированные недетерминированные функции часто зависят от внешнего состояния, которое трудно отслеживать в коде приложения. Типичные примеры таких компонентов:

- реляционные базы данных или другие средства структурированного хранения данных;
- сторонние службы, доступные по сети (веб-API);
- файловые системы.

Обратите внимание, что такая реализация является компромиссной. Если вы не выполняете часть кода тогда, когда это необходимо, и берете результаты из кэша, вы рискуете использовать данные, которые устарели или представляют несогласованное состояние вашей системы. В этом случае вы жертвуете точностью и/или полнотой ради быстродействия.

Конечно, такое кэширование эффективно, только если взаимодействие с кэшем происходит быстрее, чем выполняется кэшируемая функция. Если значение попросту быстрее вычислить заново — так и поступайте! Применяйте кэширование только тогда, когда оно того стоит, ведь его правильная настройка тоже не обходится даром.

Результаты, которые можно кэшировать, обычно появляются после взаимодействия с другими компонентами вашей системы. Например, если вы хотите сэкономить время и ресурсы при взаимодействии с базой данных, стоит кэшировать частые и высокозатратные запросы. Если нужно сократить количество операций ввода/вывода, кэшируйте содержимое файлов, к которым программа обращается чаще всего, или ответы от внешних API.

Методы кэширования недетерминированных функций очень похожи на те, которые используются для детерминированных. Самое заметное отличие заключается в том, что нужно предусмотреть удаление кэшированных значений по истечении срока годности. Это означает, что польза от декоратора `lru_cache()` из модуля `functools` ограничена, хотя не так уж сложно расширить эту функцию, чтобы она поддерживала срок годности. Поскольку это весьма типичная задача, которую многократно решали многие разработчики, в PyPI можно найти немало библиотек для кэширования недетерминированных значений.

Использование памяти локальных процессов обеспечивает быстрое кэширование, но каждый процесс поддерживает собственный кэш. Если процессов много, то независимые кэши будут занимать значительный объем памяти. В распределенных системах обычно используются службы распределенного кэширования.

Службы кэширования

Хотя недетерминированное кэширование можно реализовать с помощью локальной памяти процесса, в распределенных системах так обычно не делается: кэш приходится дублировать для каждой службы, что часто приводит к неэффективному расходу ресурсов. Более того, у разных экземпляров процессов может быть разное содержимое кэшей, и это может привести к рассогласованию данных.

Если вы попадете в ситуацию, когда недетерминированное кэширование окажется предпочтительным решением проблем с быстродействием, то вам, скорее всего, понадобится что-то еще. Обычно такое кэширование становится практически обязательным, если вы предоставляете данные или услуги многим пользователям одновременно.

Рано или поздно вам также придется позаботиться о том, чтобы пользователи обслуживались параллельно. Хотя данные, которые совместно используются несколькими потоками, можно хранить в локальной памяти, потоковое выполнение может оказаться не лучшей моделью конкурентности для каждого приложения. Оно недостаточно хорошо масштабируется, так что в конце концов вам придется запускать приложение в виде нескольких процессов.

Если повезет, ваше приложение сможет запускаться на сотнях и тысячах машин. Если вы при этом решите хранить кэшированные значения в локальной памяти, то кэш придется дублировать для каждого процесса, которому он нужен. Дело даже не в том, что это пустая трата ресурсов; если у каждого процесса свой собственный кэш, который уже означает компромисс между скоростью и целостностью, то как гарантировать, что все кэши будут согласованы друг с другом?

Согласование данных между последовательными запросами — серьезная проблема, особенно для веб-приложений с распределенными служебными подсистемами. В сложных распределенных системах чрезвычайно трудно гарантировать, что пользователь всегда будет обслуживаться одним и тем же процессом, который работает на одной машине. Конечно, это возможно до определенной степени, но после решения этой проблемы сразу возникают десять других.

Если приложение должно обслуживать многих пользователей одновременно, лучшее решение для недетерминированного кэширования — использовать специализированную службу. С помощью таких инструментов, как Redis или Memcached, вы позволяете всем процессам приложения совместно использовать одни и те же результаты кэширования. Тем самым снижаются затраты драгоценных вычислительных ресурсов и у вас не возникает множества независимых и рассогласованных кэшей.

Такие службы кэширования, как Memcached, полезны, чтобы реализовывать кэши в стиле мемоизации с состоянием, которые могут легко совместно использоваться между разными процессами и даже между разными серверами. Есть и другой способ кэширования, который можно реализовать на уровне системной архитектуры; он часто применяется в приложениях, работающих на основе протокола HTTP. Многие элементы типичного стека приложений HTTP предоставляют эластичные средства кэширования, в которых часто используются механизмы, стандартизированные протоколом HTTP. Например, кэширование такого рода может быть устроено так:

- Кэширующий обратный прокси-сервер (например, nginx или Apache) кэширует полные ответы от разных веб-воркеров (web workers), которые работают на одном хосте.
- Кэширующий распределитель нагрузки (например, HAProxy) не только распределяет нагрузку по нескольким хостам, но и кэширует их ответы.
- Сеть распространения контента кэширует ресурсы ваших серверов, пытаясь поддерживать их географическую близость к пользователю, чтобы сократить круговую задержку.

В следующем разделе рассматривается Memcached.

Memcached

Если вы хотите серьезно подойти к кэшированию, обратите внимание на Memcached — чрезвычайно популярное и проверенное решение. С помощью этого сервера кэширования масштабируют свои сайты крупные приложения, в том числе Facebook и Wikipedia. Среди простых функций кэширования Memcached также предоставляет средства кластеризации, которые позволяют

за минимальное время построить эффективную систему распределенного кэширования.

Memcached — мультиплатформенная служба, и для взаимодействия с ней существует множество библиотек для разных языков программирования. Клиенты для Python слегка отличаются друг от друга, но основная схема обычно остается неизменной. Простейшее взаимодействие с Memcached почти всегда состоит из трех методов:

- `set(key, value)`: сохраняет значение для заданного ключа.
- `get(key)`: получает значение для заданного ключа, если он существует.
- `delete(key)`: удаляет значение, связанное с заданным ключом, если он существует.

В следующем примере представлен пример интеграции с Memcached с помощью `pymemcache` — популярного пакета Python, доступного в PyPI:

```
from pymemcache.client.base import Client

# Настройка клиента Memcached на localhost, порт 11211
client = Client(('localhost', 11211))

# Кэширование значения с ключом; значение
# становится недействительным через 10 секунд
client.set('some_key', 'some_value', expire=10)

# Чтение значения для того же ключа
result = client.get('some_key')
```

К недостаткам Memcached можно отнести то, что система хранит значения в виде двоичных объектов. Это означает, что более сложные типы нужно сериализовать перед тем, как сохранять в Memcached. Простые данные обычно сериализуются в формате JSON. Рассмотрим пример использования сериализации JSON с `pymemcached`:

```
import json
from pymemcache.client.base import Client

def json_serializer(key, value):
     if type(value) == str:
         return value, 1
     return json.dumps(value), 2

def json_deserializer(key, value, flags):
    if flags == 1:
        return value
    if flags == 2:
```

```
        return json.loads(value)
    raise Exception("Unknown serialization format")

client = Client(('localhost', 11211), serializer=json_serializer,
                deserializer=json_deserializer)
client.set('key', {'a':'b', 'c':'d'})
result = client.get('key')
```

Еще одна распространенная проблема серверов кэширования, которые работают с парами «ключ — значение», связана с выбором имен ключей.

Когда вы кэшируете простые вызовы функций с несложными параметрами, решение обычно оказывается простым. Имя функции и ее аргументы преобразуются в строки, которые затем конкатенируются. Нужно беспокоиться только о том, чтобы не было конфликтов между ключами от разных функций, если кэширование происходит в разных точках приложения.

Более хитрая ситуация возникает, если у кэшируемых функций есть сложные аргументы со словарями или пользовательскими классами. В этом случае приходится придумывать, как логически непротиворечиво преобразовывать сигнатуры вызова в ключи кэша.

Многие службы кэширования (включая Memcached) хранят свои кэши в оперативной памяти, чтобы поиск в кэше был максимально быстрым. Если рабочий набор данных становится слишком большим, старые ключи обычно удаляются из кэша. Также весь кэш может очищаться при перезапуске службы. Важно учитывать это обстоятельство, если вы используете службы кэширования для данных, которые должны храниться долгосрочно. Иногда имеет смысл обеспечить процедуру инициализации кэша, которая заполнит его самыми частыми элементами (например, в случае обновления службы или выпуска новой версии приложения).

Последняя проблема заключается в том, что Memcached, как и многие другие серверы кэширования, плохо реагирует на очень длинные строки ключей, что может снизить быстродействие и даже исчерпать жесткие ограничения службы. Например, если вы кэшируете целые запросы SQL, то в общем случае сами строки запросов являются уникальными идентификаторами, которые можно использовать в качестве ключей. С другой стороны, сложные запросы обычно слишком длинны, чтобы их можно было хранить в Memcached. Как правило, в таких ситуациях сначала вычисляется MD5, SHA или другая функция хеширования, результат которой затем используется в качестве ключа кэша. В стандартную библиотеку Python входит модуль `hashlib`, который предоставляет реализации нескольких популярных алгоритмов хеширования. Используя функции хеширования, также не забывайте о конфликтах значений. Ни одна хеш-функция не исключает конфликты полностью, поэтому будьте готовы преодолевать потенциальные риски.

Итоги

В этой главе вы узнали о процессе оптимизации — от выявления возможных узких мест с номощью стандартных методов профилирования до полезных стратегий оптимизации для решения широкого спектра проблем.

Эта глава завершает книгу подобно тому, как оптимизация обычно завершает цикл разработки приложепия. Онтимизация применяется к приложениям, которые заведомо хорошо работают. Вот почему важно иметь готовые методологии и процессы, которые гарантируют, что ваше приложение продолжит нормально работать.

Хотя оптимизация часто сокращает алгоритмическую и вычислительную сложность, она может повышать другие виды сложности. Онтимизированные приложения нередко сложнее читать и нонимать, а следовательно, и сопровождать. Архитектурные компромиссы часто заключаются в том, чтобы использовать специализированные службы или примепять решения, которые жертвуют частью правильности или точности приложения. У приложений, которые прибегают к таким архитектурным комнромиссам, почти всегда более сложная архитектура.

Онтимизация кода, как и любая другая практика разработки, требует квалификации и опыта. Часть этого опыта — умение находить баланс между различными процессами и операциями разработки. Иногда заниматься мелкой оптимизацией вообще не стоит. Иногда стоит нарушить отдельные нравила, чтобы удовлетворить потребности бизнеса. Поэтому в книге мы постарались отразить целостное нредставление всего процесса разработки приложения. Возможно, вы не всегда будете делать все своими руками, но если вы будете знать, как нриложения разрабатываются, сопровождаются, тестируются, выпускаются, контролируются и оптимизируются, это поможет выдержать правильный баланс между этими видами деятельности.

Поделитесь впечатлениями.

Спасибо за то, что вы не пожалели времени на чтение книги. Если она вам понравилась, помогите найти ее другим — оставьте отзыв по адресу

https://www.amazon.com/dp/1801071101.

Михал Яворски, Тарек Зиаде

Python. Лучшие практики и инструменты

4-е издание

Перевел с английского Е. Матвеев

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*Р. Чебыкин*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 07.07.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 47,730. Тираж 700. Заказ 9168.

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н

Python. Лучшие практики и инструменты

Packt>

Написать код на Python легко, но сделать его удобочитаемым и пригодным для повторного использования и сопровождения может оказаться проблемой. Четвертое издание этой книги дополнено лучшими практиками, полезными инструментами и стандартами, которые применяют профессиональные разработчики, что поможет вам не только преодолеть эти затруднения, но и освоить новейшие возможности и расширенные концепции языка.

Книга начинается с легкой разминки, которая познакомит вас с последними улучшениями Python, элементами синтаксиса и полезными инструментами, делающими разработку эффективнее. Кроме того, начальные главы помогут программистам с опытом работы на других языках успешно влиться в экосистему Python.

Следующие главы посвящены распространенным паттернам проектирования и методологиям программирования — таким как событийно-ориентированное программирование, параллелизм и метапрограммирование. Также вы разберете сложные примеры кода и будете решать содержательные задачи, связывая Python с C и C++ и создавая расширения, сочетающие сильные стороны разных языков. В заключительных главах рассматривается полный жизненный цикл приложения после ввода в эксплуатацию.

К концу книги вы освоите разработку эффективного и простого в сопровождении кода на Python.

Вы научитесь:

- по-современному настраивать воспроизводимые и целостные среды разработки Python;
- эффективно формировать пакеты;
- использовать современные элементы синтаксиса Python, включая f-строки, классы данных, перечисления и лямбда-функции;
- применять средства метапрограммирования с помощью метаклассов;
- отслеживать и оптимизировать производительность приложений;
- расширять код на Python и интегрировать его с кодом на других языках.

ПИТЕР
WWW.PITER.COM
интернет-магазин
Заказ книг:
(812) 703-73-74
books@piter.com
PiterBooks
PiterForPeople
ThePiterBooks
Company/piter

ISBN:978-5-4461-2064-2