Python

Лучшие практики и инструменты

ТРЕТЬЕ ИЗДАНИЕ

Михал Яворски, Тарек Зиаде

ПИТЕР®

Packt›

Expert Python Programming
Third Edition

Become a master in Python by learning coding best practices and advanced programming concepts in Python 3.7

Michal Jaworski
Tarek Ziade

Packt>

BIRMINGHAM - MUMBAI

Python

ЛУЧШИЕ ПРАКТИКИ И ИНСТРУМЕНТЫ

Михал Яворски
Тарек Зиаде

ТРЕТЬЕ ИЗДАНИЕ

ПИТЕР®
Санкт-Петербург • Москва • Минск
2021

ББК 32.973.2-018.1
УДК 004.43
Я22

Яворски Михал, Зиаде Тарек

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

ISBN 978-5-4461-1589-1

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

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

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1789808896 англ.

ISBN 978-5-4461-1589-1

Краткое содержание

Оглавление

Часть I. Перед началом работы

Часть II. Ремесло Python

Часть III. Качество, а не количество

Часть IV. Жажда скорости

Часть V. Техническая архитектура

Благодарю любимую жену Оливию за ее любовь,
вдохновение и бесконечное терпение,
моих верных друзей Петра, Дэниела и Павла
за их поддержку,
мою мать, открывшую предо мной удивительный
мир программирования.

Михал Яворски

О создателях книги

Об авторах

Михал Яворски — программист на Python с десятилетним опытом. Занимал разные должности в различных компаниях: от обычного фулстек-разработчика, затем архитектора программного обеспечения и, наконец, до вице-президента по разработке в динамично развивающейся стартап-компании. В настоящее время Михал — старший бэкенд-инженер в Showpad. Имеет большой опыт в разработке высокопроизводительных распределенных сервисов. Кроме того, является активным участником многих проектов Python с открытым исходным кодом.

Тарек Зиаде — Python-разработчик. Живет в сельской местности недалеко от города Дижон во Франции. Работает в Mozilla, в команде, отвечающей за сервисы. Тарек основал французскую группу пользователей Python (называется Afpy) и написал несколько книг о Python на французском и английском языках. В свободное от хакинга и тусовок время занимается любимыми хобби: бегом или игрой на трубе.

Вы можете посетить его личный блог (Fetchez le Python) и подписаться на него в Twitter (tarek_ziade).

О научном редакторе

Коди Джексон — кандидат наук, основатель компании Socius Consulting, работающей в сфере IT и консалтинга по управлению бизнесом в Сан-Антонио, а также соучредитель Top Men Technologies. В настоящее время работает в CACI International ведущим инженером по моделированию ICS/SCADA. В IT-индустрии с 1994 года, еще со времен службы в ВМФ в качестве ядерного химика и радиотехника. До CACI он работал в университете в ECPI в должности ассистента профессора по компьютерным информационным системам. Выучился программированию на Python самостоятельно, написал книги *Learning to Program Using Python* и *Secret Recipes of the Python Ninja*.

Предисловие

Python — динамический язык программирования, применимый в широком спектре задач благодаря своей простой, но мощной сути. Писать на Python легко, но сделать код удобочитаемым, универсальным и простым в сопровождении — сложно. В третьем издании данной книги вы ознакомитесь с практическими рекомендациями, полезными инструментами и стандартами, используемыми профессиональными разработчиками на Python, так что сумеете преодолеть данную проблему.

Мы начнем эту книгу с новых возможностей, добавленных в Python 3.7. Изучим синтаксис Python и рассмотрим, как применять самые современные концепции и механизмы объектно-ориентированного программирования. Помимо этого, исследуем различные подходы к реализации метапрограммирования. Данная книга расскажет о присваивании имен при написании пакетов, создании исполняемых файлов, а также о применении мощных инструментов, таких как `buildout` и `virtualenv`, для развертывания кода на удаленных серверах. Вы узнаете, как создавать полезные расширения Python на языках C, C++, Cython и Pyrex. Кроме того, чтобы писать чистый код, вам будет полезно изучить инструменты управления кодом, написания ясной документации и разработки через тестирование.

Изучив эту книгу, вы станете экспертом в написании эффективного и удобного в сопровождении кода на Python.

Для кого эта книга

Книга написана для разработчиков на Python, желающих продвинуться в освоении этого языка. Под разработчиками мы имеем в виду в основном программистов, которые зарабатывают на жизнь программированием на Python. Дело в том, что книга сосредоточена на средствах и методах, наиболее важных для создания производительного, надежного и удобного в сопровождении программного обеспечения на Python.

Это не значит, что в книге нет ничего интересного для любителей. Она отлично подойдет для тех, кто хочет выйти на новый уровень в изучении Python. Базовых

навыков языка будет достаточно, чтобы понять изложенный материал, хотя менее опытным программистам придется приложить некоторые усилия. Книга также будет хорошим введением в Python 3.7 для тех, кто слегка отстал от жизни и пользуется версией Python 2.7 или еще более ранней.

Наибольшую пользу данная книга принесет веб- и бэкенд-разработчикам, поскольку в ней представлены две темы, особенно важные именно для этих специалистов: надежное развертывание кода и параллелизм.

Что мы рассмотрим

В главе 1 описано текущее состояние Python и его сообщества. Мы увидим, как меняется язык, из-за чего это происходит и почему данные факты очень важны для тех, кто хочет называть себя профессионалом в Python. Мы также рассмотрим наиболее известные и канонические способы работы с кодом Python, а именно популярные инструменты обеспечения производительности и правила, которые сегодня фактически являются стандартами.

В главе 2 представлены современные способы создания повторяемых и последовательных сред разработки для программистов на Python. Мы сосредоточимся на двух популярных инструментах для изоляции среды: средах типа `virtualenv` и контейнерах Docker.

В главе 3 даны практические рекомендации по написанию кода на Python (идиомы языка), а также краткое описание отдельных элементов синтаксиса Python, которые могут оказаться новыми для программистов, более привыкших к старым версиям Python. Кроме того, мы изложим пару полезных идей о внутренних реализациях типа CPython и их вычислительной сложности в качестве обоснования для рассмотренных идиом.

В главе 4 рассмотрены более сложные концепции и механизмы объектно-ориентированного программирования, доступные в Python.

В главе 5 представлен обзор общих подходов к метапрограммированию для программистов на Python.

В главе 6 приведено руководство по наиболее общепринятому стилю написания кода на Python (PEP 8), а также указано, когда и почему разработчики должны соблюдать его. Вдобавок мы рассмотрим некоторые общие рекомендации по назначению имен.

В главе 7 описаны особенности создания пакетов на Python и даны рекомендации по созданию пакетов, распространяемых в виде открытого исходного кода в **каталоге пакетов Python** (Python Package Index, PyPI). Мы также рассмотрим тему, которую часто игнорируют, — исполняемые файлы.

В главе 8 представлены некоторые облегченные инструменты для развертывания кода Python на удаленных серверах. Развертывание — это одна из областей,

где Python предстает во всей красе в реализации бэкенда для веб-сервисов и приложений.

В главе 9 объясняется, почему иногда удобно добавлять в код расширения на C и C++, и показывается, что при наличии подходящих инструментов сделать это будет проще, чем кажется.

В главе 10 рассказывается, как правильно управлять кодовой базой и почему следует использовать систему управления версиями. Мы опробуем на деле возможности такой системы (а именно, Git) в осуществлении непрерывных процессов, таких как непрерывная интеграция и непрерывная доставка.

В главе 11 приводятся общие правила написания технической документации, применимые к программному обеспечению, написанному на любом языке, а также различные инструменты, которые будут особенно полезны при создании документации для вашего кода на Python.

В главе 12 представлены плюсы подхода «разработка через тестирование» и подробно рассказывается о том, как использовать популярные инструменты Python для тестирования.

В главе 13 обсуждаются базовые правила оптимизации, которые должен знать каждый разработчик. Мы также научимся выявлять узкие места в производительности приложений и использовать общие инструменты профилирования.

В главе 14 показывается, как применить эти знания так, чтобы ваше приложение действительно работало быстрее или эффективнее с точки зрения используемых ресурсов.

В главе 15 объясняется, как реализовать параллелизм на Python с помощью различных подходов и готовых библиотек.

В главе 16 рассказывается, что такое событийно-ориентированное и сигнальное программирование и как оно связано с асинхронным и различными моделями параллелизма. Мы представим разные подходы к событийному программированию, доступные программистам на Python, а также полезные библиотеки, позволяющие применять эти шаблоны.

В главе 17 описаны несколько полезных паттернов проектирования и примеры их реализации на Python.

Приложение содержит краткое руководство по использованию языка разметки reStructuredText.

Как получить максимум от этой книги

Данная книга написана для программистов, работающих в любой операционной системе, где установлен Python 3.

Издание не подходит для начинающих, поэтому мы предполагаем, что вы установили Python в своей среде или вы знаете, как это сделать. Однако в книге учитывается

тот факт, что не все могут знать о последних функциях Python или официально рекомендованных инструментах. Именно поэтому в первой главе приведен обзор наиболее часто используемых утилит (например, виртуальных сред и pip), которые в настоящее время профессиональные разработчики на Python считают стандартными инструментами.

Скачивание файлов с примерами кода

Вы можете скачать файлы примеров кода для этой книги на сайте github.com/PacktPublishing/Expert-Python-Programming-Third-Edition.

Чтобы скачать файлы кода, выполните следующие действия.

1. Перейдите по указанной ссылке на сайт github.com.
2. Нажмите кнопку Clone or Download.
3. Щелкните кнопкой мыши на ссылке Download ZIP.
4. Скачайте архив с файлами примеров.

После скачивания файла распакуйте его с помощью последней версии одной из следующих программ:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Скачивание цветных изображений

Мы также выложили оригинальный файл PDF, в котором приведены цветные изображения снимков экрана/схем, используемых в этой книге. Вы можете скачать его по ссылке www.packtpub.com/sites/default/files/downloads/9781789808896_ColorImages.pdf.

Условные обозначения

В этой книге используется ряд текстовых и символьных обозначений.

`Код в тексте`: такой формат обозначает кодовые слова в тексте, имена таблиц базы данных, имена папок и файлов, расширения файлов, пути к файлам, URL-адреса, пользовательский ввод и инструменты Twitter. Например: «Любая попытка запустить код, в котором есть такие проблемы, заставит интерпретатор завершить работу, выбросив исключение `SyntaxError`».

Блок кода выглядит следующим образом:

```
print("hello world")
print "goodbye python2"
```

Любой ввод или вывод из командной строки записывается так:

```
$ Python3 script.py
```

Новые термины и важные слова выделены *курсивом*.

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.

Часть I
Перед началом работы

Эта часть призвана помочь пользователю подготовиться к современным реалиям разработки на Python. Мы рассмотрим, как язык изменился за последние несколько лет и какими инструментами разработки пользуются современные программисты на Python.

1 Текущее состояние Python

Python удивителен.

На протяжении долгого времени одним из самых важных достоинств Python была его совместимость. Независимо от того, какую операционную систему используете вы или ваши клиенты, если у нее есть интерпретатор Python, то ваше написанное на Python ПО будет в ней работать. И что важнее всего — работать так, как нужно. Однако сейчас этим никого не удивишь. Современные языки, например Ruby и Java, предоставляют аналогичные возможности для взаимодействия. Но в наше время совместимость не самое важное качество языка программирования. С появлением облачных вычислений, веб-приложений и надежного программного обеспечения для создания виртуальных окружений вопросы совместимости и независимости от операционной системы отошли на второй план. Однако попрежнему важны инструменты, позволяющие программистам эффективно писать надежное и удобное в сопровождении ПО. К счастью, Python относится к тем языкам, благодаря которым программисты могут работать наиболее эффективно, и для развития компаний это лучший выбор.

Python так долго не теряет актуальности благодаря тому, что постоянно развивается. Эта книга ориентирована на последнюю версию Python 3.7, и все примеры кода написаны именно в ней, если не сказано иное. Поскольку Python имеет очень длинную историю и еще есть программисты, пишущие на Python 2, данная книга начинается с обзора текущего *статус-кво* Python 3. В этой главе вы узнаете, как и почему Python изменился и как писать программное обеспечение, совместимое и со старыми, и с последними версиями Python.

В этой главе:

- где мы находимся и куда движемся;
- почему и как изменился язык Python;
- как не отставать от изменений в документации PEP;
- принятие Python 3 на момент написания этой книги;
- основные различия между Python 3 и Python 2;
- не только CPython;
- полезные ресурсы.

Технические требования

Для этой главы скачать последнюю версию Python можно по ссылке www.python.org/downloads/.

Альтернативные реализации интерпретатора Python можно найти на следующих сайтах:

- Stackless Python: github.com/stackless-dev/stackless;
- PyPy: pypy.org;
- Jython: www.jython.org;
- IronPython: ironpython.net;
- MicroPython: micropython.org.

Файлы с примерами кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter1.

Где мы находимся и куда движемся

История Python началась где-то в конце 1980-х годов, но релиз версии 1.0 состоялся в 1994 году. То есть это не молодой язык. Мы бы могли пройтись по всей хронологии версий Python, однако на самом деле нас интересует только одна дата: 3 декабря 2008 года — выход Python 3.0.

На момент написания этой книги прошло почти десять лет с появления первого релиза Python 3. Кроме того, прошло семь лет после выпуска PEP 404 — официального документа, в котором был *отменен выпуск* Python 2.8 и официально закрыта вся серия 2.x. С тех пор прошло много времени, однако сообщество Python все еще делится на два лагеря: несмотря на то что язык развивается очень быстро, есть большая группа пользователей, которые не хотят идти с ним в ногу.

Почему и как изменился язык Python

Ответ прост — Python изменяется, поскольку в этом есть необходимость. Конкуренты не спят. Каждые несколько месяцев из ниоткуда появляется новый язык, претендующий на решение всех проблем своих предшественников. Разработчики быстро утрачивают интерес к большинству подобных проектов, и их популярность часто вызвана исключительно хайпом.

За этим кроется более серьезная проблема. Люди берутся за разработку новых языков, поскольку считают, что существующие не решают их проблем. Было бы глупо отрицать необходимость в новых решениях. Кроме того, все более широкое использование Python показывает: язык можно и нужно улучшать.

Множество улучшений в Python обусловлены потребностями конкретных сфер, в которых он применяется. Наиболее значимая из них — веб-разработка. Так, постоянно растущий спрос на скорость и производительность в этой области привел к тому, что работа с параллелизмом в Python значительно упростилась.

Некоторые изменения вызваны попросту солидным возрастом и зрелостью проекта Python. На протяжении многих лет он обрастал множеством неорганизованных и избыточных модулей стандартных библиотек и даже плохими проектными решениями. То есть выпуск Python 3 был призван подчистить и освежить язык. К сожалению, время показало: данный план имел и неприятные последствия. В течение долгого времени Python 3 использовался многими разработчиками несерьезно. Будем надеяться, это изменится.

Как не отставать от изменений в документации PEP

Сообщество Python придумало устоявшийся способ реагирования на изменения. Хотя рискованные идеи языка Python в основном обсуждаются в рассылках (python-ideas@python.org), по-настоящему серьезные изменения сопровождаются выходом документа под названием *Python Enhancement Proposal (PEP)*.

Это формализованный документ, в котором подробно описывается предложение об изменении Python. Он также является отправной точкой для обсуждения в сообществе. Вся цель, формат и рабочий процесс вокруг данных документов также стандартизированы в документе PEP 1 (www.python.org/dev/peps/pep-0001).

PEP-документация очень важна для Python и, в зависимости от темы, выполняет разные функции:

- *информирования* — приводит информацию, необходимую разработчикам ядра Python, и графики выпуска версий Python;
- *стандартизации* — содержит указания по стилю кода, документации или другие руководящие принципы;
- *проектирования* — описывает предлагаемые функции.

Список всех предлагаемых PEP приведен в *постоянно обновляемом* документе PEP 0 (www.python.org/dev/peps/). Найти их легко, а ссылку на них нетрудно сформировать самостоятельно, поэтому в книге мы будем называть их лишь по номерам.

Документ PEP 0 — важный источник информации для тех, кому интересно, в каком направления движется язык Python, но некогда отслеживать каждое обсуждение в рассылках Python. В PEP 0 показано, какие документы уже были приняты, но еще не реализованы, а какие находятся на рассмотрении.

Документы PEP выполняют и другие функции. Очень часто люди задают вопросы а-ля:

- «Почему функция А работает именно так?»;
- «Почему в Python нет функции Б?».

В большинстве случаев в конкретных документах PEP уже имеется развернутый ответ. Существует много PEP-документов с описанием возможностей языка Python, которые были предложены, но не приняты. Эти документы играют роль своего рода исторической справки.

Внедрение Python 3 на момент написания этой книги

Если в Python столько новых и интересных функций, то наверняка он хорошо принят в сообществе? Сложно сказать. Некогда популярная страница «Стена суперсил Python 3» (python3wos.appspot.com), на которой отслеживалась совместимость самых популярных пакетов и Python 3, изначально была названа «Стеной позора Python 3». Этот сайт больше не поддерживается, однако на момент последнего обновления от 22 апреля 2018 года видно, что 191 из 200 наиболее популярных пакетов Python были совместимы с Python 3. Таким образом, можно убедиться, что эту версию хорошо приняли в сообществе программистов Python с открытым исходным кодом. Тем не менее это не значит, что все команды программистов полностью перешли на Python 3. По крайней мере, коль скоро большинство популярных пакетов Python доступны в Python 3, отговорки наподобие *«то, чем мы пользуемся, еще не портировали»* уже не актуальны.

Основная причина такой ситуации заключается в том, что портирование существующего приложения с Python 2 на Python 3 — всегда сложная задача. Есть инструменты, такие как `2to3`, позволяющие выполнять автоматизированный перевод кода, но не гарантирующие, что результат будет правильным на 100 %. Кроме того, такой «переведенный» код может утратить производительность, если не прибегнуть к ручной регулировке. Перевод существующего сложного кода на Python 3 может повлечь огромные усилия и расходы, которые могут себе позволить не все организации. К счастью, подобные расходы можно распределить во времени. Некоторые хорошие методологии проектирования архитектуры программного обеспечения, такие как сервис-ориентированная архитектура или микросервисы, дают возможность постепенно достичь этой цели. Новые компоненты проекта (сервисы или микросервисы) можно писать по новой технологии, а существующие — портировать по одному.

В перспективе переход на Python 3 может иметь только положительные последствия для проекта. Согласно PEP 404 поддержка Python 2 закончилась в 2020 году. До этого времени мы можем ожидать только обновления версии патча, решающего проблемы безопасности, но ничего более. Кроме того, в будущем, возможно, настанет время, когда все крупные проекты, такие как Django, Flask и NumPy, отключат совместимость с 2.x и полностью перейдут на Python 3. В Django уже сделали этот шаг, и, начиная с версии 2.0.0, он больше не поддерживает Python 2.7.

Наше мнение по данному вопросу противоречиво. Мы думаем, что лучшим стимулом для отказа сообщества от Python 2 будет прекращение поддержки Python 2 при создании новых пакетов. Конечно, это ограничивает создание нового программного обеспечения, но, возможно, послужит единственно верным способом изменить мышление тех, кто никак не отвыкнет от Python 2.x.

Мы рассмотрим основные различия между Python 3 и Python 2 в следующем разделе.

Основные различия между Python 3 и Python 2

Как уже было сказано, в Python 3 нет обратной совместимости с Python 2 на уровне синтаксиса. Однако не все так плохо. Кроме того, не каждый модуль Python, написанный под версию 2.x, перестает работать в Python 3. Можно писать полностью кросс-совместимый код, который будет работать в обеих версиях без дополнительных инструментов или методов, но обычно это возможно только для простых приложений.

Почему это должно нас волновать

Несмотря на наше мнение о совместимости Python 2, которое мы высказали ранее в данной главе, нельзя просто взять и забыть об этой проблеме. Есть несколько действительно полезных пакетов, но в ближайшее время они вряд ли будут портированы.

Кроме того, иногда ограничения исходят от организации, в которой мы работаем. Уже имеющийся код может быть настолько сложным, что портировать его экономически нецелесообразно. Таким образом, даже если мы решили двигаться дальше и с этого момента пользоваться исключительно Python 3, сразу полностью отказаться от Python 2 все равно будет невозможно.

В наши дни трудно назвать себя профессиональным разработчиком, если не вносить свой вклад в деятельность сообщества. Таким образом, помочь разработчикам, пишущим открытый исходный код, внедрить совместимость Python 3

с существующими пакетами — отличный способ погасить моральный долг, возникший в результате использования последних. Конечно, это невозможно сделать, не зная различий между Python 2 и Python 3. Кстати, это также отличное упражнение для новичков в Python 3.

Основные синтаксические различия и распространенные ошибки

Документация Python — лучшее место, где можно почитать о различиях между версиями Python. Тем не менее для удобства читателей здесь перечислены наиболее важные различия. Это не отменяет того факта, что документация обязательна к прочтению тому, кто еще не знаком с Python 3 (docs.python.org/3.0/whatsnew/3.0.html).

Важнейшие нововведения Python 3 можно разделить на три группы:

- изменения синтаксиса, в которых одни элементы синтаксиса были удалены/изменены, а другие — добавлены;
- изменения в стандартной библиотеке;
- изменения типов данных и коллекций.

Изменения синтаксиса

Изменения синтаксиса, затрудняющие запуск кода, обнаружить легче всего, ведь код просто не сможет выполняться. Код Python 3, в котором используются новые элементы синтаксиса, не будет работать на Python 2, и наоборот. Элементы, удаленные из официального синтаксиса, сделают код Python 2 явно несовместимым с Python 3. Любая попытка запустить подобный код немедленно приведет к сбою интерпретатора, вызывая исключение `SyntaxError`. Ниже представлен пример «сломанного» скрипта из двух команд, ни одна из которых не будет выполнена из-за ошибки синтаксиса:

```
print("hello world")
print "goodbye python2"
```

Результат запуска скрипта на Python 3 выглядит следующим образом:

```
$ python3 script.py
  File "script.py", line 2
    print "goodbye python2"
                          ^
  SyntaxError: Missing parentheses in call to 'print'
```

Если говорить о новых элементах синтаксиса Python 3, то на перечисление всех различий уйдет много времени, и в каждой версии Python 3.x могут снова появиться новые элементы синтаксиса, которые будут так же несовместимы с более ранними версиями Python (даже если это уже Python 3.x). Наиболее важные из них рассмотрены в главах 2 и 3, так что нет необходимости перечислять их все здесь.

Список вещей, которые работали в Python 2 и вызывали синтаксические или функциональные ошибки в Python 3, гораздо короче. Ниже представлены наиболее важные несовместимые изменения:

- `print` уже не оператор, а функция, поэтому скобки обязательны;
- указание исключений изменилось с `except exc, var` на `except exc as var`;
- оператор сравнения `<>` был заменен на `!=`;
- `from module import *` (docs.python.org/3.0/reference/simple_stmts.html#import) уже допускается не только на уровне модуля и больше не допускается внутри функций;
- `from .[module] import name` — теперь единственный общепринятый синтаксис для относительного импорта. Весь импорт, не начинающийся с точки, интерпретируется как абсолютный;
- функция `sorted()` и метод списков `sort()` больше не принимают аргумент `cmp`, нужно использовать аргумент `key`;
- целочисленное деление на числа с плавающей точкой возвращает числа с плавающей точкой. Отсечение дробной части достигается за счет оператора `//`, например `1 // 2`. С числами с плавающей точкой это также работает: `5.0 // 2.0 == 2.0`.

Изменения в стандартной библиотеке

Критические изменения в стандартной библиотеке обнаружить чуть сложнее, чем изменения синтаксиса. В каждой последующей версии Python добавляются, улучшаются или полностью удаляются стандартные модули. Данный процесс был распространен и в старых релизах Python (1.x и 2.x), так что в Python 3 это не является чем-то из ряда вон. В большинстве случаев, в зависимости от модуля, который был удален или реорганизован (например, `urlparse` перемещен в `urllib.parse`), он будет вызывать исключения на время импорта только после интерпретации. Поэтому такие проблемы легко выявить. Чтобы быть уверенными, что будут обнаружены все подобные моменты, необходимо тестировать весь код. В некоторых случаях (например, при использовании лениво загруженных модулей) проблемы, обычно заметные во время импорта, не будут проявляться, пока какая-либо функция не обратится к «проблемному» модулю. Именно поэтому важно убедиться, что во время теста выполняется каждая строчка кода.

Лениво загруженные модули

Лениво загруженный модуль — это модуль, который не был загружен во время импорта. В Python операторы import могут быть включены в функции, поэтому импорт будет происходить при вызове функции, а не во время основного импорта. Иногда такая загрузка модулей может быть разумным решением, но в большинстве случаев это обходной путь для плохо разработанной конструкции модуля (например, чтобы избежать циклического импорта). Такой код считается «с душком», и подобных действий вообще следует избегать. Уважительной причины для ленивой загрузки модулей стандартной библиотеки нет. В хорошо структурированном коде весь импорт должен быть сгруппирован в верхней части модуля.

Изменения типов данных, коллекций, строковых литералов

Разница в том, как Python представляет типы данных и коллекции, особенно заметна и создает больше всего проблем, когда разработчик пытается сохранить совместимость или просто портирует существующий код на Python 3. В то время как несовместимый синтаксис или изменения стандартной библиотеки легко найти и часто легко исправить, изменения в коллекциях и типах бывают неочевидны или требуют большого объема монотонной работы. Перечень таких изменений будет весьма длинным, поэтому официальная документация — самый лучший справочник.

Тем не менее здесь мы поговорим о том, как в Python 3 рассматриваются строковые литералы, поскольку это одно из самых спорных изменений Python 3, несмотря на то что это очень хороший ход, прояснивший многое.

Все строковые литералы теперь имеют кодировку Unicode, а у литералов `bytestring` должен быть префикс `b` или `B`. В Python 3.0 и 3.1 старый префикс Unicode U (например, `u"foo"`) не принимается и вызывает синтаксическую ошибку. Отказ от него был основной причиной большинства споров. Стало очень трудно создать код, совместимый с различными ответвлениями Python, — в версии 2.x Python ссылался на эти префиксы при создании литералов Unicode. Данный префикс был возвращен обратно в Python 3.3, чтобы облегчить процесс интеграции, хотя в настоящее время в этом нет какого-либо синтаксического смысла.

Популярные инструменты и методы поддержания кросс-версионной совместимости

Поддержание совместимости между версиями Python — трудная задача. Она может добавить много дополнительной работы, в зависимости от размера проекта, но это тем не менее можно и нужно делать. Для пакетов, которые многократно используются во многих средах, это абсолютно необходимо. Пакеты с открытым исходным кодом без четко определенной и проверенной совместимости вряд ли

станут популярны, да и сторонний код, применяемый в пределах компании, тоже будет полезно протестировать в различных средах.

Следует отметить, что, хоть эта глава сосредоточена в основном на совместимости между различными версиями Python, данные подходы применяются для поддержания совместимости с внешними зависимостями, такими как различные версии пакетов, бинарные библиотеки, системы или внешние сервисы.

Весь процесс можно разделить на три основных направления, расположенных в порядке их важности:

- определение и документирование целевой оценки совместимости и управления совместимостью;
- тестирование во всех средах и версиях, совместимость с которым была заявлена;
- реализация совместимости кода.

Заявление о том, что считается совместимым, — наиболее важная часть всего процесса, поскольку дает пользователям и разработчикам возможность иметь ожидания и делать предположения о работе кода и о том, как он может измениться в будущем. Наш код можно задействовать в качестве зависимости в различных проектах, в которых тоже будет внедрено управление совместимостью, поэтому способность понимать его поведение очень важна.

В этой книге мы всегда пытаемся предоставить несколько возможностей выбора и не давать абсолютных рекомендаций по конкретным вариантам, но здесь будет одно из немногих исключений. Лучший способ определить, каким образом совместимость может измениться в будущем, — использовать правильный подход к нумерации версий *Semantic Versioning (semver)* (semver.org). Это широко принятый стандарт маркировки изменений в коде версии с помощью всего лишь трех цифр. В нем также содержатся несколько советов о том, как работать с политикой устаревания. Вот выдержка из него (под лицензией Creative Commons — CC BY 3.0).

Допустим, номер версии приложения выглядит как MAJOR.MINOR.PATCH. Тогда прибавляем единицу к номеру:

1) MAJOR-версии, если вносятся изменения, делающие текущий код несовместимым с предыдущей версией;
2) MINOR-версии, если изменения вносятся вместе с обратной совместимостью;
3) PATCH-версии, *если вы исправляете ошибки обратной совместимости*.

Дополнительные отметки предварительных версий и метаданных могут быть дополнением к формату MAJOR.MINOR.PATCH.

Когда дело доходит до проверки совместимости кода с каждой заявленной версией и в любой среде (в нашем случае — версии Python), он должен быть проверен в каждой комбинации. Конечно, это почти невозможно, если у проекта много зависимостей, поскольку количество комбинаций быстро растет с каждой новой

версией зависимости. Таким образом, как правило, ищут некий компромисс, чтобы на тестирование совместимости не уходило много времени. Подборка инструментов, призванная облегчить тестирование в так называемых матрицах, представлена в главе 12, в которой мы поговорим о процедуре тестирования в целом.

Преимущество использования проектов, которые следуют практике semver, заключается в том, что, как правило, тестировать нужно только крупные релизы, поскольку мелкие и патч-релизы гарантированно не имеют изменений без обратной совместимости. Конечно, это справедливо, только когда проект неукоснительно следует данной практике. К сожалению, ошибки случаются с каждым и несовместимые изменения возникают во многих проектах, даже в мелких патчах. Тем не менее нарушение объявленной semver строгой совместимости в мелких изменениях и патчах считается ошибкой, и ее нужно исправлять.

Реализация слоя совместимости — последний и наименее важный шаг процесса, если границы данной совместимости четко определены и тщательно протестированы. Тем не менее есть ряд инструментов и методов, которые должен знать каждый программист, занимающийся этим.

Основным является модуль `__future__`. Он портирует некоторые новые возможности в старые версии и принимает форму оператора импорта:

```
from __future__ import <feature>
```

Функции, предоставляемые оператором `future`, — это синтаксические элементы, которые не так уж легко обрабатывать различными способами. Данный оператор влияет только на тот модуль, где был использован. Ниже представлен пример интерактивной сессии Python 2.7, которая переносит литералы Unicode с Python 3.0:

```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> type("foo")  # Старые литералы
<type 'str'>
>>> from __future__ import unicode_literals
>>> type("foo")  # Теперь Unicode
<type 'unicode'>
```

Вот список всех доступных вариантов оператора `__future__`, которые должны знать разработчики, занимающиеся сопровождением:

- `division` — добавляет оператор деления Python 3 (PEP 238);
- `absolute_import` — заставляет каждую форму оператора `import` интерпретироваться «без точки», то есть как абсолютный импорт (PEP 328);

- `print_function` — заменяет оператор `print` функцией, то есть использование скобок становится обязательным (PEP 3112);
- `unicode_literals` — заставляет каждый строковый литерал интерпретироваться как литералы Unicode (PEP 3112).

Список всех доступных вариантов оператора `__future__` невелик и охватывает лишь несколько свойств синтаксиса. Другие вещи, подвергшиеся изменению, например синтаксис функции `metaclass` (о ней мы поговорим в главе 5), поддерживать намного сложнее. Этот оператор также не поможет надежной работе с реорганизациями стандартных библиотек. К счастью, существуют инструменты, которые позволяют получить последовательный фрагмент готового к использованию совместимого кода. Наиболее известный — это Six (pypi.python.org/pypi/six/), который обеспечивает сопровождение как одиночный модуль. Другой перспективный, но чуть менее популярный инструмент — модуль `future` (python-future.org/).

Иногда разработчики могут не захотеть включать дополнительные зависимости в небольшие пакеты. Часто используется дополнительный модуль, обычно именуемый `compat.py`, который собирает весь код совместимости. Ниже представлен пример таких модулей из проекта `python-gmaps` (github.com/swistakm/python-gmaps):

```
# -*- coding: utf-8 -*-
"""Этот модуль обеспечивает совместимость
   кода между разными версиями Python
"""
import sys

if sys.version_info < (3, 0, 0):
    import urlparse  # noqa

    def is_string(s):
        """Возвращает True, если значение является строкой"""
        return isinstance(s, basestring)

else:
    # Примечание: urlparse перемещен в urllib.parse в Python 3
    from urllib import parse as urlparse # noqa
    def is_string(s):
        """Возвращает True, если значение является строкой"""
        return isinstance(s, str)
```

Такие модули `compat.py` популярны даже в тех проектах, сопровождение которых зависит от Six (https://pypi.python.org/pypi/six/), поскольку это очень удобный способ хранения кода, позволяющий получить совместимость с различными версиями пакетов.

В следующем разделе мы рассмотрим, что такое CPython.

Не только CPython

Эталонная реализация интерпретатора Python называется *CPython* и, как следует из названия, полностью написана на языке C. Это всегда был C и, вероятно, будет еще очень долго. Данную реализацию выбирает большинство программистов на Python, поскольку она всегда идет в ногу со спецификациями языка и является интерпретатором, на котором протестировано большинство библиотек. Но, кроме C, интерпретатор Python был написан на нескольких других языках. Кроме того, существуют модифицированные версии интерпретатора CPython, доступные под разными названиями и адаптированные для некоторых нишевых приложений. Большинство из них сильно отстают от CPython, но позволяют использовать и продвигать язык в узкоспециализированных задачах.

В этом разделе мы обсудим некоторые из наиболее известных и интересных альтернативных реализаций Python.

Почему это должно нас волновать

Существует много реализаций Python. На «Вики»-странице Python по этой теме (wiki.python.org/moin/PythonImplementations) представлены десятки различных вариантов языка, диалектов или реализаций интерпретатора Python, созданных не на C. Одни из них реализуют лишь часть синтаксиса основного языка, функций и встроенных расширений, но есть почти полностью совместимые с CPython. Надо понимать, что, хотя некоторые из них — просто игрушка или эксперимент, большинство из них созданы для решения реальных проблем, которые было сложно или невозможно решить с помощью CPython.

Примеры таких проблем:

- запуск кода Python на встраиваемых системах;
- интеграция с кодом, написанным для фреймворков вроде Java или .NET или на разных языках;
- запуск кода Python в браузерах.

В следующих подразделах кратко описаны субъективно наиболее популярные и современные варианты, в настоящее время доступные для программистов на Python.

Stackless Python

Stackless Python преподносит себя как улучшенную версию Python. Он носит такое имя, поскольку позволяет избежать зависимости от стека вызова C и имеет свой собственный стек. Это, по сути, модифицированный код CPython, в котором также

добавлены новые функции, отсутствовавшие в ядре Python на момент создания Stackless. Наиболее важными из них являются микропотоки, управляемые интерпретатором, как дешевая и облегченная альтернатива обычным потокам, которые должны зависеть от ядра системы и планирования задач.

Последние доступные версии 2.7.15 и 3.6.6 реализуют Python 2.7 и 3.6 соответственно. Все дополнительные функции в версии Stackless показаны в качестве основы в фреймворке через встроенный модуль `stackless`.

Stackless — не самая популярная альтернатива реализации Python, но о ней стоит знать, поскольку некоторые из реализованных в ней идей сильно повлияли на сообщество. Функциональность переключения ядра была извлечена из Stackless и опубликована в качестве самостоятельного пакета под названием `greenlet`, в настоящее время лежащего в основе многих полезных библиотек и фреймворков. Кроме того, большинство из его функций были вновь реализованы в PyPy — еще одной реализации Python, о которой мы поговорим позже. Официальную онлайн-документацию по Stackless Python можно найти по адресу stackless.readthedocs.io, а «Вики»-проект — на github.com/stackless-dev/stackless.

Jython

Jython — это реализация на Java. Код компилируется в байт-код Java и позволяет разработчикам легко задействовать классы Java в модулях Python. Jython дает возможность использовать Python как скриптовый язык верхнего уровня для сложных прикладных систем, например J2EE. Он также открывает Java-приложениям путь в мир Python. Создание Apache Jackrabbit (хранилище документов API на основе JCR, jackrabbit.apache.org) является хорошим примером того, что можно сделать с помощью Jython.

Основные отличия Jython от CPython:

- сбор мусора на Java вместо подсчета ссылок;
- отсутствие *глобальной блокировки интерпретатора* (global interpreter lock, GIL) позволяет более эффективно использовать несколько ядер в многопоточных приложениях.

Основной недостаток данной реализации языка — отсутствие поддержки расширений Python на C, поэтому написанные на C расширения не будут работать на Jython.

Последняя доступная версия Jython — Jython 2,7, и она соответствует версии языка 2.7. По заявлению разработчиков, в ней реализовано почти все ядро стандартной библиотеки Python и используются те же регрессионные тесты. К сожалению, Jython 3.x так и не был выпущен, и проект можно смело считать мертвым.

Тем не менее Jython заслуживает хотя бы внимания, поскольку в свое время был уникальным явлением, которое значительно повлияло на другие реализации Python.

Официальная страница проекта: www.jython.org.

IronPython

IronPython — это объединение Python и .NET Framework. Проект поддерживается корпорацией Microsoft, где работают ведущие разработчики IronPython. Это довольно крутая реклама для продвижения языка. За исключением Java, .NET — одно из крупнейших сообществ разработчиков в Microsoft. Стоит также отметить, что Microsoft предоставляет набор бесплатных инструментов разработки, которые превращают Visual Studio в полноценную IDE для Python. Он распространяется в виде плагинов Visual Studio под названием *Python Tools for Visual Studio (PVTS)*, доступных с открытым исходным кодом на GitHub (microsoft.github.io/PTVS).

Последний стабильный релиз — версия 2.7.8, и она совместима с Python 2.7. В отличие от Jython, здесь мы можем наблюдать активное развитие обеих веток — и 2.x, и 3.x, хотя поддержка Python 3 до сих пор официально не выпущена. Несмотря на то что .NET работает в основном на Microsoft Windows, IronPython можно также запустить на macOS и Linux. Это реализовано с помощью Mono, кросс-платформенной реализации .NET с открытым исходным кодом.

Основные отличия и преимущества CPython, по сравнению с IronPython, заключаются в следующем:

- как и в Jython, отсутствие *глобальной блокировки интерпретатора (GIL)* позволяет более полно использовать несколько ядер в многопоточных приложениях;
- код, написанный на C# и других языках .NET, легко интегрируется в IronPython и наоборот;
- он может работать во всех основных браузерах при наличии Silverlight (хотя Microsoft обещает прекратить поддержку Silverlight в 2021 году).

Есть у IronPython и отрицательные стороны — он очень похож на Jython, поскольку не поддерживает API расширений Python/C. Это важно для разработчиков, которые хотели бы использовать пакеты вроде NumPy, основанные на C. Сообщество несколько раз пыталось внедрить поддержку API Python/C в IronPython или по крайней мере совместимость с пакетом NumPy, но, к сожалению, ни один проект не стал успешным.

Узнать больше о IronPython можно на официальной странице проекта ironpython.net.

PyPy

PyPy — вероятно, самая интересная альтернативная реализация Python, поскольку в ней Python переписан на Python. Интерпретатор PyPy написан на Python. В CPython есть код на C, который делает всю работу. Но в PyPy этот код написан на чистом Python.

Это значит, что вы можете изменить поведение интерпретатора во время выполнения, а также реализовать паттерны проектирования, которые в CPython реализовать сложно.

PyPy в настоящее время полностью совместим с Python 2.7.13, в то время как последняя версия PyPy 3 совместима с Python версии 3.5.3.

В прошлом PyPy был интересен больше из теоретических соображений и только тем, кто увлечен особенностями языка. Обычно он не использовался в продакшене, но со временем это изменилось. В настоящее время многие тесты показывают, что, как ни удивительно, PyPy часто работает намного быстрее, чем реализация CPython. У этого проекта есть собственный бенчмаркинг, в котором отслеживается эффективность всех версий, измеренная с помощью десятков различных критериев (см. speed.pypy.org). Это говорит о том, что PyPy с JIT работает, как правило, в несколько раз быстрее, чем CPython. Эти и другие особенности PyPy побуждают все больше и больше разработчиков использовать PyPy в их production-среде.

Основные отличия PyPy, по сравнению с реализацией CPython, заключаются в следующем:

- используется сбор мусора вместо подсчета ссылок;
- имеется встроенный компилятор JIT, который дает серьезные улучшения в производительности;
- используется Stackless на уровне приложения, заимствованный из Stackless Python.

Как и почти любой другой альтернативной реализации Python, PyPy не хватает полноценной официальной поддержки расширений Python на языке C. Тем не менее в ней есть хоть какая-то поддержка расширений C через подсистему CPyExt, хотя у той пока нет нормальной документации. Кроме того, сообщество постоянно пытается портировать NumPy на PyPy, поскольку это наиболее востребованная функция.

Официальную страницу проекта PyPy можно найти на сайте pypy.org.

MicroPython

MicroPython — одна из самых молодых альтернативных реализаций в данном перечне, так как ее первая официальная версия была выпущена 3 мая 2014 года. Кроме того, это одна из самых интересных реализаций. MicroPython — интерпре-

татор Python, который был оптимизирован для использования на микроконтроллерах, то есть в стесненных условиях. Небольшой размер и кое-какие оптимизации позволяют ему работать всего в 256 килобайтах кода и всего в 16 килобайтах оперативной памяти.

Протестировать этот интерпретатор можно на контроллерах BBC — это разрядные устройства и пайборды, ориентированные на обучение программированию и основам электроники.

Интерпретатор MicroPython написан на C99 (это стандарт языка C) и может быть построен для многих аппаратных архитектур, включая x86, x86-64, ARM, ARM Thumb и Xtensa. Он основан на Python 3, однако ввиду многих различий синтаксиса нельзя достоверно сказать о полной совместимости с любой версией Python 3.x. Это скорее диалект Python 3 с функцией `print()`, ключевыми словами `async/await` и многими другими функциями Python 3. Не стоит ожидать, что ваши любимые библиотеки Python 3 будут работать должным образом без дополнительных настроек.

Узнать больше о MicroPython можно на официальной странице проекта micropython.org.

Полезные ресурсы

Лучший способ знать все о состоянии Python — быть в курсе всего нового и читать тематические ресурсы. В Интернете их множество. Наиболее важные и очевидные из них уже упоминались ранее, но для порядка повторим:

- документация Python;
- каталог пакетов Python (Python Package Index, PyPI);
- PEP 0 — индекс Python Enhancement Proposals (PEP).

Другие ресурсы, такие как книги и учебные пособия, тоже полезны, но быстро теряют актуальность. Не устаревают ресурсы, активно обновляемые сообществом. Те немногие, которые стоит рекомендовать, представлены ниже.

- Awesome Python (github.com/vinta/awesome-python) включает список популярных пакетов и структур.
- r/Python (www.reddit.com/r/Python/) — сабреддит Python, на котором можно найти новости и интересные посты о Python, каждый день размещаемые многими членами сообщества Python.
- Python Weekly (www.pythonweekly.com) — популярная информационная рассылка, в которой каждую неделю появляются десятки новых интересных пакетов и ресурсов Python.

- Pycoder's Weekly (pycoders.com) — еще одна популярная еженедельная информационная рассылка с дайджестом новых пакетов и интересных статей. Контент там часто пересекается с Python Weekly, но иногда можно найти что-то уникальное, еще не опубликованное в другом месте.

  На этих сайтах можно найти множество дополнительных материалов.

Резюме

Данная глава была посвящена текущему состоянию Python и изменениям, которые происходили на протяжении всей истории этого языка. Мы начали с обсуждения того, как и почему Python изменяется, и описали основные результаты этого процесса, особенно различия между версиями Python 2 и 3. Мы научились работать с данными изменениями и узнали о некоторых полезных методах, позволяющих писать код, совместимый с различными версиями языка и его библиотек.

Затем мы по-другому взглянули на идею изменений в языке программирования. Рассмотрели несколько популярных альтернативных реализаций Python и поговорили об их основных отличиях от реализации CPython по умолчанию.

В следующей главе мы опишем современные способы создания повторяемых и последовательных сред разработки для программистов Python и обсудим два популярных инструмента для изоляции окружающей среды: `virtualenv` и контейнеры Docker.

2 Современные среды разработки на Python

Глубокое понимание выбранного языка программирования — основа профессионализма. Это касается любой технологии. Действительно, трудно создавать хорошее программное обеспечение, не умея работать с инструментами и методами, общепринятыми в сообществе. В Python нет ни одной функции, которой не было бы в каком-либо другом языке. Если сравнивать синтаксис, выразительность или производительность, то всегда найдется решение, которое окажется лучше в том или ином смысле. Но вот чем Python действительно выделяется, так это экосистемой, возникшей вокруг языка. Сообщество Python долгие годы оттачивало стандартные методы и библиотеки, которые помогают создавать более надежное программное обеспечение в кратчайшие сроки.

Наиболее очевидная и важная часть экосистемы — огромная коллекция бесплатных пакетов и пакетов с открытым исходным кодом, которые решают множество проблем. Написание нового программного обеспечения — всегда дорогостоящий и трудоемкий процесс. Возможность использовать уже готовый код значительно сокращает время разработки. Для некоторых компаний это единственный способ сделать проекты экономически выгодными.

Разработчики на Python вложили много усилий в создание инструментов и стандартов для работы с пакетами с открытым исходным кодом, написанными другими разработчиками, начиная с виртуальных окружений, усовершенствованных интерактивных оболочек и отладчиков и заканчивая программами, которые позволяют найти и проанализировать огромную коллекцию пакетов, имеющихся в *каталоге пакетов Python* (Python Package Index, PyPI).

В этой главе мы обсудим:

- установку дополнительных пакетов Python с использованием `pip`;
- изоляцию сред исполнения;
- venv — виртуальное окружение Python;
- изоляцию среды на уровне системы;
- популярные инструменты повышения производительности.

Технические требования

Скачать бесплатные инструменты виртуализации, о которых мы будем говорить в этой главе, можно со следующих сайтов:

- Vagrant: www.vagrantup.com;
- Docker: www.docker.com.

Ниже приведены пакеты Python, которые упоминаются в этой главе, их вы можете скачать с PyPI:

- `virtualenv`;
- `ipython`;
- `ipdb`;
- `ptpython`;
- `ptbdb`;
- `bpython`;
- `bpdb`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы с примерами кода для этой главы можно найти по адресу github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter2.

Установка дополнительных пакетов Python с использованием pip

Сегодня многие операционные системы поставляются с Python в качестве стандартного компонента. Большинство дистрибутивов Linux и UNIX на основе FreeBSD, NetBSD, OpenBSD или macOS поставляются с Python либо сразу «из коробки», либо из репозитория. Многие из них даже используют его в своих основных компонентах — на Python работают инсталляторы Ubuntu (Ubiquity), Red Hat Linux (Anaconda) и Fedora (опять же Anaconda). К сожалению, обычно предустановленной версией является Python 2.7, которая уже устарела.

Из-за популярности Python в качестве компонента операционной системы многие пакеты PyPI доступны и в виде нативных пакетов, управляемых такими инструментами, как `apt-get` (Debian, Ubuntu), `rpm` (Red Hat Linux) или `emerge` (Gentoo). Следует помнить, однако, что список доступных библиотек весьма ограничен и они в основном устарели по сравнению с PyPI. Поэтому для получения новых пакетов

в последней версии всегда нужно использовать `pip`, как было рекомендовано *Python Packaging Authority (PyPA)*. Несмотря на то что это независимый пакет, начиная с CPython 2.7.9 и 3.4, он по умолчанию идет в комплекте с каждой новой версией. Установить новый пакет очень просто:

```
pip install <имя-пакета>
```

Среди прочего `pip` позволяет устанавливать конкретные версии пакетов (с помощью команды `pip install package-name==version`) и обновлять их до последней доступной версии (используя переключатель `--upgrade`). Полное описание применения большинства инструментов командной строки, представленных в книге, можно легко получить, запустив команду с переключателем `-h` или `--help`. Ниже приведен пример сеанса, который демонстрирует наиболее часто используемые опции:

```
$ pip show pip
Name: pip
Version: 18.0
Summary: The PyPA recommended tool for installing Python packages.
Home-page: https://pip.pypa.io/
Author: The pip developers
Author-email: pypa-dev@groups.google.com
License: MIT
Location: /Users/swistakm/.envs/epp-3rd-ed/lib/python3.7/site-packages
Requires:
Required-by:

$ pip install 'pip>=18.0'
Requirement already satisfied: pip>=18.0 in (...)/lib/python3.7/sitepackages
(18.0)

$ pip install --upgrade pip
Requirement already up-to-date: pip in (...)/lib/python3.7/site-packages
(18.0)
```

Не всегда `pip` бывает доступен по умолчанию. В Python 3.4 и далее (а также в Python 2.7.9) его можно загрузить с помощью модуля `ensurepip`:

```
$ python -m ensurepip
Looking in links:
/var/folders/z6/3m2r6jgd04q0m7yq29c6lbzh0000gn/T/tmp784u9bct
Requirement already satisfied: setuptools in /Users/swistakm/.envs/epp-3rd
ed/lib/python3.7/site-packages (40.4.3)
Collecting pip
Installing collected packages: pip
Successfully installed pip-10.0.1
```

Самая актуальная информация о том, как установить `pip` в более старых версиях Python, доступна на странице документации проекта по ссылке pip.pypa.io/en/stable/installing/.

Изоляция сред выполнения

Можно использовать `pip` для установки систем пакетов. В UNIX-системах и в Linux для этого нужны права суперпользователя, так что фактический вызов будет выглядеть следующим образом:

```
sudo pip install <имя-пакета>
```

Обратите внимание: в ОС Windows это не требуется, поскольку в ней по умолчанию нет интерпретатора Python и Python на Windows, как правило, устанавливается пользователем вручную без привилегий суперпользователя.

Не рекомендуется выполнять установку общесистемных пакетов непосредственно из PyPI. Данное утверждение на первый взгляд может противоречить предыдущему о том, что PyPA рекомендует использовать `pip`, но тому есть серьезные причины. Как объяснялось ранее, Python часто является составной частью многих пакетов, доступных в репозиториях ОС, и на нем может работать много сервисов. В распределительных системах немало усилий тратится на выбор правильных версий пакетов для обеспечения совместимости. Очень часто в пакеты Python, доступные в репозиториях, включены также пользовательские патчи или намеренно старые версии, чтобы обеспечить совместимость с некоторыми другими компонентами системы. Принудительное обновление такого пакета с помощью `pip` до версии, которая нарушает обратную совместимость, может привести к критическим багам в ряде важных системных сервисов.

Делать подобные вещи даже на локальном компьютере в целях разработки не рекомендуется. Безрассудно использовать `pip` таким образом — почти всегда риск, в конечном итоге создающий проблемы, которые очень трудно отлаживать. Это не значит, что установка пакетов из PyPI строго запрещена, но делать это нужно сознательно и с пониманием риска.

К счастью, существует простое решение данной проблемы: изоляция среды. Есть различные инструменты, позволяющие изолировать среду выполнения Python на разных уровнях абстракции системы. Основная идея заключается в том, чтобы изолировать зависимости проекта от пакетов, которые нужны системным сервисам. Преимущества такого подхода заключаются в следующем.

- Это позволяет решить дилемму *«Проекту X нужна версия 1.x, но проекту Y необходима 4.x»*. Программист может работать над несколькими проектами с различными зависимостями без риска их влияния друг на друга.
- Проекты больше не ограничиваются версиями пакетов, которые установлены в распределительных системах хранилищ разработчика.
- Нет рисков поломки других системных сервисов, которые зависят от определенных версий пакетов, так как новые версии пакетов доступны только в изолированной среде.

- Список пакетов-зависимостей можно *заморозить* и легко воспроизвести на другом компьютере.

Если вы работаете параллельно над несколькими проектами, то быстро обнаружите, что невозможно сохранить их зависимости, не прибегая к какой-либо изоляции.

Сравнение изоляции на уровне приложений с изоляцией на уровне системы. Самый простой и облегченный подход к изоляции — использование виртуальных окружений на уровне приложений. Они выполняют изоляцию интерпретатора Python и пакетов, доступных внутри него. Такие окружения весьма просты в установке, и очень часто их хватает для обеспечения надлежащей изоляции в процессе разработки небольших проектов и пакетов.

К сожалению, их не всегда хватает, когда нужно обеспечить достаточную согласованность и воспроизводимость. Несмотря на то что программное обеспечение, написанное на Python, как правило, считается очень компактным, все равно есть шанс столкнуться с проблемами, которые возникают в специфических системах или даже конкретных распределениях таких систем (например, Ubuntu по сравнению с Gentoo). Это очень распространено в крупных и сложных проектах, особенно если они зависят от скомпилированных расширений Python или внутренних компонентов хостинга операционной системы.

В подобных случаях изоляция на уровне системы отлично дополнит ваш рабочий процесс. При таком подходе делается попытка повторить и изолировать полные операционные системы со всеми их библиотеками и важнейшими системными компонентами либо с классическими инструментами виртуализации системы (например, VMWare, Parallels и VirtualBox) или контейнерными системами (например, Docker и Rocket). Некоторые из доступных решений, позволяющие выполнить такую изоляцию, рассматриваются далее в этой главе.

venv — виртуальное окружение Python

Есть несколько способов изолировать среду выполнения Python. Самый простой и очевидный, хоть и трудный в сопровождении, — вручную изменить значения переменных среды `PATH` и `PYTHONPATH` и/или переместить бинарные исходники Python в другое, кастомизированное (настроеное специально для этого) место, где мы бы могли хранить зависимости проекта таким образом, что это изменило бы то, как Python распознает доступные пакеты. К счастью, есть инструменты, которые могут помочь в поддержании виртуальных окружений и установленных для них пакетов. В основном это `virtualenv` и `venv`. Они делают, по сути, то же самое, что мы будем делать вручную. Текущая стратегия зависит от конкретной реализации инструмента, но они, как правило, более удобны в использовании и могут дать дополнительные преимущества.

Создать новое виртуальное окружение можно с помощью следующей команды:

```
python3.7 -m venv ENV
```

Нужно заменить `ENV` на желаемое имя для нового окружения. Это создаст новый каталог `ENV` в текущем рабочем каталоге. Внутри появится несколько новых каталогов:

- `bin/` — здесь хранятся новый исполняемый файл Python и скрипты/исполняемые файлы других пакетов;
- `lib/` и `include/` — эти каталоги содержат вспомогательные файлы библиотек для нового Python в виртуальном окружении. Новые пакеты будут установлены в `ENV/Lib/pythonX.Y/site-packages/`.

Созданное новое окружение нужно активировать в текущем сеансе оболочки с помощью команды UNIX:

```
source ENV/bin/activate
```

Таким образом изменяется состояние текущих сессий оболочки благодаря воздействию на переменные окружения. Чтобы пользователь понимал, что он активировал виртуальное окружение, в подсказке появится приписка `(ENV)`. Приведем пример сеанса, который создает и активирует новое окружение:

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

Важно отметить, что `venv` полностью зависит от состояния, которое хранится в файловой системе. Она не дает каких-либо дополнительных возможностей и не позволяет отслеживать, какие пакеты должны быть установлены. Виртуальные окружения также не портируемы, и их нельзя перенести на другую машину. То есть новое виртуальное окружение создается заново для каждого нового развертывания приложения. Из-за этого пользователи `venv` часто хранят зависимости проекта в файле `requirements.txt` (общепринятое имя), как показано в следующем коде:

```
# Строки после решетки (#) рассматриваются как комментарии

# Строгие имена версий лучше для воспроизводимости
eventlet==0.17.4
graceful==0.1.1

# Для проектов, которые хорошо протестированы с различными
# версиями зависимостей, принимаются относительные спецификаторы
falcon>=0.3.0,<0.5.0

# Пакетов без версий следует избегать, если не требуется последняя версия
pytz
```

При наличии таких файлов все зависимости можно легко установить с помощью `pip`, поскольку он принимает файл зависимостей на выходе:

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

Важно помнить: файл требований не всегда идеальное решение, так как в нем есть только те зависимости, которые будут установлены. Из-за этого весь проект может без проблем функционировать в одних средах разработки и совсем не запускаться в других, если файл требований устарел и не соответствует фактическому состоянию среды. Конечно, есть команда `pip freeze`, выводящая все пакеты в текущей среде, но ее не следует применять бездумно. Эта команда выводит все пакеты и даже те из них, которые не задействованы в проекте, а установлены только для тестирования.

Примечание для пользователей Windows

В venv под Windows действует другое соглашение об именах во внутренней структуре каталогов. Нужно использовать Scripts/, Libs/, а также Include/ вместо bin/, lib/ и include/, чтобы лучше соответствовать соглашениям в области развития в этой операционной системе. Команды для активации/деактивации среды также отличаются: нужно применять ENV/Scripts/activate.bat и ENV/Scripts/deactivate.bat вместо source в скриптах activate и deactivate.

Устаревший скрипт pyvenv

Модуль venv включает дополнительный скрипт командной строки pyvenv. Начиная с Python 3.6, он был отмечен как устаревший, и его использование официально не рекомендуется, поскольку команда pythonX.Y -m venv явно говорит о том, какая версия Python будет применяться для создания новой среды, в отличие от скрипта pyvenv.

venv против virtualenv. Задолго до создания стандартного модуля библиотеки `pyenv` инструмент `virtualenv` был одним из самых популярных средств создания облегченных виртуальных окружений. Его название так и расшифровывается: виртуальное окружение, или среда (virtual environment). Он не входит в стандартный дистрибутив Python, поэтому его можно получить с помощью `pip`. Если вы хотите применять данный инструмент, то стоит его устанавливать во всей системе (используя `sudo` в системах на базе Linux и UNIX).

Мы бы рекомендовали использовать модуль `venv` вместо `virtualenv` всегда, когда это возможно. Он должен выбираться по умолчанию для проектов, ориентированных на версию Python 3.4 и выше. Применять `venv` в Python 3.3 может быть немного неудобно из-за отсутствия встроенной поддержки `setuptools` и `pip`. Для проектов, ориентированных на более широкий спектр сред выполнения Python

(в том числе альтернативные интерпретаторы версий 2.x), `virtualenv` — более удачный выбор.

В следующем разделе мы рассмотрим изоляцию среды на уровне системы.

Изоляция среды на уровне системы

В большинстве случаев реализация ПО выполняется быстро, поскольку разработчики по многу раз задействуют множество существующих компонентов. *«Не занимайтесь самокопированием»* — вот популярное правило и девиз многих программистов. Применение других пакетов и модулей с целью включить их в базу кода — лишь часть этой культуры. Еще *повторно используемыми компонентами* можно считать бинарные библиотеки, базы данных, системные сервисы, сторонние API и т. д. Даже целые операционные системы можно считать такими компонентами.

Бэкенд-сервисы веб-приложений — отличный пример того, сколь сложными могут быть такие приложения. Самый простой стек, как правило, состоит из нескольких слоев (начиная с самого нижнего), таких как:

- база данных или другой вид хранилищ;
- код приложения, реализованный на Python;
- серверы HTTP, такие как Apache или NGINX.

Конечно, такие стеки могут быть еще проще, но это очень маловероятно. На самом деле большие приложения зачастую настолько сложны, что трудно выделить отдельные слои. Такие приложения могут использовать множество различных баз данных, делиться на несколько независимых процессов, а также задействовать много других системных сервисов для кэширования, создания очередей, регистрации, открытия сервисов и т. д. К сожалению, нет никаких ограничений по сложности, и код, похоже, следует второму закону термодинамики.

Вот что действительно важно: не все элементы стека программного обеспечения можно изолировать на уровне среды исполнения Python. Будь то сервер HTTP, например Nginx, или СУБД наподобие PostgreSQL, они, как правило, доступны в различных версиях и системах. Без подходящих инструментов сложно убедиться, что вся команда разработчиков использует одни и те же версии компонентов. Теоретически возможно, что все программисты в команде, работающей над одним проектом, смогут установить одинаковые версии сервисов на свои рабочие станции. Но все эти усилия тщетны, если разработчики не используют ту же ОС, что будет в production-среде. Просто нереально заставить программиста променять любимую систему на нечто другое.

Серьезной проблемой все еще является портируемость. Не все сервисы будут работать в production-средах так же, как на машине разработчика, и это вряд ли

изменится. Даже поведение Python может отличаться в разных системах, несмотря на все те усилия, который сообщество приложило, чтобы сделать его кроссплатформенным. Как правило, поведение хорошо задокументировано и отмечается только в тех местах, которые непосредственно зависят от инициируемых системой вызовов, но полагаться на способность программиста помнить длинный список проблем совместимости — большая ошибка.

Популярное решение данной проблемы заключается в изоляции целых систем как среды приложения. Обычно это достигается за счет использования различных типов инструментов виртуализации системы. Виртуализация, конечно, снижает производительность, но у современных компьютеров, снабженных аппаратной поддержкой виртуализации, потери производительности, как правило, незначительны. А вот список возможных преимуществ довольно велик:

- среда разработки может точно соответствовать версиям системы и сервиса, используемым в продакшене, что помогает решить проблемы совместимости;
- определения для инструментов конфигурации системы, таких как Puppet, Chef, или Ansible (если они применяются), можно повторно использовать для настройки среды разработки;
- вновь прибывшие члены команды могут легко включиться в проект, если создание таких сред автоматизировано;
- программисты могут работать непосредственно с низкоуровневыми системными функциями, не всегда доступными в операционных системах, которые они используют для работы, например с *файловыми системами в пространстве пользователя (FUSE)*, недоступными в Windows.

В следующем подразделе мы рассмотрим виртуальные среды разработки, использующие Vagrant.

Виртуальные среды разработки, использующие Vagrant

В настоящее время Vagrant — один из самых популярных инструментов для программистов, позволяющий управлять виртуальными машинами в локальной разработке. Он предоставляет разработчику простой и удобный способ описания среды разработки со всеми зависимостями с привязкой к исходному коду вашего проекта. Доступен для Windows, Mac OS, а также в некоторых популярных дистрибутивах Linux (см. www.vagrantup.com). Vagrant не имеет дополнительных зависимостей. Он создает новые среды разработки в виде виртуальных машин или контейнеров. Конкретная реализация будет зависеть от выбора поставщиков виртуализации. Поставщиком по умолчанию является VirtualBox, который предоставляется в комплекте с программой установки Vagrant, однако доступны и дополнительные

провайдеры. Наиболее известные варианты: VMware, Docker, *Linux Containers (LXC)* и Hyper-V.

Наиболее важную конфигурацию Vagrant предоставляет в одном файле под названием `Vagrantfile`. Он должен быть независимым для каждого проекта. Ниже приведены его наиболее важные функции:

- выбор поставщика виртуализации;
- создание образа виртуальной машины;
- выбор способа инициализации;
- наличие общего хранилища между VM и хостом VM;
- наличие портов, которые передаются между VM и ее хостом.

Языком синтаксиса `Vagrantfile` является Ruby. В файле конфигурации есть удобный шаблон, позволяющий запустить проект, а также отличная документация, поэтому знание языка не требуется. Конфигурацию шаблона можно создать с помощью одной команды:

```
vagrant init
```

Эта команда создаст в текущем рабочем каталоге новый файл под названием `Vagrantfile`. Лучшее место для хранения данного файла, как правило, корневая папка проекта. Этот файл — конфигурация, которая создаст новую виртуальную машину с помощью поставщика по умолчанию и базового образа. Файл `Vagrantfile`, созданный командой `vagrant init`, содержит много комментариев, которые дают описание всей процедуры настройки.

Ниже приведен пример минимального файла `Vagrantfile` для среды разработки Python 3.7 на основе операционной системы Ubuntu со значениями по умолчанию, в которых среди прочего установлена переадресация порта 80 на случай, если вы захотите заниматься веб-разработкой на Python:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  # Для каждой среды Vagrant требуется контейнер.
  # Их можно найти на https://vagrantcloud.com/search.
  # Здесь мы используем версию Bionic системы Ubuntu с архитектурой x64.
  config.vm.box = "ubuntu/bionic64"

  # Создание отображения порта переадресации, которое позволяет получить
  # доступ к указанному порту на машине из порта на хост-машине
  # и разрешение доступа через 127.0.0.1 и отключение публичного доступа.
  config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip: "127.0.0.1"

  config.vm.provider "virtualbox" do |vb|
    # Отображение VirtualBox GUI при загрузке машины
    vb.gui = false
```

```
    # Настройка объема памяти на VM
    vb.memory = "1024"
  end
  # Активация с помощью скрипта оболочки
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install python3.7 -y
  SHELL
end
```

В предыдущем примере мы установили дополнение в системный пакет с помощью простого скрипта оболочки. Почувствовав, что `Vagrantfile` готов, вы можете запустить виртуальную машину, используя следующую команду:

```
vagrant up
```

Первый запуск может занять несколько минут, поскольку образ скачивается из Интернета. Каждый раз при запуске виртуальной машины будут выполнены процедуры инициализации, продолжительность которых зависит от выбора поставщика, образа и производительности вашей системы. Как правило, это занимает всего несколько секунд. После того как новая среда Vagrant будет запущена, разработчик сможет подключаться к ней через SSH, используя следующую команду:

```
vagrant ssh
```

Это можно сделать в любом месте в исходном дереве проекта ниже местоположения `Vagrantfile`. Для удобства разработчиков Vagrant будет обходить все каталоги выше текущего рабочего каталога пользователя в дереве файлов, найдет файл конфигурации и сопоставит его с соответствующим экземпляром виртуальной машины. Затем будет установлено надежное соединение с оболочкой, и со средой разработки можно будет взаимодействовать так же, как с обычной удаленной машиной. Единственное отличие состоит в том, что дерево всего исходного проекта (корнем считается местоположение `Vagrantfile`) доступно в файловой системе виртуальной машины в `/vagrant/`. Этот каталог автоматически синхронизируется с файловой системой, вследствие чего можно нормально работать в IDE или в любимом редакторе на хосте и относиться к сессии SSH на виртуальной машине Vagrant так же, как к обычной локальной сессии.

В следующем подразделе мы рассмотрим виртуальные среды, использующие Docker.

Виртуальные среды, использующие Docker

Контейнеры — альтернатива полноценным виртуальным машинам. Это облегченный инструмент виртуализации, в котором ядро и операционная система позволяют запускать несколько экземпляров изолированных пространств пользователя.

Работа ОС распределяется между контейнерами и хостом, что теоретически требует меньше затрат, чем при полной виртуализации. Такой контейнер содержит только код приложения и его зависимости системного уровня, но с точки зрения его работы выглядит как полностью изолированная среда.

Программные контейнеры получили свою популярность в основном благодаря Docker, который является одной из доступных реализаций. Docker позволяет описать контейнер в виде простого текстового документа под названием `Dockerfile`. Контейнеры из таких определений можно собирать и сохранять. Docker также поддерживает постепенные изменения, поэтому при добавлении в контейнер чего-то нового его не придется переделывать с нуля.

Попробуем сравнить контейнеризацию и виртуализацию.

Контейнеризация против виртуализации

Различные инструменты вроде Docker и Vagrant имеют некоторые общие функции, но главное различие между ними заключается в их предназначении. Vagrant, о чем уже было сказано ранее, создан в первую очередь как инструмент разработки. Он позволяет загружать целую виртуальную машину одной командой, но не дает упаковать такую среду и включить в состав чего-то отдельным пакетом и затем развернуть ее. А вот Docker создан именно для этой цели — подготовки готовых контейнеров, которые можно отправлять и развертывать как единый пакет. При грамотной реализации это может значительно улучшить процесс развертывания продукта. Поэтому применять Docker и аналогичные решения (например, Rocket) для разработки имеет смысл только в том случае, если такие контейнеры нужно будет использовать в процессе развертывания в продакшене.

Из-за некоторых нюансов реализации поведение сред, основанных на контейнерах и на виртуальных машинах, может различаться. Если вы решили использовать контейнеры для разработки, но при этом они не нужны в продакшене, то потеряется часть гарантий совместимости, из-за которых вы, собственно, и занимались изоляцией среды. Но если вы уже применяете контейнеры в вашей целевой production-среде, то следует воспроизводить условия продакшена. К счастью, Docker, который в настоящее время является наиболее популярным контейнером, предоставляет удивительный инструмент `docker-compose`, делающий управление местной контейнерной средой чрезвычайно легким.

Напишем наш первый Dockerfile.

Создание первого Dockerfile

Каждая среда на основе Docker начинается с Dockerfile. Это формат описания того, как создать образ Docker. Можно считать образы Docker чем-то аналогичным образам виртуальных машин. Это один файл (состоящий из многих слоев), который включает в себя все системные библиотеки, файлы исходного кода и другие зависимости, необходимые для выполнения приложения.

Каждый слой образа Docker описывается в Dockerfile с помощью одной команды в следующем формате:

```
ИНСТРУКЦИЯ аргументы
```

Docker поддерживает множество команд, но вам нужны основные из них, чтобы начать работу:

- `FROM <имя-образа>` — описывает базовый образ, на котором будет основан новый;
- `COPY <src> ... <dst>` — копирует файлы из локальнй сборки (обычно файлы проекта) и добавляет их в файловую систему контейнера;
- `ADD <src> ... <dst>` — работает аналогично `COPY`, но автоматически распаковывает архивы и позволяет вставлять в `<src>` ссылки;
- `RUN <команда>` — запускает заданные команды поверх предыдущих слоев и подтверждает изменения, которые эта команда внесла в файловую систему в качестве нового слоя образа;
- `ENTRYPOINT ["<executable>", "<param>", ...]` — настройка команды по умолчанию, которая будет работать в качестве контейнера. Если ни одна точка входа не указана в любом месте в слоях образа, то Docker по умолчанию выполняет `/bin/sh -c`;
- `CMD ["<param>", ...]` — определяет параметры по умолчанию для входных точек образа. Зная, что точка входа по умолчанию для Docker — это `/bin/sh -c`, эта команда может также принимать форму `CMD["<executable>", "<param>", ...]`, хотя рекомендуется определить целевой исполняемый файл непосредственно в инструкции `ENTRYPOINT` и использовать `CMD` лишь для аргументов по умолчанию;
- `WORKDIR <dir>` — устанавливает текущий рабочий каталог для инструкций `RUN`, `CMD`, `ENTRYPOINT`, `COPY` и `ADD`.

Для правильной демонстрации типичной структуры Dockerfile предположим, что хотим *задокерить* встроенный в Python веб-сервер, доступный через модуль `http.server` с кое-какими файлами, нужными для этого сервера. Структура наших файлов проекта может выглядеть следующим образом:

```
.
├── Dockerfile
├── README
└── static
    ├── index.html
    └── picture.jpg
```

Локально вы можете запустить `http.server` Python на HTTP-порте по умолчанию следующей простой командой:

```
python3.7 -m http.server --directory static/ 80
```

Данный пример, конечно, весьма тривиален, и использовать для этого Docker — все равно что колоть орехи кувалдой. Поэтому для целей данного примера представим: у нас в проекте есть много кода, генерирующего эти статические файлы.

Нам нужно доставить только их, а не код, который их генерирует. Кроме того, предположим, получатели нашего образа знают, как использовать Docker, но не знают, как работать с Python.

Итак, наша цель заключается в следующем:

- скрыть от пользователя некоторые сложности, особенно тот факт, что мы задействуем Python и встроенный в него HTTP-сервер;
- упаковать исполняемый файл Python 3.7 со всеми его зависимостями и статическими файлами в основной каталог проекта;
- выставить настройки по умолчанию для запуска сервера на порте 80.

С учетом всех этих требований наш Dockerfile может иметь следующий вид:

```
# Определим основной образ.
# "python" является официальным образом Python.
# Разумно использовать «тонкие версии»
# для других облегченных образов на основе Python
FROM python:3.7-slim

# Для того чтобы образ был чист, переключимся
# на выбранный рабочий каталог "/app/", который
# обычно используется для этой цели.
WORKDIR /app/

# Это наши статические файлы, скопированные
# из дерева проекта в рабочий каталог.
COPY static/ static/

# Мы бы запустили "python -m http.server" локально,
# так что сделаем его точкой входа.
ENTRYPOINT ["python3.7", "-m", "http.server"]

# Мы хотим брать файлы из каталога static/
# на порте 80 по умолчанию, так что установим это в качестве аргументов
# по умолчанию для встроенного в Python HTTP-сервера.
CMD ["--directory", "static/", "80"]
```

Рассмотрим, как запустить контейнеры.

Запуск контейнеров

Прежде чем запускать контейнер, нужно в первую очередь создать образ, определенный в Dockerfile. Это можно сделать с помощью следующей команды:

```
docker build -t <name> <path>
```

Аргумент `-t <name>` позволяет дать образу понятное имя. Это совершенно не обязательно, но без него вы не сможете легко сослаться на вновь созданный образ. Аргумент `<path>` определяет путь к папке, где находится Dockerfile. Предположим, мы уже выполнили команду из корневого каталога проекта, который представили

выше, и хотим дать образу имя `webserver`. Команда `docker build` после этого даст следующий вывод:

```
$ docker build -t webserver .
Sending build context to Docker daemon 4.608kB
Step 1/5 : FROM python:3.7-slim
3.7-slim: Pulling from library/python
802b00ed6f79: Pull complete
cf9573ca9503: Pull complete
b2182f7db2fb: Pull complete
37c0dde21a8c: Pull complete
a6c85c69b6b4: Pull complete
Digest:
sha256:b73537137f740733ef0af985d5d7e5ac5054aadebfa2b6691df5efa793f9fd6d
Status: Downloaded newer image for python:3.7-slim
 ---> a3aec6c4b7c4
Step 2/5 : WORKDIR /app/
 ---> Running in 648a5bb2d9ab
Removing intermediate container 648a5bb2d9ab
 ---> a2489d084377
Step 3/5 : COPY static/ static/
 ---> 958a04fa5fa8
Step 4/5 : ENTRYPOINT ["python3.7", "-m", "http.server", "--bind", "80"]
 ---> Running in ec9f2a63c472
Removing intermediate container ec9f2a63c472
 ---> 991f46cf010a
Step 5/5 : CMD ["--directory", "static/"]
 ---> Running in 60322d5a9e9e
Removing intermediate container 60322d5a9e9e
 ---> 40c606a39f7a
Successfully built 40c606a39f7a
Successfully tagged webserver:latest
```

После создания вы можете просмотреть список доступных образов с помощью команды:

```
$ docker images
REPOSITORY    TAG           IMAGE ID          CREATED           SIZE
webserver     latest        40c606a39f7a      2 minutes ago     143MB
python        3.7-slim      a3aec6c4b7c4      2 weeks ago       143MB
```

Поразительный размер образа контейнера

Вес простого образа Python в 143 Мбайт — это многовато, однако на самом деле беспокоиться не о чем. Для краткости мы задействовали базовый образ, который прост в применении. Есть и другие образы, размер которых был специально ужат, но они, как правило, для более опытных пользователей Docker. Кроме того, благодаря слоистой структуре образов Docker если вы применяете много контейнеров, то базовые слои можно кэшировать и использовать повторно, так что в конечном итоге вы не будете думать о размере.

Когда образ будет собран и помечен, вы сможете запустить контейнер с помощью команды `docker run`. Наш контейнер является примером веб-сервиса, поэтому мы должны дополнительно сказать Docker, что хотим открыть порты контейнера, связав их локально:

```
docker run -it --rm -p 80:80 webserver
```

Вот объяснение некоторых аргументов предыдущей команды:

- `-it` — это на самом деле две сопряженные опции: `-i` и `-t`. `-i` (от *interactive*) держит STDIN открытым, даже если процесс контейнера будет отсоединен, и `-t` (например, `tty`) выделяет псевдо-TTY для контейнера. Короче говоря, эти две опции позволяют увидеть живые логи от `http.server` и убедиться, что прерывание клавиатуры приведет к выходу из процесса. Он станет вести себя так же, как если мы запустим Python прямо из командной строки;
- `--rm` — дает Docker указание автоматически удалять контейнер при выходе;
- `-p 80:80` — дает Docker указание открыть порт 80, привязывая его к интерфейсу хоста.

Настройка сложных сред

Хотя использовать Docker довольно легко в простых проектах, все может усложниться, как только вы начнете применять его сразу в нескольких проектах. Бывает очень легко забыть о конкретных параметрах командной строки или о том, на каких образах надо открывать те или иные порты. Все становится очень сложно при наличии сервиса, который должен общаться с другими сервисами. Одиночные контейнеры должны содержать только один запущенный процесс.

Это значит, что вам не нужно будет устанавливать дополнительные инструменты мониторинга процессов, такие как Supervisor или Circus, а вместо этого следует создать несколько контейнеров, взаимодействующих друг с другом. Каждый сервис может использовать свой образ, обеспечивать различные варианты конфигурации и открывать порты, которые могут перекрывать или не перекрывать друг друга.

Лучший инструмент, который можно использовать для простых и сложных случаев, — это Compose. Он обычно распространяется с Docker, но в некоторых дистрибутивах Linux (например, Ubuntu) его может и не быть по умолчанию и его придется установить как отдельный пакет из репозитория пакетов. Compose — это мощная утилита командной строки под именем `docker-compose`, которая позволяет описывать мультиконтейнеры приложения с помощью синтаксиса YAML.

Compose ожидает, что в каталоге проекта находится специально названный файл `docker-compose.yml`. Пример такого файла для нашего предыдущего проекта может выглядеть следующим образом:

```
version: '3'

services:
  webserver:
    # Инструктирует Compose собирать образ
    # из локального каталога (.)
    build: .
    # Эквивалентно опции "-p" команды docker build
    ports:
    - "80:80"

    # Эквивалентно опции "-t" команды docker build
    tty: true
```

Если создать в проекте файл `docker-compose.yml`, то всю вашу прикладную среду можно запустить и остановить двумя простыми командами:

- `docker-compose up`;
- `docker-compose down`.

Полезные рецепты Docker для Python

Docker и контейнеры — столь обширная тема, что невозможно обсудить ее всю в одной главе этой книги. Compose позволит легко начать работать с Docker, не имея особого понимания, как он функционирует внутри. Если вы новичок в Docker, то вам стоит немного притормозить, взять документацию и вдумчиво почитать ее, чтобы эффективно использовать Docker и преодолеть некоторые из неизбежных сложностей.

Ниже приведены несколько простых советов и рецептов, позволяющих решить большинство стандартных проблем, с которыми вы, вероятно, рано или поздно столкнетесь.

Уменьшение размера контейнеров

Общая проблема новых пользователей Docker — размер образов контейнеров. Действительно, контейнеры занимают много пространства по сравнению с простыми пакетами Python, но совсем немного по сравнению с размером образов для виртуальных машин. По-прежнему очень часто разработчики размещают несколько сервисов на одной виртуальной машине, но при использовании контейнеров у вас обязательно должен быть отдельный образ для каждого сервиса. Это значит, что при большом количестве сервисов расходы могут стать заметными.

Ограничить размер образов можно двумя дополнительными методами.

- **Использовать базовый образ, который разработан специально для этой цели.** Alpine Linux — пример компактного варианта Linux, специально приспособленного для создания очень маленьких и облегченных образов Docker. Базовый

образ весит всего 5 Мбайт и включает элегантный менеджер пакетов, который позволяет сохранить компактность образа.

- **Принять во внимание характеристики наложения файловой системы Docker.** Образы Docker состоят из слоев, каждый из которых включает разницу в корневой файловой системе между собой и предыдущим слоем. Когда слой создан, размер образа уже не может быть уменьшен. Это значит, что если вам нужен системный пакет в качестве зависимости сборки и его можно позже удалить из образа, то вместо использования нескольких команд `RUN` будет лучше все делать в одной команде `RUN` с командами оболочки.

Эти два метода можно проиллюстрировать следующим Dockerfile:

```
# Здесь мы используем alpine для иллюстрации
# управления пакетами, так как ему не хватает Python
# по умолчанию. Для проектов Python в целом
# лучше выбрать python:3.7-alpine.
FROM alpine:3.7

# Добавить пакет python3, так как у образа alpine его нет по умолчанию.
RUN apk add python3

# Запуск нескольких команд в одной команде RUN.
# Пространство может быть использовано после "apk del py3-pip", потому что
# слой изображения создается только после выполнения всей инструкции.
RUN apk add py3-pip && \
    pip3 install django && \
    apk del py3-pip

# (...)
```

Обращение к сервисам внутри среды Compose

Сложные приложения часто состоят из нескольких сервисов, которые взаимодействуют друг с другом. Compose позволяет легко определить такие приложения. Ниже приведен пример файла `docker-compose.yml`, определяющего приложение как комбинацию двух сервисов:

```
version: '3'

services:
  webserver:
    build: .
    ports:
    - "80:80"
    tty: true

  database:
    image: postgres
    restart: always
```

Эта конфигурация определяет два сервиса:

- `webserver` — это основной контейнер сервиса приложения, образы которого взяты из локального Dockerfile;
- `database` — это контейнер базы данных PostgreSQL с официального образа Docker `postgress`.

Мы предполагаем, что сервис `webserver` хочет общаться с сервисом `database` по сети. Для настройки таких связей необходимо знать IP-адрес сервиса или имя хоста, чтобы их можно было использовать в качестве конфигурации приложения. К счастью, Compose — инструмент, который был разработан именно для таких сценариев, поэтому нам будет намного проще.

Всякий раз, когда вы запускаете среду с помощью команды `docker-compose up`, Compose создаст выделенную сеть Docker по умолчанию и будет регистрировать все сервисы в данной сети, используя их имена в качестве их имен хостов. Это значит, что сервис `webserver` может использовать `database:5432` для связи с базой данных (5432 — порт PostgreSQL по умолчанию), а также любые другие сервисы, чтобы Compose имел возможность доступа к конечной точке HTTP сервиса веб-сервера `http://webserver:80`.

Несмотря на то что имена хостов в Compose легко предсказуемы, нежелательно жестко прописывать любые адреса в приложении или его конфигурации. Лучше всего задавать их через переменные среды, которые приложение может считать при запуске. В следующем примере показано, как определить произвольные переменные среды для каждого сервиса в файле `docker-compose.yml`:

```
version: '3'

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

  database:
    image: postgres
    restart: always
```

Обмен данными между несколькими средами Compose

Если вы создаете систему, состоящую из нескольких независимых сервисов и/или приложений, то наверняка заходите сохранить свой код в нескольких независимых кодовых хранилищах (проектах). Файлы `docker-compose.yml` для каждого приложения

Compose, как правило, хранятся в одном и том же хранилище кода, где и код приложения. Сеть по умолчанию, которую Compose создает для одного приложения, изолирована от сетей других приложений. Итак, что вы можете сделать, если внезапно захотите, чтобы ваши независимые приложения общались друг с другом?

К счастью, такое намерение тоже легко реализуется с помощью Compose. Синтаксис файла `docker-compose.yml` позволяет определить именованную внешнюю сеть Docker как сеть по умолчанию для всех сервисов, определенных в этой конфигурации. Ниже приведен пример конфигурации, которая определяет внешнюю сеть с именем `my-interservice-network`:

```
version: '3'

networks:
  default:
    external:
      name: my-interservice-network
services:
  webserver:
    build: .
    ports:
    - "80:80"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432

  database:
    image: postgres
    restart: always
```

Такие внешние сети не управляются Compose, поэтому вам придется создать ее вручную с помощью команды `docker network create` следующим образом:

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

Сделав это, вы сможете использовать созданную внешнюю сеть в других файлах `docker-compose.yml` для всех приложений, которые должны иметь свои сервисы, зарегистрированные в той же сети. Ниже приведен пример конфигурации для других приложений, которые смогут взаимодействовать с сервисами `database` и `webserver` через `my-interservice-network`, даже если они не определены в том же файле `docker-compose.yml`:

```
version: '3'

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

```
services:
  other-service:
    build: .
    ports:
    - "80:80"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432
      - WEBSERVER_ADDRESS=http://webserver:80
```

В следующем разделе рассмотрим популярные инструменты повышения производительности.

Популярные инструменты повышения производительности

«Инструмент повышения производительности» — немного расплывчатый термин. С одной стороны, почти каждый пакет с открытым исходным кодом, появившийся в Интернете, является своего рода усилителем производительности, так как предоставляет готовые к использованию решения для некоторых проблем, и поэтому не нужно тратить время на изобретение велосипеда (в идеале). С другой стороны, можно сказать, что весь Python заточен на производительность, и это тоже, несомненно, верно. Почти все в данном языке и его сообществе словно предназначено для того, чтобы сделать разработку программного обеспечения как можно более продуктивной.

Поскольку написание кода — веселый процесс, многие программисты в свободное время пытаются создать инструменты, которые сделают его еще проще и еще веселее. Данный факт будет использоваться здесь в качестве основы для очень субъективного и ненаучного определения инструмента повышения производительности. Мы будем так называть ту часть программного обеспечения, которая делает разработку проще и веселее.

Инструменты повышения производительности сосредоточены главным образом на определенных элементах процесса разработки, таких как тестирование, отладка и управление пакетами, и не являются основными частями продуктов, которые разрабатываются. В ряде случаев эти инструменты могут даже не упоминаться нигде в кодовой базе проекта, несмотря на их ежедневное использование.

Ранее в этой главе мы уже обсуждали наиболее важные инструменты повышения производительности, `pip` и `venv`. В них есть пакеты для решения конкретных проблем, например профилирования и тестирования, и им посвящены отдельные

главы в данной книге. Текущий раздел рассказывает о других инструментах, которые действительно стоит упомянуть, но отдельная глава в этой книге под них не отведена.

Пользовательские оболочки Python — ipython, bpython, ptpython и т. д.

Python-программисты тратят много времени на интерактивные сессии интерпретатора. Это допустимо для тестирования небольших фрагментов кода, доступа к документации или даже отладки кода во время выполнения. Обычно интерактивная сессия Python довольно проста и не предоставляет большое количество функций, таких как автозаполнение или помощники по самоанализу кода. К счастью, оболочка Python по умолчанию легко поддается расширению и настройке.

Если вы часто работаете в интерактивной оболочке, то можете легко изменить ее поведение. Python при запуске считывает переменную среды `PYTHONSTARTUP` и ищет путь к пользовательским сценариям инициализаций. Некоторые дистрибутивы ОС, где Python является встроенным компонентом (например, Linux, macOS), могут быть предварительно настроены на выполнение загрузочного скрипта по умолчанию. Они обычно находятся в домашнем каталоге пользователя под названием `.pythonstartup`. Эти сценарии часто задействуют модуль `readline` (основанный на библиотеке Readline GNU) совместно с `rlcompleter` для того, чтобы обеспечить интерактивное автозаполнение и историю команд.

Если у вас нет скрипта запуска по умолчанию, то вы можете легко создать собственный. Базовый скрипт для истории команд и автозаполнения выглядит просто:

```
# Файл запуска python

import atexit
import os

try:
    import readline
except ImportError:
    print("Completion unavailable: readline module not available")
else:
    import rlcompleter
    # Автозаполнение
    readline.parse_and_bind('tab: complete')

    # Путь к файлу истории в домашнем каталоге пользователя
    # Можно использовать собственный путь
    history_file = os.path.join(os.environ['HOME'],
        '.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 вашем домашнем каталоге и назовите его `.pythonstartup`. Затем добавьте переменную `PYTHONSTARTUP` в вашу среду, используя путь к этому файлу.

Настройка переменной среды PYTHONSTARTUP

Если вы работаете под Linux или macOS, то самый простой способ — создать скрипт запуска в домашней папке. Затем нужно связать его с переменной среды `PYTHONSTARTUP`, которая задается в скрипте запуска оболочки. Например, в оболочках Bash и Korn используется файл `.profile`, в который вы можете вставить такую строку:

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

Если вы работаете под Windows, то легко установить новую переменную среды с правами администратора в настройках системы, а затем сохранить скрипт в общем месте, а не задействовать заданное пользователем местоположение.

Писать скрипт для `PYTHONSTARTUP` — хорошее упражнение, но создавать хорошие пользовательские оболочки в одиночку бывает сложно и немногие разработчики находят на это время. К счастью, существует несколько реализаций пользовательской оболочки Python, которые позволяют чрезвычайно упростить интерактивные сессии в Python.

IPython

Оболочка IPython (ipython.readthedocs.io/en/stable/overview.html) имеет встроенную расширенную командную оболочку Python. Среди функций, которые она предоставляет, наиболее интересны следующие:

- динамический анализ объектов;
- доступ к оболочке системы через командную строку;
- прямая поддержка профилирования;
- работа с отладочными средствами.

Теперь IPython является частью более крупного проекта под названием Jupyter, предоставляющего интерактивные заметки/ноутбуки с кодом, который можно в них же выполнять и который можно записать на разных языках.

bpython

Оболочка `bpython` (bpython-interpreter.org) позиционирует себя как крутой интерфейс для интерпретатора Python. Вот некоторые из ее функций, перечисленных на странице проекта:

- подсветка синтаксиса;
- построчное автозаполнение кода с отображением вариантов при вводе;
- список передаваемых параметров для любой функции Python;
- автоотступы;
- поддержка Python 3.

ptpython

Оболочка `ptpython` (github.com/jonathanslenders/ptpython/) — иной подход к современным оболочкам Python. Интересно в данном проекте то, что реализация основных функций доступна в виде отдельного пакета, называемого `prompt_toolkit` (от того же автора). Это позволяет легко создавать различные эстетически приятные интерактивные интерфейсы командной строки.

Эту оболочку часто по функциональности сравнивают с `bpython`, но основное отличие состоит в том, что она позволяет работать в режиме совместимости с IPython и в ее синтаксисе есть дополнительные функции, такие как `%pdb`, `%cpaste` и `%profile`.

Включение оболочек в собственные скрипты и программы

Иногда возникает необходимость встроить цикл *read-eval-print (REPL)*, похожий на интерактивную сессию Python, в ваше ПО. Это облегчает экспериментирование с кодом и его проверку «изнутри». Самый простой модуль, который позволяет эмулировать интерактивный интерпретатор Python, входит в стандартную библиотеку и называется `code`.

Скрипт, который запускает интерактивную сессию, состоит из одного импорта и вызова функции:

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

Вы можете легко внести в него незначительные изменения, например сменить значение ввода или добавить какие-нибудь сообщения, но для вещей покруче потребуется намного больше работы. Если вы хотите получить больше возможностей, например подсветку кода, завершение выполнения или прямой доступ к системной оболочке, то всегда лучше использовать что-то созданное кем-то ранее. К счастью,

все интерактивные оболочки, упомянутые выше, можно встроить в вашу программу так же легко, как модуль `code`.

Ниже приведены примеры того, как можно сослаться на все ранее упомянутые оболочки внутри вашего кода:

```
# 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())
```

Интерактивные отладчики

Отладка кода — неотъемлемая часть процесса разработки ПО. Многие программисты проводят большую часть своей жизни, используя только логи и операторы `print` в качестве основного средства отладки, но большинство профессиональных разработчиков предпочитают задействовать какой-нибудь отладчик.

Python поставляется с уже встроенным интерактивным отладчиком под названием `pdb` (см. docs.python.org/3/library/pdb.html). Его можно вызвать из командной строки в скрипте, чтобы Python запустил постмортем-отладку, если программа завершается аварийно:

```
python -m pdb script.py
```

Постмортем-отладка хоть и полезна, но не универсальна. Она полезна, только когда приложение завершается с исключением, если происходит ошибка. Часто некорректный код просто ведет себя неправильно, но не завершается ошибкой. В таких случаях можно установить пользовательские точки останова (брейкпойнты) в конкретных строках кода с помощью вот такой строки:

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

Это заставит интерпретатор Python начать сеанс отладки во время выполнения с этой строки.

Отладчик `pdb` очень полезен для отслеживания проблем и на первый взгляд может показаться знакомым тем, кто работал с *GNU Debugger (GDB)*. Поскольку Python — динамический язык, сессия отладки `pdb` часто бывает похожа на обычную сессию интерпретатора. Это значит, что разработчик не ограничивается отслеживанием выполнения кода, а может вызвать любой код и даже выполнить импорт модуля.

К сожалению, первый опыт работы с `pdb` может быть немного шокирующим из-за наличия коротких команд отладчика, таких как `h`, `b`, `s`, `n`, `j` и `r`. Если сомневаетесь, используйте команду `help pdb`, которую можно вводить во время сеанса отладчика, — она позволит получить немало дополнительной информации.

Сессия отладки в `pdb` выглядит очень просто и не дает дополнительных функций, таких как автозаполнение или подсветка кода. К счастью, в PyPI есть несколько пакетов, которые предоставляют такие функции из альтернативных оболочек Python, уже упоминавшихся выше. Наиболее известные примеры:

- `ipdb` — это отдельный пакет на основе `ipython`;
- `ptpdb` — отдельный пакет на основе `ptpython`;
- `bpdb` — идет в комплекте с `bpython`.

Резюме

Эта глава была целиком посвящена средам разработки для Python-программистов. Мы обсудили важность изоляции окружающей среды для проектов Python. Вы узнали о двух различных уровнях изоляции среды (уровень приложения и уровень системы), а также о нескольких инструментах, которые позволяют создавать их воспроизводимыми и целостными. В конце главы мы привели обзор нескольких инструментов, позволяющих улучшить экспериментирование с Python и отладку программ.

Теперь, когда в вашем арсенале есть все эти инструменты, вы готовы изучить следующие несколько глав, в которых мы обсудим особенности современного синтаксиса Python.

В следующей главе мы рассмотрим практические рекомендации по написанию кода на Python (идиомы языка) и приведем краткое описание отдельных элементов синтаксиса Python, которые могут быть новыми для «середнячков» Python, но знакомы тем, кто работал с более старыми версиями языка.

Кроме того, мы рассмотрим внутренние реализации CPython и их вычислительные сложности в качестве обоснования для этих идиом.

Часть II
Ремесло Python

В этой части представлен обзор текущего развития Python с точки зрения разработчика — того, кто зарабатывает на жизнь программированием и должен знать свои инструменты и язык вдоль и поперек и даже изнутри. Вы узнаете о новейших элементах синтаксиса Python и о том, как надежно и последовательно создавать качественное программное обеспечение.

3 Современные элементы синтаксиса — ниже уровня класса

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

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

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

В этой главе:

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

Технические требования

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

Встроенные типы языка Python

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

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

Посмотрим на строки и байты.

Строки и байты

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Байты или строка байтов: путаница в именах

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

Рассмотрим подробности реализации строк и байтов.

Детали реализации

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

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

Поговорим подробнее о конкатенации.

Конкатенация строк

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Рассмотрим форматирование f-строками.

Форматирование f-строками

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

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

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

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

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

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

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

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

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

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

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

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

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

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

В следующем подразделе мы рассмотрим коллекции языка.

Контейнеры

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

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

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

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

Списки и кортежи

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

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

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

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

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

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

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

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

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

Таблица 3.1

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

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

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

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

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

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

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

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

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

Списковое включение и изменение размера массива

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ниже приведен специальный пример этого синтаксического элемента:

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

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

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

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

Словари

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Таблица 3.2

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

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

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

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

простой способ представить это — с помощью целых чисел, поскольку их хеши совпадают с их значениями:

```
>>> {number: None for number in range(5)}.keys()
dict_keys([0, 1, 2, 3, 4])
```

Использование типов данных с другими правилами хеширования может показать, что порядок не сохраняется. Ниже представлен пример, выполненный в CPython 3.5:

```
>>> {str(number): None for number in range(5)}.keys()
dict_keys(['1', '2', '4', '0', '3'])
>>> {str(number): None for number in reversed(range(5))}.keys()
dict_keys(['2', '3', '1', '4', '0'])
```

Как было показано в предыдущем коде, в CPython 3.5 (а также в более ранних версиях) полученный порядок зависит и от хеширования объекта, и от порядка, в котором добавлялись элементы. На это не стоит полагаться, поскольку данная ситуация может меняться в различных реализациях Python.

А что насчет Python 3.6 и более поздних версий? Начиная с Python 3.6, интерпретатор CPython перешел на новое компактное представление словарей, которое занимает меньше памяти, а также как побочный эффект этой новой реализации сохраняет порядок. И если в Python 3.6 сохранение порядка было лишь побочным эффектом реализации, то в Python 3.7 эта функция официально объявлена в спецификации языка Python. Таким образом, начиная с Python 3.7, наконец можно полагаться на порядок вставки элементов словарей.

Параллельно с реализацией словарей CPython в Python 3.6 появилось еще одно изменение в синтаксисе, связанное с порядком элементов в словарях. Как определено в документе PEP 468 (см. https://www.python.org/dev/peps/pep-0468/), порядок именованных аргументов, полученных с помощью синтаксиса `**kwargs`, должен быть таким же, как в вызове функции. Данное поведение хорошо видно на следующем примере:

```
>>> def fun(**kwargs):
...     print(kwargs)
...
>>> fun(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> fun(c=1, b=2, a=3)
{'c': 1, 'b': 2, 'a': 3}
```

Однако эти изменения могут эффективно использоваться только в новейших версиях Python. Что нужно делать, если у вас есть библиотека, которая должна работать еще и на старых версиях Python, и в некоторых частях кода требуется сохранение порядка в словарях? Самый лучший вариант — применить тип, который явно сохраняет порядок элементов.

К счастью, в стандартной библиотеке Python в модуле `collections` есть упорядоченный словарь `OrderedDict`. Конструктор этого типа принимает в качестве аргумента инициализации итерируемый тип. Каждый элемент этого аргумента должен быть парой «ключ — значение», как показано в следующем примере:

```
>>> from collections import OrderedDict
>>> OrderedDict((str(number), None) for number in range(5)).keys()
odict_keys(['0', '1', '2', '3', '4'])
```

У этого типа также есть дополнительные функции, например извлечение элементов с обоих концов с использованием метода `popitem()` или перемещение указанного элемента на один из концов с помощью метода `move_to_end()`. Полный справочник по данной коллекции можно найти в документации по Python (см. docs.python.org/3/library/collections.html). Даже если вы планируете работать только в Python версии 3.7 или более новых версиях, что гарантирует сохранение порядка вставки элементов, тип `OrderedDict` все равно будет полезен. Он позволяет явным образом показать, что вам нужно именно сохранение порядка. Если вы определяете `OrderedDict` вместо простого `dict`, то становится очевидным, что в данном конкретном случае порядок вставки элементов имеет важное значение.

Последнее интересное замечание: в очень старых кодовых базах можно найти `dict` как примитивную реализацию множества, которая обеспечивает уникальность элементов. Несмотря на то что это работает правильно, стоит избегать подобных костылей, если вы не работаете в Python старше 2.3. Такое использование словарей расточительно с точки зрения затрат ресурсов. В Python есть встроенный тип `set`, предназначенный именно для этой цели. На самом деле он очень похожим образом реализуется в словари CPython, но в нем есть некоторые дополнительные функции, а также кое-какие оптимизации.

Множества

Множества — очень надежная структура данных, в основном полезная в ситуациях, когда порядок элементов не так важен, как их уникальность. Кроме того, они важны, когда необходимо продуктивно проверить эффективность, если элемент содержится в коллекции. Множества в Python являются обобщением математических множеств и представлены в виде встроенных типов в двух вариантах:

- `set()` — это изменяемое неупорядоченное конечное множество уникальных неизменяемых (хешируемых) объектов;
- `frozenset()` — это неизменяемое хешируемое неупорядоченное конечное множество уникальных неизменяемых (хешируемых) объектов.

Неизменяемость объектов `frozenset()` позволяет использовать их в качестве ключей словаря, а также других объектов `set()` и `frozenset()`. Изменяемый объ-

ект `set()` нельзя использовать подобным образом. Попытка сделать это приведет к выбрасыванию исключения `TypeError`, как в следующем примере:

```
>>> set([set([1,2,3]), set([2,3,4])])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

С другой стороны, инициализация множеств в следующем примере будет совершенно правильна и не приведет к выбрасыванию исключений:

```
>>> set([frozenset([1,2,3]), frozenset([2,3,4])])
{frozenset({1, 2, 3}), frozenset({2, 3, 4})}
>>> frozenset([frozenset([1,2,3]), frozenset([2,3,4])])
frozenset({frozenset({1, 2, 3}), frozenset({2, 3, 4})})
```

Изменяемые множества можно создавать тремя способами с помощью:

- функции `set()`, которая принимает в качестве аргумента итерируемый объект, например `set([0, 1, 2])`;
- генерации `{element for element in range(3)}`;
- литералов: `{1, 2, 3}`.

Обратите внимание: использование литералов и генерации множеств требует особой осторожности, так как методика очень похожа на работу со словарями. Кроме того, в Python не предусмотрено литералов для пустого множества, поскольку пустые фигурные скобки `{}` зарезервированы для пустых литералов словаря.

Детали реализации. Множества в CPython очень похожи на словари. По сути дела, они реализованы как словари с фиктивными значениями, где только ключи являются фактическими элементами коллекции. У множеств также не хватает значений в отображении, которые можно было бы оптимизировать.

Благодаря этому множества позволяют очень быстро добавлять, удалять и проверять существование элемента со средней сложностью $O(1)$. Тем не менее, поскольку реализация множеств в CPython опирается на аналогичную структуру хеш-таблицы, в худшем случае сложность для этих операций все еще будет $O(n)$, где n — текущий размер множества.

Другие детали реализации тоже сохраняются. Элемент, включаемый во множество, должен быть хешируемым, и если экземпляры пользовательских классов хешируются неправильно, то это негативно скажется на производительности.

Несмотря на концептуальное сходство со словарями, во множествах в Python 3.7 не сохраняется порядок элементов (ни в спецификации, ни в подробностях реализации CPython).

Рассмотрим дополнительные типы данных и контейнеры.

Дополнительные типы данных и контейнеры

В разделе *«Встроенные типы языка Python»* мы говорили в основном о типах данных, имеющих специальные литералы в синтаксисе Python. Это были типы, которые реализуются на уровне интерпретатора. Однако в стандартной библиотеке Python есть много дополнительных типов данных, которые удобно использовать там, где основные встроенные типы недорабатывают или где данные по своей природе требуют специализированной обработки (например, данные времени и даты).

Наиболее распространенными являются контейнеры данных, которые находятся в `collections`, и мы уже кратко упомянули два из них: `deque` и `OrderedDict`. Однако диапазон структур данных, доступных для Python-программистов, невероятно огромен, и почти в каждом модуле в стандартной библиотеке языка определены специализированные типы для обработки данных при возникновении различных проблем.

В этом разделе мы остановимся только на типах данных с самым широким потенциалом использования.

Специализированные контейнеры данных из модуля collections

У каждой структуры данных свои недостатки. Не существует такой коллекции, которая может решить любую проблему, и четырех основных типов коллекций (кортеж, список, множество и словарь) все-таки маловато. Это самые основные и важные коллекции, имеющие специальный синтаксис литералов. К счастью, стандартная библиотека Python дает гораздо больше возможностей с помощью встроенного модуля `collections`. Ниже приведены наиболее важные универсальные контейнеры данных, предоставляемые этим модулем:

- `namedtuple()` — функция для создания подклассов кортежей, чьи индексы доступны как именованные атрибуты;
- `deque` — двухсторонняя очередь, обобщение стеков и очередей с быстрым добавлением и извлечением на обоих концах;
- `ChainMap` — похожий на словарь класс для создания единого представления нескольких отображений;
- `Counter` — подкласс словаря для подсчета хешируемых объектов;
- `OrderedDict` — подкласс словаря, в котором сохраняется порядок добавления элементов;
- `defaultdict` — подкласс словаря, который заполняет недостающие значения с помощью определенной пользователем функции.

Подробная информация о выборе коллекций из модуля collections и несколько советов о том, как их стоит использовать, приведены в главе 14.

Символическое перечисление с модулем enum

Одним из особо удобных типов в стандартной библиотеке Python является класс `Enum` из модуля `enum`. Это базовый класс, позволяющий определять символические перечисления, близкие по концепции к перечисляемым типам, имеющимся во многих других языках программирования (C, C++, C#, Java и др.), которые часто обозначаются ключевым словом `enum`.

Чтобы определить перечисление в Python, нужно будет создать подкласс класса `Enum` и определить все элементы перечисления в качестве атрибутов класса. Ниже приведен 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 следующая номенклатура для `enum`:

- `enumeration` или `enum` — подкласс базового класса `Enum`. Здесь это был бы `Weekday`;
- `member` — атрибут, который можно определить в подклассе `Enum`. Здесь это `Weekday.MONDAY`, `Weekday.TUESDAY` и т. д.;
- `name` — имя атрибута подкласса `Enum`, который определяет элемент. Здесь это `MONDAY` для `Weekday.MONDAY`, `TUESDAY` для `Weekday.TUESDAY` и т. д.;
- `value` — значение, присвоенное атрибуту подкласса `Enum`, который определяет элемент. Здесь значение `Weekday.MONDAY` было бы `1`, для `Weekday.TUESDAY` — `2` и т. д.

Можно применить любой тип в качестве значения члена перечисления. Если член не имеет значения в коде, то можно даже использовать тип `auto()`, который будет заменен на автоматически сгенерированное значение. Вот предыдущий пример, переписанный с использованием `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 полезны в любом месте, где переменная может принимать конечное количество значений. Например, их можно применять для определения состояний объектов, как показано 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 RuntimeError(
                "Can't process order that has "
                "been already processed"
            )
        self.status = OrderStatus.PROCESSING
        ...
        self.status = OrderStatus.PROCESSED
```

Еще один случай использования перечислений — хранение выборки неуникальных вариантов. Это часто реализуется с использованием битовых флагов и битовых масок в языках, в которых распространены битовые манипуляции с числами, как в C. В Python это реализуется более выразительным и удобным способом с помощью `FlagEnum`:

```
from enum import Flag, auto

class Side(Flag):
    GUACAMOLE = auto()
    TORTILLA = auto()
    FRIES = auto()
    BEER = auto()
    POTATO_SALAD = auto()
```

Вы можете комбинировать эти флаги, используя битовые операции (операторы `|` и `&`) и тест на существование флага с помощью слова `in`. Вот некоторые примеры 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 имеют некоторое сходство со словарями и именованными кортежами, поскольку все они сопоставляют имена/ключи со значениями. Основное отличие заключается в том, что определение `Enum` неизменяемо и глобально. Его следует использовать всякий раз при наличии замкнутого множества возможных значений, которое не может изменяться динамически во время выполнения программы, и особенно если это множество определяется только один раз и на глобальном уровне. Словари и именованные кортежи являются контейнерами данных. Вы можете создать столько их экземпляров, сколько захотите.

В следующем разделе мы поговорим о расширенных элементах синтаксиса.

Расширенный синтаксис

Сложно объективно сказать, какой конкретный элемент синтаксиса является расширенным. Для целей настоящей главы мы будем считать таковыми те элементы, которые непосредственно не связаны с какими-либо конкретными встроенными типами данных и которые относительно трудно понять вначале. Наиболее распространенными функциями Python, сложными для понимания, являются:

- итераторы;
- генераторы;
- менеджеры контекста;
- декораторы.

Итераторы

Итератор — не что иное, как объект-контейнер, который реализует протокол итератора. Этот протокол состоит из двух методов:

- `__next__` — возвращает следующий элемент контейнера;
- `__iter__` — возвращает сам итератор.

Итераторы можно создать из последовательности с помощью встроенной функции `iter`. Рассмотрим пример:

```
>>> i = iter('abc')
>>> next(i)
'a'
>>> next(i)
```

```
'b'
>>> next(i)
'c'
>>> next(i)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Когда последовательность закончится, выбрасывается исключение `StopIteration`. Это делает итераторы совместимыми с циклами, так как данное исключение может служить сигналом конца итерации. При создании пользовательского итератора необходимо предоставить объекты с реализацией `__next__`, которая перебирает состояние объекта, и методом `__iter__`, возвращающим итерируемый объект.

Оба метода часто реализуются внутри одного и того же класса. Ниже приведен пример класса `CountDown`, который позволяет перебирать числа в сторону 0:

```
class CountDown:
    def __init__(self, step):
        self.step = step

    def __next__(self):
        """Возвращает следующий элемент"""
        if self.step <= 0:
            raise StopIteration
        self.step -= 1
        return self.step

    def __iter__(self):
        """Возвращает итератор"""
        return self
```

Предыдущая реализация класса позволяет ему перебрать самого себя. Это значит, что после перебора контента итерация будет исчерпана и не станет повторяться:

```
>>> count_down = CountDown(4)
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
3
2
1
0
end
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
end
```

При желании сделать итератор доступным для повторного использования всегда можно разделить его реализацию на два класса для того, чтобы отделить состояние итерации и фактические объекты итератора, как показано в следующем примере:

```
class CounterState:
    def __init__(self, step):
        self.step = step

    def __next__(self):
        """Изменение счетчика до нуля с шагом 1"""
        if self.step <= 0:
            raise StopIteration
        self.step -= 1
        return self.step

class CountDown:
    def __init__(self, steps):
        self.steps = steps

    def __iter__(self):
        """Возвращает итерируемое состояние"""
        return CounterState(self.steps)
```

Отделив итератор от его состояния, вы убедитесь, что итерация не будет заканчиваться:

```
>>> count_down = CountDown(4)
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
3
2
1
0
end
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
3
2
1
0
end
```

Итераторы сами по себе являются идеей низкого уровня, и программа может жить без них. Однако они лежат в основе гораздо более интересных элементов: генераторов.

Генераторы и операторы yield

Генераторы предоставляют элегантный способ написать простой и эффективный код для функций, которые возвращают последовательность элементов. Оператор `yield` позволяет приостановить выполнение функции и возвращает промежуточный результат. При этом контекст выполнения сохраняется и может быть возобновлен позже, если это необходимо.

Например, функцию, которая возвращает числа последовательности Фибоначчи, можно записать с помощью синтаксиса генератора. Следующий код — это пример, взятый из документа PEP 255:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b
```

Можно извлекать новые значения генераторов, как если бы они были итераторами, с помощью функции `next()` и циклов:

```
>>> fib = fibonacci()
>>> next(fib)
1
>>> next(fib)
1
>>> next(fib)
2
>>> [next(fib) for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Наша функция `fibonacci()` возвращает объект типа `generator` — специальный итератор, который умеет сохранять контекст выполнения. Его можно вызывать бесконечно, каждый раз получая очередной элемент последовательности. Синтаксис получается кратким, а бесконечность алгоритма не нарушает читабельность кода. В данном коде не должно быть способа сделать функцию останавливаемой. На самом деле подход работает аналогично тому, как работает функция генерации последовательности в псевдокоде.

Часто затраты на обработку одного элемента оказываются меньше затрат на хранение целых последовательностей. То есть такой метод делает программу более эффективной. Например, последовательность Фибоначчи является бесконечной, но при этом генератору не требуется бесконечное количество памяти, чтобы хранить все эти значения «одно за одним», и теоретически он может работать *до бесконечности*. Общий случай использования таких генераторов — потоковая передача буферов данных с генераторами (например, из файлов). Передачу можно приостановить, возобновить или полностью прервать на любом этапе обработки

данных, не прибегая к необходимости загружать целые наборы данных в память программы.

Модуль `tokenize` из стандартной библиотеки генерирует токены из потока текста, обрабатывая его построчно:

```
>>> import io
>>> import tokenize
>>> code = io.StringIO("""
... if __name__ == "__main__":
... print("hello world!")
... """)
>>> tokens = tokenize.generate_tokens(code.readline)
>>> next(tokens)
TokenInfo(type=56 (NL), string='\n', start=(1, 0), end=(1, 1), line='\n')
>>> next(tokens)
TokenInfo(type=1 (NAME), string='if', start=(2, 0), end=(2, 2), line='if
__name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=1 (NAME), string='__name__', start=(2, 3), end=(2, 11),
line='if __name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=53 (OP), string='==', start=(2, 12), end=(2, 14), line='if
__name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=3 (STRING), string='"__main__"', start=(2, 15), end=(2, 25),
line='if __name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=53 (OP), string=':', start=(2, 25), end=(2, 26), line='if
__name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=4 (NEWLINE), string='\n', start=(2, 26), end=(2, 27),
line='if __name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=5 (INDENT), string=' ', start=(3, 0), end=(3, 4), line='
print("hello world!")\n')
```

Здесь видно, что метод `open.readline` перебирает строки файла, а `generate_tokens` обрабатывает их, выполняя дополнительные действия. Генераторы могут также снизить сложность кода и повысить эффективность некоторых алгоритмов преобразования данных при условии, что процесс трансформации можно разделить на отдельные этапы обработки. Если каждый шаг обработки считать итератором, а затем объединить их в функцию верхнего уровня, то это позволит избежать использования больших, некрасивых и нечитаемых функций. Кроме того, это может дать живую обратную связь на протяжении всей цепочки обработки.

В следующем примере каждая функция выполняет некоторое преобразование последовательности. Затем они применяются поочередно. Каждый вызов обрабатывает один элемент и возвращает его результат:

```
def capitalize(values):
    for value in values:
         yield value.upper()

def hyphenate(values):
    for value in values:
        yield f"-{value}-"

def leetspeak(values):
    for value in values:
        if value in {'t', 'T'}:
            yield '7'
        elif value in {'e', 'E'}:
            yield '3'
        else:
            yield value

def join(values):
    return "".join(values)
```

Разделив ваш конвейер обработки данных на несколько самостоятельных шагов, вы сможете объединить их разными способами:

```
>>> join(capitalize("This will be uppercase text"))
'THIS WILL BE UPPERCASE TEXT'
>>> join(leetspeak("This isn't a leetspeak"))
"7his isn'7 a l337sp3ak"
>>> join(hyphenate("Will be hyphenated by words".split()))
'-Will--be--hyphenated--by--words-'
>>> join(hyphenate("Will be hyphenated by character"))
'-W--i--l--l-- --b--e-- --h--y--p--h--e--n--a--t--e--d-- --b--y--
--c--h--a--r--a--c--t--e--r-'
```

Простым должен быть код, а не данные

Лучше иметь много простых функций, которые обрабатывают последовательности значений, чем сложную функцию, вычисляющую одно значение за раз.

Еще одна важная особенность Python, касающаяся генераторов, — это возможность взаимодействовать с кодом, вызываемым функцией `next()`. Оператор `yield` становится выражением, а некоторые значения могут быть переданы через декоратор с помощью метода генератора `send()`:

```
def psychologist():
    print('Please tell me your problems')
    while True:
```

```
        answer = (yield)
        if answer is not None:
            if answer.endswith('?'):
                print("Don't ask yourself too much questions")
            elif 'good' in answer:
                print("Ahh that's good, go on")
            elif 'bad' in answer:
                print("Don't be so negative")
```

Ниже представлен пример сеанса работы с нашей функцией `psychologist()`:

```
>>> free = psychologist()
>>> next(free)
Please tell me your problems
>>> free.send('I feel bad')
Don't be so negative
>>> free.send("Why I shouldn't ?")
Don't ask yourself too much questions
>>> free.send("ok then i should find what is good for me")
Ahh that's good, go on
```

Метод `send()` действует аналогично функции `next()`, но оператор `yield` возвращает значение, переданное ему внутри определения функции. Следовательно, функция может изменять свое поведение в зависимости от клиентского кода. Чтобы завершить картину, используем два других метода: `throw()` и `close()`. Они позволяют вводить в генератор исключения:

- `throw()` — позволяет клиентскому коду выбрасывать любые исключения;
- `close()` — работает так же, но выбрасывает специфическое исключение, `GeneratorExit`. В этом случае функция генератора должна вызвать `GeneratorExit` или `StopIteration`.

Генераторы лежат в основе других концепций в Python, таких как сопрограммы и асинхронный параллелизм, которые будут рассмотрены в главе 15.

Декораторы

Декораторы были добавлены в Python для улучшения читаемости функций и методов. Декоратор — это просто любая функция, которая принимает на вход функцию и возвращает ее расширенный вариант. Раньше их использовали, чтобы можно было определить методы как методы класса или статические методы в заголовке определения. Без синтаксиса декораторов пришлось бы использовать разреженное и повторяющееся определение:

```
class WithoutDecorators:
    def some_static_method():
        print("this is static method")
    some_static_method = staticmethod(some_static_method)
    def some_class_method(cls):
        print("this is class method")
    some_class_method = classmethod(some_class_method)
```

Выделенный синтаксис декоратора короче и легче для понимания:

```
class WithDecorators:
    @staticmethod
    def some_static_method():
        print("this is static method")
    @classmethod
    def some_class_method(cls):
        print("this is class method")
```

Рассмотрим общий синтаксис и возможные реализации декораторов.

Общий синтаксис и возможные реализации

Декоратор — это, как правило, именованный вызываемый объект (лямбда-выражения не допускаются), который принимает один аргумент при вызове (это будет декорированная функция) и возвращает другой вызываемый объект. *Вызываемый объект* здесь преднамеренно используется вместо функции. Декораторы часто обсуждаются в рамках методов и функций, однако не ограничены ими. На самом деле все вызываемые объекты (любой объект, который реализует метод `__call__`, считается вызываемым) могут быть использованы в качестве декоратора, и часто возвращаемые ими объекты являются не простыми функциями, а экземплярами более сложных классов, которые реализуют собственные методы `__call__`.

Синтаксис декоратора — это просто синтаксический сахар. Рассмотрим следующий вариант использования декоратора:

```
@some_decorator
def decorated_function():
    pass
```

Этот вариант всегда можно заменить явным вызовом декоратора и переназначением функции:

```
def decorated_function():
    pass
decorated_function = some_decorator(decorated_function)
```

Однако последний вариант не слишком читабельный и понятный, если в одной функции используется несколько декораторов.

Декоратор не обязательно возвращает вызываемый объект!

На самом деле в качестве декоратора может быть использована любая функция, поскольку Python не устанавливает возвращаемый тип для декораторов. Таким образом, задействовать некую функцию в качестве декоратора, который принимает один аргумент, но не возвращает вызываемый объект, например строку, вполне допустимо с точки зрения синтаксиса. Однако это не сработает, если пользователь попытается вызвать объект, который был задекорирован таким образом. Эта часть синтаксиса декоратора создает поле для интересных экспериментов.

Как функция. Существует много способов написания пользовательских декораторов, но самый простой — написать функцию, которая возвращает вложенную функцию, обертывающую вызов исходной функции.

Обобщенный шаблон выглядит следующим образом:

```
def mydecorator(function):
    def wrapped(*args, **kwargs):
        # Действия, выполняемые перед
        # вызовом оригинальной функции
        result = function(*args, **kwargs)
        # Действия после выполнения функции
        # и результат
        return result
    # Возвращает задекорированную функцию
    return wrapped
```

Как класс. Декораторы почти всегда могут быть реализованы через функции, но бывают ситуации, когда лучше задействовать пользовательский класс. Это часто верно, когда декоратор нуждается в сложной параметризации или зависит от конкретного состояния.

Общий шаблон непараметризованного декоратора, определенного как класс, выглядит следующим образом:

```
class DecoratorAsClass:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):
        # Действия, выполняемые перед
        # вызовом оригинальной функции
        result = self.function(*args, **kwargs)
        # Действия после выполнения функции
        # и результат
        return result
```

Параметризация декораторов. В реальных сценариях использования часто возникает необходимость в применении декораторов, которые могут быть параметризованы. Когда в качестве декоратора служит функция, решение простое — нужен

второй уровень упаковки. Вот простой пример декоратора, который повторяет выполнение декорированной функции заданное количество раз при каждом вызове:

```
def repeat(number=3):
    """Повтор декорированной функции заданное количество раз.
    В результате возвращается последнее значение оригинальной функции.
    : number – количество повторов, по умолчанию равно 3.
    """
    def actual_decorator(function):
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(number):
                result = function(*args, **kwargs)
            return result
        return wrapper
    return actual_decorator
```

Определенный таким образом декоратор может принимать параметры:

```
>>> @repeat(2)
... def print_my_call():
...     print("print_my_call() called!")
...
>>> print_my_call()
print_my_call() called!
print_my_call() called!
```

Обратите внимание: даже если у параметризованных декораторов есть значения аргументов по умолчанию, круглые скобки после имени обязательны. Правильный способ использования предыдущего декоратора с аргументами по умолчанию выглядит следующим образом:

```
>>> @repeat()
... def print_my_call():
...     print("print_my_call() called!")
...
>>> print_my_call()
print_my_call() called!
print_my_call() called!
print_my_call() called!
```

Отсутствие круглых скобок приведет к ошибке при вызове декорированной функции:

```
>>> @repeat
... def print_my_call():
...     print("print_my_call() called!")
...
>>> print_my_call()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: actual_decorator() missing 1 required positional
argument: 'function'
```

Декораторы с сохранением самоанализа. Частая ошибка при использовании декораторов — отсутствие сохранения метаданных функции (обычно это строка документации и оригинальное название). Во всех предыдущих примерах данная проблема не решена. В них создается новая функция и возвращается новый объект, а оригинальный при этом никак не упоминается. Это затрудняет отладку задекорированных функций и нарушает работу большинства инструментов автодокументирования, поскольку оригинальные строки документации и сигнатуры функций исчезают.

Рассмотрим этот момент подробнее. Предположим, что у нас есть некий фиктивный декоратор, который ничего не делает, и какие-то другие функции, задекорированные им:

```
def dummy_decorator(function):
    def wrapped(*args, **kwargs):
        """Внутренняя документация обернутой функции"""
        return function(*args, **kwargs)
    return wrapped

@dummy_decorator
def function_with_important_docstring():
    """Это важная строка документации, ее терять не надо"""
```

Если мы исследуем `function_with_important_docstring()` в интерактивной сессии Python, то увидим, что первоначальное название и строка документации были потеряны:

```
>>> function_with_important_docstring.__name__
'wrapped'
>>> function_with_important_docstring.__doc__
'Internal wrapped function documentation.'
```

Правильное решение этой проблемы заключается в использовании декоратора `wraps()`, предоставляемого модулем `functools`:

```
from functools import wraps

def preserving_decorator(function):
    @wraps(function)
    def wrapped(*args, **kwargs):
        """Внутренняя документация обернутой функции"""
        return function(*args, **kwargs)
    return wrapped

@preserving_decorator
def function_with_important_docstring():
    """Это важная строка документации, ее терять не надо"""
```

Если декоратор определен именно таким образом, то все важные метаданные функции сохраняются:

```
>>> function_with_important_docstring.__name__
'function_with_important_docstring.'
>>> function_with_important_docstring.__doc__
'This is important docstring we do not want to lose.'
```

В следующем пункте рассмотрим использование декораторов.

Использование декораторов и полезные примеры

Поскольку интерпретатор загружает декораторы во время первого чтения модуля, их использование следует ограничить применимыми обертками. Если декоратор привязан к классу метода или сигнатуре функции, которую он декорирует, то должен быть превращен в обычный вызываемый объект, чтобы избежать сложностей. Часто бывает полезно группировать декораторы в специальных модулях, которые отражают их область применения, с целью упростить дальнейшую поддержку.

Общие шаблоны декораторов:

- проверка аргументов;
- кэширование;
- прокси;
- провайдер контекста.

Проверка аргументов. Проверка аргументов, которые принимает или возвращает функция, бывает полезна, когда функция выполняется в определенном контексте. Например, если функция будет вызываться через XML-RPC, то Python не сможет генерировать полную сигнатуру, как это делается в статически типизированных языках. Данная функция необходима для обеспечения возможности самоанализа, когда клиент XML-RPC запрашивает сигнатуры функций.

Протокол XML-RPC

Протокол XML-RPC представляет собой облегченный протокол удаленного вызова процедуры, в котором для кодирования вызовов используется XML через HTTP. Он часто применяется вместо SOAP для простого клиент-серверного обмена. В отличие от SOAP, предоставляющего страницу, на которой перечислены все вызываемые функции (WSDL), у XML-RPC нет каталога доступных функций. Было предложено расширение протокола, позволяющее выполнять открытие сервера API, и модуль xmlrpc реализует его (см. docs.python.org/3/library/xmlrpc.server.html).

Пользовательский декоратор может создавать такой тип сигнатуры. Кроме того, он может проверять входные и выходные данные на соответствие определенным параметрам сигнатуры:

```
rpc_info = {}

def xmlrpc(in_=(), out=(type(None),)):
    def _xmlrpc(function):
        # Объявление сигнатуры
        func_name = function.__name__
        rpc_info[func_name] = (in_, out)
        def _check_types(elements, types):
            """Подфункция проверки типов"""
            if len(elements) != len(types):
                raise TypeError('argument count is wrong')
            typed = enumerate(zip(elements, types))
            for index, couple in typed:
               arg, of_the_right_type = couple
                if isinstance(arg, of_the_right_type):
                    continue
                raise TypeError(
                    'arg #%d should be %s' % (index,
                       of_the_right_type))

        # Обернутая функция
        def __xmlrpc(*args):  # Ключевые слова не допускаются
            # Проверка входов
            checkable_args = args[1:] # Удаление self
            _check_types(checkable_args, in_)
            # Запуск функции
            res = function(*args)
            # Проверка входов
            if not type(res) in (tuple, list):
                checkable_res = (res,)
            else:
                checkable_res = res
            _check_types(checkable_res, out)

            # Проверка типа и функции успешна
            return res
        return __xmlrpc
    return _xmlrpc
```

Декоратор регистрирует функцию в глобальном словаре и хранит список типов аргументов и возвращаемых значений. Обратите внимание: этот пример был весьма упрощен, чтобы показать идею проверки аргументов в декораторе.

Пример использования выглядит следующим образом:

```
class RPCView:
    @xmlrpc((int, int))  # два int -> None
    def accept_integers(self, int1, int2):
        print('received %d and %d' % (int1, int2))
```

```
    @xmlrpc((str,), (int,))  # string -> int
    def accept_phrase(self, phrase):
        print('received %s' % phrase)
        return 12
```

В момент считывания конструктор класса заполняет словарь `rpc_infos` и может быть использован в конкретной среде, в которой проверяются типы аргументов:

```
>>> rpc_info
{'meth2': ((<class 'str'>,), (<class 'int'>,)), 'meth1': ((<class
 'int'>, <class 'int'>), (<class 'NoneType'>,))}
>>> my = RPCView()
>>> my.accept_integers(1, 2)
received 1 and 2
>>> my.accept_phrase(2)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 26, in __xmlrpc
  File "<input>", line 20, in _check_types
TypeError: arg #0 должен быть <class 'str'>
```

Кэширование. Кэширование в декораторе очень похоже на проверку аргументов, но фокусируется на тех функциях, внутреннее состояние которых не влияет на выходные данные. Каждый набор аргументов может быть связан с уникальным результатом. Подобный стиль характерен для *функционального программирования* и может использоваться, когда множество входных значений конечно.

Таким образом, кэширование в декораторе позволяет держать выходные данные вместе с аргументами, которые были необходимы для вычисления, и возвращать их непосредственно при последующих вызовах.

Подобное поведение называется *запоминанием*, и его довольно просто реализовать в качестве декоратора:

```
"""В этом модуле предусмотрены аргументы меморизации,
способные хранить кэшированные результаты
декорированной функции за заданный период времени.
"""
import time
import hashlib
import pickle

cache = {}

def is_obsolete(entry, duration):
    """Проверка актуальности записи в кэше"""
    return time.time() - entry['time']> duration

def compute_key(function, args, kw):
    """Вычисление ключа кэширования для значения"""
```

```
    key = pickle.dumps((function.__name__, args, kw))
    return hashlib.sha1(key).hexdigest()

def memoize(duration=10):
    """Декоратор меморизации по ключевому слову
       позволяет запомнить аргументы функции
       за заданное время
    """
    def _memoize(function):
        def __memoize(*args, **kw):
            key = compute_key(function, args, kw)

            # Помещено ли это в кэш?
            if (
                key in cache and
                not is_obsolete(cache[key], duration)
            ):
                # Если да и актуально,
                # то возвращает кэшированное значение
                print('we got a winner')
                return cache[key]['value']

            # Вычисление результата,
            # если кэш не был найден
            result = function(*args, **kw)
            # Сохранение результата на потом
            cache[key] = {
                'value': result,
                'time': time.time()
            }
            return result
        return __memoize
    return _memoize
```

Хеш-ключ `SHA` построен с использованием упорядоченных значений аргументов, и результат сохраняется в глобальном словаре. Хеш вычисляется с помощью ярлыка, замораживающего состояние всех объектов, которые передаются в качестве аргументов, гарантируя, что все аргументы подходят. Если в качестве аргумента используется поток или сокет, то возникает ошибка `PicklingError` (см. docs.python.org/3/library/pickle.html). Параметр длительности применяется для обнуления кэшированных значений, когда с момента последнего вызова функции прошло слишком много времени.

Ниже представлен пример использования запоминания в декораторе (при условии, что предыдущий фрагмент кода хранится в модуле `memoize`):

```
>>> from memoize import memoize
>>> @memoize()
... def very_very_very_complex_stuff(a, b):
```

```
...     # Если  компьютер не справляется,
...     # лучше остановить выполнение
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> @memoize(1) # Отключает кэш через 1 секунду
... def very_very_very_complex_stuff(a, b):
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> cache
{'c2727f43c6e39b3694649ee0883234cf': {'value': 4, 'time':
1199734132.7102251)}
>>> time.sleep(2)
>>> very_very_very_complex_stuff(2, 2)
4
```

Кэширование ресурсозатратных функций может значительно повысить общую производительность программы, но применять его следует с осторожностью. Кэшируемое значение также может быть привязано к функции вместо использования централизованного словаря для более эффективного управления данными кэша. Но в любом случае более эффективный декоратор будет задействовать специализированную библиотеку кэша и/или специализированный сервис кэширования с передовым алгоритмом кэширования. `Memcached` — хорошо известный пример такого сервиса кэширования и может легко использоваться с Python.

В главе 14 приведены подробная информация и примеры для различных методов кэширования.

Прокси. Прокси-декораторы применяются для маркировки и регистрации функций с глобальным механизмом. Например, уровень безопасности, ограничивающий доступ к коду в зависимости от текущего пользователя, может быть реализован с помощью централизованной проверки с соответствующим разрешением, которое запрашивает вызываемый объект:

```
class User(object):
    def __init__(self, roles):
        self.roles = roles
```

```
class Unauthorized(Exception):
    pass

def protect(role):
    def _protect(function):
        def __protect(*args, **kw):
            user = globals().get('user')
            if user is None or role not in user.roles:
                raise Unauthorized("I won't tell you")
            return function(*args, **kw)
        return __protect
    return _protect
```

Эта модель часто применяется в веб-фреймворках Python для определения безопасности при публикации ресурсов. Например, в Django есть декоратор, который защищает доступ к веб-ресурсам.

Ниже представлен пример случая, когда имя текущего пользователя хранится в глобальной переменной. Декоратор проверяет известные ему роли при вызове метода (предыдущий фрагмент кода хранится в модуле пользователей):

```
>>> from users import User, protect
>>> tarek = User(('admin', 'user'))
>>> bill = User(('user',))
>>> class RecipeVault(object):
...     @protect('admin')
...     def get_waffle_recipe(self):
...         print('use tons of butter!')
...
>>> my_vault = RecipeVault()
>>> user = tarek
>>> my_vault.get_waffle_recipe()
use tons of butter!
>>> user = bill
>>> my_vault.get_waffle_recipe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in wrap
__main__.Unauthorized: I won't tell you
```

Провайдер контекста. Декоратор — провайдер контекста. Он гарантирует, что функция будет работать в правильном контексте, либо выполняет код до и/или после запуска декорированной функции. Проще говоря, декоратор задает новую конкретную среду выполнения. Например, если элемент данных используется в нескольких потоках, то необходимо использовать блокировку, чтобы обеспечить ему защиту от множественного доступа. В декораторе эта блокировка может быть реализована следующим образом:

```
from threading import RLock
lock = RLock()
```

```
def synchronized(function):
    def _synchronized(*args, **kw):
        lock.acquire()
        try:
            return function(*args, **kw)
        finally:
            lock.release()
    return _synchronized

@synchronized
def thread_safe():  # Убеждаемся, что ресурсы выделяются
    pass
```

Контекстные декораторы часто заменяются менеджерами контекста (с оператором `with`), о которых мы также поговорим в этой главе.

Менеджеры контекста и оператор with

Оператор `try..finally` позволяет попытаться выполнить некий код, даже если возникает ошибка. Существует много случаев его использования, например:

- закрытие файла;
- снятие блокировки;
- создание временного кода;
- запуск защищенного кода в особой среде.

Оператор `with` выделяет эти случаи, тем самым предоставляя простой способ обернуть блок кода методами, определенными внутри менеджера контекста. Это позволяет вызывать некий код до и после выполнения блока, даже если этот блок выбрасывает исключение. Например, при работе с файлом часто делается так:

```
>>> hosts = open('/etc/hosts')
>>> try:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print(line.strip())
... finally:
...     hosts.close()
...
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
```

Данный пример является специфичным для Linux, поскольку в нем считывается файл host, расположенный в папке /etc/, но с тем же успехом это мог бы быть любой текстовый файл.

С помощью оператора `with` код можно переписать лаконичнее и чище:

```
>>> with open('/etc/hosts') as hosts:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print(line.strip())
...
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
```

В предыдущем примере в качестве менеджера контекста используется функция `open()`, которая гарантирует, что файл будет закрыт после выполнения цикла `for`, даже если в процессе выбрасывается исключение.

Ниже приведены общие элементы из стандартной библиотеки Python, совместимые с этим оператором, — классы из модуля `threading`:

- `threading.Lock`;
- `threading.RLock`;
- `threading.Condition`;
- `threading.Semaphore`;
- `threading.BoundedSemaphore`.

Общий синтаксис и возможные реализации

Простейший синтаксис оператора `with` выглядит следующим образом:

```
with context_manager:
    # Блок кода
    ...
```

Кроме того, если менеджер контекста определяет переменные контекста, то они могут храниться локально с помощью оператора `as`:

```
with context_manager as context:
    # Блок кода
    ...
```

Обратите внимание: можно использовать несколько менеджеров контекста одновременно:

```
with A() as a, B() as b:
    ...
```

Это эквивалентно вложенности, показанной ниже:

```
with A() as a:
    with B() as b:
        ...
```

В качестве класса. Любой объект, который реализует *протокол менеджера контекста*, можно использовать в качестве менеджера контекста. Этот протокол состоит из двух специальных методов:

- `__enter__(self)` — позволяет определить, что должно произойти перед выполнением кода, который был обернут менеджером контекста, и возвращает переменную контекста;
- `__exit__(self, exc_type, exc_value, traceback)` — позволяет провести кое-какую чистку после выполнения кода, обернутого менеджером контекста, и захватывает все исключения, выброшенные в процессе.

Вкратце выполнение оператора `with` можно представить таким образом.

1. Вызывается метод `__enter__`. Любое возвращаемое значение привязано к целевому объекту, указанному в операторе `as`.
2. Выполняется внутренний блок кода.
3. Вызывается метод `__exit__`.

Метод `__exit__` принимает три аргумента, которые заполняются при возникновении ошибки в блоке кода. Если ошибка не возникнет, то все три аргумента будут иметь значение `None`. При возникновении ошибки метод `__exit__()` не должен повторно вызывать ее, поскольку за это отвечает вызывающий объект. Так можно предотвратить выброс исключения, но будет возвращено `True`. Это открывает новые случаи применения, например, декоратора `contextmanager`, который мы увидим ниже. Но в большинстве ситуаций этот метод выполняет чистку так же, как оператор `finally`. Обычно он ничего не возвращает, независимо от того, что происходит в блоке.

Ниже приведен пример фиктивного менеджера контекста, реализующего данный протокол, чтобы показать, как это работает:

```
class ContextIllustration:
    def __enter__(self):
        print('entering context')

    def __exit__(self, exc_type, exc_value, traceback):
        print('leaving context')

        if exc_type is None:
            print('with no error')
        else:
            print(f'with an error ({exc_value})')
```

При запуске без выброшенных исключений вывод выглядит следующим образом (предыдущий фрагмент хранится в модуле `context_illustration`):

```
>>> from context_illustration import ContextIllustration
>>> with ContextIllustration():
```

```
...     print("inside")
...
entering context
inside
leaving context
with no error
```

Когда выбрасывается исключение, вывод выглядит так:

```
>>> from context_illustration import ContextIllustration
>>> with ContextIllustration():
...     raise RuntimeError("raised within 'with'")
...
entering context
leaving context
with an error (raised within 'with')
Traceback (most recent call last):
  File "<input>", line 2, in <module>
RuntimeError: raised within 'with'
```

В качестве функции — модуль contextlib. Использование классов кажется наиболее гибким способом реализации любого протокола языка Python, но может быть слишком шаблонным в простых случаях. В стандартную библиотеку был добавлен модуль, в котором есть помощники, упрощающие создание менеджеров контекста. Самая полезная часть — декоратор `contextmanager`. Это позволяет задействовать процедуры `__enter__` и `__exit__` внутри одной функции, разделенной оператором `yield` (обратите внимание, что это превращает функцию в генератор). Если переписать предыдущий пример с этим декоратором, то код будет выглядеть следующим образом:

```
from contextlib import contextmanager

@contextmanager
def context_illustration():
    print('entering context')

    try:
        yield
    except Exception as e:
        print('leaving context')
        print(f'with an error ({e})')
        # Исключение будет выброшено заново
        raise
    else:
        print('leaving context')
        print('with no error')
```

При выбрасывании исключения функция должна вновь вызвать его, чтобы передать далее. Обратите внимание: у модуля `context_illustration` могут быть

аргументы, если необходимо. Этот маленький помощник упрощает нормальный класс на основе API менеджера контекста так же, как генераторы с итераторами на основе класса.

У данного модуля есть еще четыре помощника:

- `closing(element)` — возвращает менеджер контекста, который вызывает метод элемента `close()` на выходе. Это полезно для классов, работающих с потоками и файлами;
- `supress(*exceptions)` — подавляет любое из указанных исключений, если они происходят в теле оператора `with`;
- `redirect_stdout(new_target)` и `redirect_stderr(new_target)` — перенаправляют вывод `sys.stdout` или `sys.stderr` любого кода внутри блока к другому файлу или файл-подобному объекту.

Рассмотрим функционально-стилевые особенности Python.

Функционально-стилевые особенности Python

Парадигма программирования — это очень важное понятие, позволяющее классифицировать различные языки программирования. Парадигма определяет конкретный способ мышления о моделях исполнения языка (определение того, как все работает) или о структуре и организации кода. Существует много парадигм программирования, но, как правило, они сгруппированы в две основные категории:

- *императивные парадигмы*, в которых программист в основном занимается состоянием программы, а сама она является определением того, как компьютер должен управлять ее состоянием для генерации ожидаемого результата;
- *декларативные парадигмы*, в которых программист занимается формальным определением задачи или свойств желаемого результата, но не думает о том, как вычислять этот результат.

Благодаря модели исполнения и вездесущим классам и объектам наиболее естественные для Python парадигмы — это объектно-ориентированное программирование и структурное программирование. Это также две наиболее распространенные императивные парадигмы программирования среди всех современных языков программирования. Однако Python считается языком многопрофильной парадигмы и содержит функции, которые типичны и для императивных, и для декларативных языков.

Одной из прекраснейших особенностей программирования на Python является то, что вы всегда можете посмотреть на программу с разных сторон. Всегда суще-

ствуют различные способы решения проблемы, а иногда наилучшим подходом оказывается не тот, который является наиболее очевидным. В ряде случаев такой подход требует использования декларативного программирования. К счастью, Python с его богатым синтаксисом и большой стандартной библиотекой имеет все инструменты для функционального программирования, а оно — одна из основных парадигм декларативного.

Функциональное программирование мы обсудим в следующем подразделе.

Что такое функциональное программирование

Функциональное программирование — парадигма, в которой программа — это в основном вычисление функций (в математическом смысле), а состояние программы не изменяется с помощью определенных действий. Чисто функциональные программы не предусматривают изменения состояния (побочных эффектов) и изменяемых данных. В Python функциональное программирование реализуется за счет использования сложных выражений и деклараций функций.

Один из лучших способов глубже понять общую концепцию функционального программирования — ознакомиться с его основными терминами.

- *Побочные эффекты*. Функция имеет «побочный эффект», если изменяет состояние программы за пределами своей локальной среды. Иными словами, побочный эффект — это любые наблюдаемые изменения за пределами области видимости функции, которые возникают в результате вызова функции. Примером таких побочных эффектов может быть изменение значения глобальной переменной, изменение атрибута или объекта, доступного за пределами области видимости функции, или сохранение данных в каком-либо внешнем сервисе. Побочные эффекты находятся в центре концепции объектно-ориентированного программирования, где экземпляры класса — это объекты, используемые для инкапсуляции состояния приложения, а методы — это функции, привязанные к этим объектам и используемые для изменения их состояния.
- *Ссылочная прозрачность*. Ссылочно-прозрачная функция или выражение могут быть заменены значениями, которые соответствуют входным данным, и при этом поведение программы не изменится. Таким образом, отсутствие побочных эффектов — необходимое условие для ссылочной прозрачности, но не каждая функция, не имеющая побочных эффектов, является ссылочно-прозрачной. Например, встроенная функция Python `pow(x, y)` ссылочно-прозрачна, поскольку не имеет побочных эффектов, и любую пару аргументов x и y можно заменить значением x^y. С другой стороны, метод `datetime.now()` типа `datetime` не имеет наблюдаемых побочных эффектов, но при каждом вызове возвращает разные значения. Таким образом, он является ссылочно-непрозрачным.

- *Чистые функции*. Чистой является функция, не имеющая каких-либо побочных эффектов и всегда возвращающая одинаковое значение для одного и того же набора входных параметров. Другими словами, это функция, которая является ссылочно-прозрачной. Любая математическая функция по определению чистая.
- *Функции первого класса*. Язык содержит такие функции, если функции на этом языке можно рассматривать как любое другое значение или сущность. Функции первого класса могут быть переданы в качестве аргументов другим функциям, являться возвращаемыми значениями и присваиваться переменным. Иными словами, язык, в котором есть функции первого класса, — это язык, относящийся к функциям как к сущностям первого класса. Функции в Python являются функциями первого класса.

Используя эти понятия, мы могли бы описать чисто функциональный язык как язык, содержащий чистые функции верхнего уровня, в которых отсутствуют неясные ссылки и побочные эффекты. Python, конечно же, не является чисто функциональным языком программирования, и было бы очень трудно представить полезную программу на Python, в которой задействованы только чистые функции без каких-либо побочных эффектов. В Python есть немало функций, в течение долгих лет доступные только в чисто функциональных языках, поэтому на Python можно писать большие фрагменты кода в чисто функциональном стиле, хоть язык в целом не является таковым.

В следующем подразделе рассмотрим лямбда-функции.

Лямбда-функции

Лямбда-функции — очень популярная концепция программирования, особенно в функциональном программировании. В других языках программирования лямбда-функции иногда называются анонимными функциями, лямбда-выражениями или функциональными литералами. Лямбда-функции — это анонимные функции, которые не привязываются к какому-либо идентификатору (переменной).

Лямбда-функция в Python может быть определена только с помощью выражений. Синтаксис такой функции выглядит следующим образом:

```
lambda <аргументы>: <выражение>
```

Лучший способ представить синтаксис лямбда-функций — это сравнить «обычную» и анонимную функции. Ниже показана простая функция, которая возвращает площадь круга заданного радиуса:

```
import math

def circle_area(radius):
    return math.pi * radius ** 2
```

В виде лямбда-функции это будет иметь следующий вид:

```
lambda radius: math.pi * radius ** 2
```

Лямбды в функциях анонимны, но это не значит, что их нельзя применить к любому идентификатору. Функции в Python — объекты первого класса, поэтому всякий раз, используя имя функции, вы на самом деле задействуете переменную, которая является ссылкой на объект функции. Как и любые другие функции, лямбда-функции — «граждане первого класса», поэтому их тоже можно отнести к новой переменной. После назначения переменной они будут неотличимы от обычных функций, за исключением некоторых атрибутов метаданных. Следующие фрагменты из интерактивных сессий интерпретатора демонстрируют это:

```
>>> 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>'
```

В следующем подразделе рассмотрим функции `map()`, `filter()` и `reduce()`.

map(), filter() и reduce()

Функции `map()`, `filter()` и `reduce()` — три встроенные функции, которые наиболее часто используются в сочетании с лямбда-функциями. Они широко применяются в функциональном программировании на Python, так как позволяют выполнять преобразования данных любой сложности, избегая побочных эффектов. В Python 2 все три функции были доступны по умолчанию и не требовали дополнительного

импорта. В Python 3 функция `reduce()` была перенесена в модуль `functools`, и его нужно импортировать.

Функция `map(fun, iterable, ...)` применяет аргумент функции к каждому элементу объекта `iterable`. Вы можете передать больше таких объектов функции `map()`. Если вы сделаете это, то функция `map()` будет получать элементы из всех итераторов одновременно. Функция `func` будет получать столько элементов, сколько итерируемых объектов доступно на каждом шаге. Если итерируемые объекты разных размеров, то функция `map()` остановится, пока не закончится кратчайший из них. Стоит помнить: функция `map()` не вычисляет весь результат сразу, но возвращает итератор так, что каждый полученный элемент можно вычислить, только когда это необходимо.

Ниже приведен пример использования функции `map()` для вычисления квадратов первых десяти положительных целых чисел, включая 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]
```

Ниже приведен пример использования функции `map()` для итераторов различных размеров:

```
>>> list(map(print, range(5), range(4), range(5)))
0 0 0
1 1 1
2 2 2
3 3 3
```

Функция `filter(function, iterable)` работает аналогично функции `map()`, оценивая входные элементы «один за одним». В отличие от `map()` функция `filter()` не преобразует входные элементы в новые значения, но позволяет отфильтровать те входные значения, которые соответствуют предикату, определяемому аргументом `function`. Ниже приведены примеры использования функции `filter()`:

```
>>> evens = filter(lambda number: number % 2 == 0, range(10))
>>> odds = filter(lambda number: number % 2 == 1, range(10))
>>> print(f"Even numbers in range from 0 to 9 are: {list(evens)}")
Even numbers in range from 0 to 9 are: [0, 2, 4, 6, 8]
>>> print(f"Odd numbers in range from 0 to 9 are: {list(odds)}")
Odd numbers in range from 0 to 9 are: [1, 3, 5, 7, 9]

>>> animals = ["giraffe", "snake", "lion", "squirrel"]
>>> animals_with_s = filter(lambda animal: 's' in animal, animals)
>>> print(f"Animals with letter 's' are: {list(animals_with_s)}")
Animals with letter 's' are: ['snake', 'squirrel']
```

Функция `reduce(function, iterable)` работает полностью противоположно функции `map()`. Вместо того чтобы брать элементы `iterable` и применять к каждому из них `function`, возвращая список результатов, она кумулятивно выполняет операции, указанные в `function`, ко всем элементам `iterable`. Рассмотрим пример вызова функции `reduce()` для вычисления суммы элементов в разных итерируемых объектах:

```
>>> 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
```

Один интересный аспект функций `map()` и `filter()` таков: они могут работать с бесконечными последовательностями. Понятно, что программа в данном случае будет работать вечно. Однако возвращаемые значения `map()` и `filter()` являются итераторами, и мы уже узнали в этой главе, что можем получить новые значения итераторов с помощью функции `next()`. Функция `range()`, которую мы использовали в предыдущих примерах, к сожалению, требует конечного входного значения, но модуль `itertools` предоставляет полезную функцию `count()`, которая позволяет считать от определенного числа в любом направлении до бесконечности. Следующий пример показывает, как можно использовать все эти функции, чтобы декларативно сформировать бесконечную последовательность:

```
>>> from itertools import count
>>> sequence = filter(
...     # Нам нужны только числа, кратные 3,
...     # но не кратные 2
...     lambda square: square % 3 == 0 and square % 2 == 1,
...     map(
...         # Все числа должны быть квадратами чисел
...         lambda number: number ** 2,
...         # Считаем до бесконечности
...         count()
...     )
... )
>>> next(sequence)
9
>>> next(sequence)
81
>>> next(sequence)
225
>>> next(sequence)
441
```

В отличие от функций `map()` и `filter()` функции `reduce()` приходится вычислять все элементы ввода, чтобы вернуть значение, поскольку она не дает промежуточных результатов. То есть она не может быть использована на бесконечных последовательностях.

Рассмотрим частичные объекты и функцию `partial()`.

Частичные объекты и функция partial()

Частичные объекты слабо связаны с концепцией частичных функций из математики. Частичной называется обобщенная математическая функция, в которой не приходится отображать все возможные значения входного сигнала (домен) на результаты. В Python частичные объекты могут служить для создания среза входных значений данной функции путем фиксации значений некоторых ее аргументов.

В предыдущих разделах мы использовали выражение `x ** 2`, чтобы получить квадрат значения `x`. В Python есть встроенная функция под названием `pow(x, y)`, которая позволяет вычислить любую степень любого числа. То есть `lambda x: x ** 2` является частичной функцией для `pow(x, y)`, так как мы ограничиваем диапазон значений `y` одним значением `2`. Функция `partial()` из модуля `functools` предоставляет альтернативный способ определения частичных функций, не прибегая к лямбда-функциям, которые иногда бывают громоздкими.

Допустим, теперь мы хотим создать несколько иную частичную функцию из `pow()`. Недавно мы генерировали квадраты последовательных чисел. Теперь сузим область других входных аргументов и предположим, что хотим сгенерировать ряд степеней двойки: 1, 2, 4, 8, 16 и т. д.

Сигнатура конструктора частичного объекта выглядит так: `partial(func, *args, **keywords)`. Частичный объект станет вести себя подобно `func`, но его входные аргументы будут предварительно заполнены из `*args` (начиная с самого левого) и `**keywords`. Функция `pow(x, y)` не поддерживает именованные аргументы, поэтому предварительно заполнить крайний левый аргумент `x` можно 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: не обязательно назначать функцию `partial` всякий раз, когда вы хотите зафиксировать аргумент. Для одноразовых функций этот метод так же хорош, как и лямбда-функции. Следующий пример показывает, как различные функции, которые были представлены в этой главе, можно использовать для создания простого генератора бесконечных степеней двойки без явного определения функции:

```
from functools import partial
from itertools import count
infinite_powers_of_2 = map(partial(pow, 2), count())
```

Модуль itertools — это кладезь помощников и утилит для работы с итераторами любого типа. В нем есть различные функции, которые среди прочего позволяют зацикливать контейнеры, группировать их содержимое, разделять итерируемые объекты на части и объединять их и т. д., и все функции в данном модуле возвращают итераторы. Если вам интересно функциональное программирование на Python, то вы должны обязательно познакомиться с этим модулем.

В следующем подразделе рассмотрим выражения генераторов.

Выражения генераторов

Это один элемент, который позволяет писать код в более функциональном стиле. Его синтаксис похож на тот, что используется со словарями, множествами и списочными литералами. Выражение генератора обозначается круглыми скобками, как показано в следующем примере:

```
(item for item in iterable_expression)
```

Выражения генератора можно использовать в качестве входных аргументов в любой функции, которая принимает итераторы. Данные выражения также позволяют использовать операторы `if` для фильтрации определенных элементов. Это значит, что вы сможете заменить сложночитаемые `map()` и `filter()` более читаемыми и компактными выражениями генератора. Сравним один из предыдущих примеров с выражением генератора, которое делает то же самое:

```
sequence = filter(
    lambda square: square % 3 == 0 and square % 2 == 1,
    map(
        lambda number: number ** 2,
        count()
    )
)
```

```
sequence = (
    square for square
    in (number ** 2 for number in count())
    if square % 3 == 0 and square % 2 == 1
)
```

Теперь поговорим об аннотациях функций и переменных.

Аннотации функций и переменных

Аннотации функций — одна из самых уникальных особенностей Python 3. Официальная документация гласит: *«Аннотации — это необязательные метаданные, информация о типах используемых функций, определяемых пользователем»*.

Но указание типа — еще не все. В Python и его стандартной библиотеке не существует ни одного элемента, у которого есть такие аннотации. Именно поэтому данная особенность уникальна — она не имеет никакого синтаксического значения. Аннотации могут быть определены для функции и вызваны во время ее выполнения. Что с ними делать — остается на усмотрение разработчика.

В следующих разделах рассмотрим общий синтаксис аннотаций и возможные способы их использования.

Общий синтаксис

Слегка модифицированный пример из документации Python показывает, как определить и получить аннотации функций:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     pass
...
>>> print(f.__annotations__)
{'return': <class 'str'>, 'eggs': <class 'str'>, 'ham': <class 'str'>}
```

Как видим, аннотации параметров определяются выражением, оценивающим значение аннотации с двоеточием перед ним. Возвращаемые аннотации определяются выражением между двоеточием, обозначающим конец оператора `def`, и литералом `->`, за которым следует список параметров.

Определенные аннотации доступны в атрибуте `__annotations__` объекта в виде словаря и могут быть извлечены во время выполнения приложения.

Тот факт, что любое выражение можно использовать в качестве аннотации и оно расположено в непосредственной близости аргументов по умолчанию, позволяет создавать некоторые запутанные определения функций, а именно:

```
>>> def square(number: 0<=3 and 1=0) -> (\
...     +9000): return number**2
>>> square(10)
100
```

Однако такое использование аннотаций не имеет никакой другой цели, кроме запутывания, а написать трудночитаемый и труднообслуживаемый код легко и без них.

Возможные способы применения

Несмотря на большой потенциал аннотаций, широкого применения они не нашли. В статье о новых функциях Python 3 (см. https://docs.python.org/3/whatsnew/3.0.html) говорится следующее: *«Цель состоит в том, чтобы поощрить экспериментирование с помощью метаклассов, декораторов или фреймворков»*.

С другой стороны, в документе PEP 3107, в котором официально были предложены аннотации функций, перечислен следующий набор возможных вариантов использования.

- Предоставление информации о вводе:
  - проверка типов;
  - отображение типов входных и выходных данных для функций;
  - перегрузка/генерация функций;
  - работа с иностранным языком;
  - адаптация;
  - использование предикатов логических функций;
  - отображение запроса базы данных;
  - определение параметров RPC.
- Иная информация:
  - документация параметров и возвращаемых значений.

Хотя аннотации появились вместе с Python 3, на данный момент все еще очень трудно найти какой-либо популярный и активно поддерживаемый пакет, в котором они бы использовались для чего-то еще, кроме проверки типов. Помимо статической проверки типов, аннотации функций по-прежнему годятся только для экспериментов — изначальной цели их добавления в первоначальную версию Python 3.

Статическая проверка типа с помощью mypy

Статическая проверка типа — это метод, позволяющий быстро найти возможные ошибки и дефекты в коде до его выполнения. Это нормальная черта компилируемых языков со статической типизацией. Python, конечно, не хватает таких встроенных функций, но есть сторонние пакеты, которые позволяют проводить в Python статический анализ типов, чтобы улучшить качество кода. Аннотации

функций и переменных в настоящее время лучше всего использовать в качестве подсказок для статической проверки типов. Ведущий пакет статической проверки на Python — это `mypy`. Он анализирует функции и аннотации, которые могут быть определены с помощью иерархии подсказок от типов модулей (см. PEP 484).

Лучшее в `mypy` то, что подсказки типов использовать необязательно. При наличии большой кодовой базы вам не придется аннотировать весь код, чтобы добиться пользы от проверки статического типа. Вы можете просто начать постепенно вводить аннотации в наиболее используемые фрагменты кода и со временем получить нужный результат. Кроме того, `mypy` поддерживается разработчиками на Python в виде typeshed-проекта. Typeshed (см. github.com/python/typeshed) — это набор библиотек заглушек со статическими определениями типов как для стандартной библиотеки, так и для многих популярных сторонних проектов.

Более подробную информацию о `mypy` и его консоли можно найти на официальной странице проекта в mypy-lang.org.

Иные элементы синтаксиса, о которых вы, возможно, не знаете

В Python есть непопулярные и редко задействуемые элементы. Это связано с тем, что они не слишком полезны. Как следствие, многие программисты на Python (даже с многолетним опытом) просто не знают об их существовании. В числе наиболее ярких примеров можно назвать:

- оператор `for… else…`;
- именованные аргументы.

Оператор for… else…

Использование оператора `else` после цикла `for` позволяет выполнить блок кода, если цикл закончился *естественным образом* (без оператора `break`):

```
>>> for number in range(1):
...     break
... else:
...     print("no break")
...
>>> for number in range(1):
...     pass
... else:
...     print("no break")
...
no break
```

Это позволяет избавиться от некоторых *сигнальных* переменных, в которых хранится информация о том, произошел ли экстренный выход из цикла. Код становится чище, но это может сбить с толку программистов, незнакомых с таким синтаксисом. Некоторые говорят, что подобное использование `else` противоречит здравому смыслу, но вот простой совет, который поможет вам вспомнить, как это работает, — `else` срабатывает после `for`, если не было `break`.

Именованные аргументы

Конструкция `for… else…` скорее курьезная, и мало кто из разработчиков стремится использовать ее, но есть и еще более экзотическая часть синтаксиса Python, которую стоило бы применять чаще. Это именованные аргументы.

Именованные аргументы появились в Python довольно давно, но их можно было использовать только в ряде встроенных функций или расширений, построенных с помощью API Python/C. Только начиная с Python 3.0, именованные аргументы стали официальным элементом синтаксиса языка, который можно использовать в любой сигнатуре функции. В сигнатурах функций каждый именованный аргумент определяется после одного символа *. То есть вы не можете передать значение в качестве позиционного аргумента.

Чтобы лучше понять, какую проблему позволяют решить такие аргументы, рассмотрим следующий набор функциональных заглушек, которые были определены без этой функции:

```
def process_order(order, client, suppress_notifications=False):
    ...

def open_order(order, client):
    ...

def archive_order(order, client):
    ...
```

Этот API довольно последователен. Четко видно, что каждая функция имеет по два аргумента, вероятно весьма важные для любой программы, в которой выполняется работа с заказами. Кроме того, в функции `process_order()` появился аргумент `suppress_notifications`. У него есть значение по умолчанию, то есть это, скорее всего, флаг, который можно включать и выключать. Мы не знаем, что делает данная программа, но можем предположить, как использовать эти функции. Самый простой пример выглядит следующим образом:

```
order = ...
client = ...

open_order(order, client)
process_order(order, client)
archive_order(order, client)
```

Все кажется ясным и простым. Тем не менее любопытный разработчик API увидит в таком интерфейсе нечто странное. При необходимости подавить уведомления в функции `process_order()` пользователь API может сделать это двумя способами:

```
process_order(order, client, suppress_notifications=True)
process_order(order, client, True)
```

Первый вариант лучше, так как семантика функции будет ясной и понятной. Здесь два крайних слева аргумента (`order` и `client`) лучше представить в виде позиционных аргументов, поскольку они связаны с содержательными именами переменных и их положение обычное для API. Значение аргумента `suppress_notifications` будет полностью утрачено, если мы представим его в виде простого `True`.

Что еще более тревожно, так это то, что такие нестрогие ограничения на использование API ставят разработчика в довольно неудобное положение, и он должен быть предельно осторожным при расширении существующих интерфейсов. Представим, что потребовалась возможность отмены платежа по запросу. Это можно сделать, добавив новый аргумент с именем `suppress_payment`. Изменение сигнатуры будет довольно простым:

```
def process_order(
    order, client,
    suppress_notifications=False,
    suppress_payment=False,
):
    ...
```

Для нас все ясно — `suppress_notifications` и `suppress_payment` должны быть поданы функции в качестве именованных, а не позиционных аргументов. А вот пользователям это может быть неясно. Очень скоро мы начнем видеть примерно такое:

```
process_order(order, client, True)
process_order(order, client, False)
process_order(order, client, False, False)
process_order(order, client, True, False)
process_order(order, client, False, True)
process_order(order, client, True, True)
```

Такой паттерн опасен еще по одной причине. Представьте, что кто-то хуже знает общую структуру API. Этот человек добавил новый аргумент, но не в конце списка аргументов, а перед другими аргументами, которые должны были быть использованы в качестве ключевых слов. Такая ошибка нарушит вообще все вызовы функций, поскольку аргументы поменяют позицию.

В больших проектах чрезвычайно трудно уберечь код от таких ошибок. Без достаточной защиты каждый неправильно употребленный вызов функций станет создавать проблемы в большом количестве. Лучший способ защитить свои функции

от такой «порчи» — явно указывать, какие аргументы следует использовать в качестве ключевых слов. Для нашего примера это будет выглядеть следующим образом:

```
def process_order(
    order, client,
    *,
    suppress_notifications=False,
    suppress_payment=False,
):
...
```

Резюме

В этой главе мы рассмотрели различные практические рекомендации по синтаксису, которые не имеют прямого отношения к классам Python и объектно-ориентированному программированию. Мы начали с анализа синтаксиса основных встроенных типов, а также технических деталей их реализации на интерпретаторе CPython.

Систематизировав наши базовые знания о встроенных типах Python, мы наконец перешли к действительно серьезным элементам Python: итераторам, генераторам, декораторам и менеджерам контекста. Конечно, мы не могли полностью обойтись без классов, так как все в Python является объектом и даже элементы синтаксиса, которые не являются объектно-ориентированными, в сути своей определены на уровне класса. Чтобы оправдать выбор названия этой главы, мы рассмотрели еще один важный аспект программирования на Python — функции языка, позволяющие программировать в функциональном стиле.

Чтобы закончить главу на более легкой ноте, мы рассмотрели менее известные, но от этого не менее важные и полезные особенности языка Python.

В следующей главе мы будем применять все изученное до сих пор, чтобы лучше понять объектно-ориентированные возможности Python. Мы подробнее рассмотрим понятие протоколов языка и порядок разрешения методов. Мы увидим, что каждая парадигма в Python имеет свое место, и поймем, как объектно-ориентированные элементы языка делают его более гибким.

4 Современные элементы синтаксиса — выше уровня класса

В этой главе мы сосредоточимся на современных элементах синтаксиса Python и подробнее поговорим о классах и объектно-ориентированном программировании. Однако мы не будем касаться темы объектно-ориентированных паттернов проектирования, так как им посвящена глава 17. В данной главе мы проведем обзор самых передовых элементов синтаксиса Python, которые позволят улучшить код ваших классов.

Модель классов Python, известная нам, сильно эволюционировала в процессе истории Python 2. Долгое время мы жили в мире, в котором две реализации парадигмы объектно-ориентированного программирования сосуществовали на одном языке. Эти две модели были названы *старым* и *новым стилем класса*. Python 3 положил конец этой дихотомии, так что разработчикам доступен только новый стиль. Но по-прежнему важно знать, как обе модели работали в Python 2, поскольку это поможет в случае, если потребуется портировать старый код и написать обратно совместимые приложения. Знание того, как изменилась объектная модель, также поможет понять, почему сейчас она такая, какая есть. Именно по этой причине в следующей главе мы частенько будем говорить о Python 2, несмотря на то что книга посвящена последним версиям Python 3.

В этой главе:

- протоколы языка Python;
- сокращение шаблонного кода с помощью классов данных;
- создание подклассов встроенных типов;
- доступ к методам из суперклассов;
- слоты.

Технические требования

Файлы с примерами кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter4.

Протоколы в языке Python — методы и атрибуты с двойным подчеркиванием

Модель данных Python определяет много специально именованных методов, которые могут быть переопределены в пользовательских классах и тем самым расширять их синтаксис. Вы можете узнать эти методы по обрамляющему их названия *двойному подчеркиванию* — таково соглашение по наименованию для данных методов. Из-за этого они иногда называются *dunder* (сокращение от double underline — «двойное подчеркивание»).

Наиболее распространенный и очевидный пример — метод `__init__()`, который используется для инициализации экземпляра класса:

```
class CustomUserClass:
    def __init__(self, initiatization_argument):
        ...
```

Эти методы, определенные отдельно или в комбинации, представляют собой так называемые языковые протоколы. Если объект реализует конкретные протоколы языка, то становится совместимым с конкретными частями синтаксиса Python. В табл. 4.1 приведены наиболее важные протоколы языка Python.

Таблица 4.1

Протокол	Методы	Описание
Протокол вызываемых объектов	__call__()	Объекты можно вызывать с помощью скобок: instance()
Протоколы дескрипторов	__set__(), __get__() и __del__()	Позволяют манипулировать паттерном атрибутов доступа в классах (см. подраздел «Дескрипторы» на с. 142)
Протокол контейнеров	__contains__()	Позволяет проверить, содержит ли объект некое значение через ключевое слово in: value in instance
Протокол итерируемых объектов	__iter__()	Позволяет объектам быть итерируемыми и использоваться для цикла for: for value in instance: ...

Протокол	Методы	Описание
Протоколы последовательности	__len__(), __getitem__()	Позволяют организовать индексацию объектов через синтаксис квадратных скобок и определять их длину с помощью встроенной функции: item = instance[index] length = len(instance)

Это наиболее важные протоколы языка с точки зрения данной главы. Полный список, конечно, гораздо длиннее. Например, в Python есть более 50 таких методов, которые позволяют эмулировать числовые значения. Каждый из этих методов коррелирует с конкретным математическим оператором и поэтому может рассматриваться как отдельный протокол языка. Полный список всех методов с двойным подчеркиванием можно найти в официальной документации модели данных Python (см. docs.python.org/3/reference/datamodel.html).

Языковые протоколы — основа концепции интерфейсов в Python. Одна из реализаций интерфейсов Python — это абстрактные базовые классы, которые позволяют задать произвольный набор атрибутов и методов в определении интерфейса. Такие определения интерфейсов в виде абстрактных классов могут впоследствии служить для проверки совместимости данного объекта с конкретным интерфейсом. Модуль `collections.abc` из стандартной библиотеки Python включает набор абстрактных базовых классов, которые относятся к наиболее распространенному протоколу языка Python. Более подробную информацию об интерфейсах и абстрактных базовых классах см. в пункте «Интерфейсы» на с. 530.

То же соглашение об именах применяется для определенных атрибутов пользовательских функций и хранения различных метаданных об объектах Python. Рассмотрим эти атрибуты:

- `__doc__` — перезаписываемый атрибут, который содержит документацию функции. По умолчанию заполняется функцией `docstring`;
- `__name__` — перезаписываемый атрибут, содержащий имя функции;
- `__qualname__` — перезаписываемый атрибут, который содержит полное имя функции, то есть полный путь к объекту (с именами классов) в глобальной области видимости модуля, в котором определен объект;
- `__module__` — перезаписываемый атрибут, содержащий имя модуля, к которому принадлежит функция;
- `__defaults__` — перезаписываемый атрибут, который содержит значения аргументов по умолчанию, если у функции есть таковые;
- `__code__` — перезаписываемый атрибут, содержащий код объекта компиляции функции;

- `__globals__` — атрибут только для чтения, который содержит ссылку на словарь глобальных переменных сферы действия этой функции. Сфера действия — пространство имен модуля, где определена эта функция;
- `__dict__` — перезаписываемый атрибут, содержащий словарь атрибутов функции. Функции в Python являются объектами первого класса, поэтому могут иметь любые произвольные аргументы, так же как и любой другой объект;
- `__closure__` — атрибут только для чтения, который содержит кортеж клеток со свободными переменными функции. Позволяет создавать параметризованные функции декораторов;
- `__annotations__` — перезаписываемый атрибут, который содержит аргумент функции и возвращает аннотации;
- `__kwdefaults__` — перезаписываемый атрибут, содержащий значение аргументов по умолчанию для именованных аргументов, если у функции они есть.

Далее рассмотрим, как сократить шаблонный код с помощью классов данных.

Сокращение шаблонного кода с помощью классов данных

Прежде чем углубиться в обсуждение классов Python, сделаем небольшое отступление. Мы обсудим относительно новые дополнения языка Python — а именно, классы данных. Модуль `dataclasses`, введенный в Python 3.7, включает в себя декоратор и функцию, которая позволяет легко добавлять сгенерированные специальные методы в пользовательские классы.

Рассмотрим следующий пример. Мы разрабатываем программу, выполняющую некие геометрические вычисления, и нам нужен класс, который позволяет хранить информацию о двумерных векторах. Мы будем выводить данные векторов на экран и выполнять простые математические операции, такие как сложение, вычитание и проверка равенства. Нам уже известно, что для этой цели можно использовать специальные методы. Мы можем реализовать наш класс `Vector` следующим образом:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Два вектора с оператором +"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )
```

```
    def __sub__(self, other):
        """Вычитание векторов 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):
        """Сравнение векторов на равенство"""
        return self.x == other.x and self.y == other.y
```

Ниже приведен пример интерактивной сессии, где показано поведение программы при использовании обычных операторов:

```
>>> 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
```

Данная реализация вектора довольно проста, но в ней много повторяющегося кода, от которого можно было бы избавиться. Если в вашей программе используется много подобных простых классов, которые не требуют сложной инициализации, то понадобится много кода только для методов `__init__()`, `__repr__()` и `__eq__()`.

С помощью модуля `dataclasses` мы можем сделать код класса `Vector` намного короче:

```
from dataclasses import dataclass

@dataclass
class Vector:
    x: int
    y: int

    def __add__(self, other):
        """Два вектора с оператором +"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )

    def __sub__(self, other):
        """Вычитание векторов оператором -"""
```

```
        return Vector(
            self.x - other.x,
            self.y - other.y,
        )
```

Декоратор класса `dataclass` считывает аннотации атрибута класса `Vector` и автоматически создает методы `__init__()`, `__repr__()` и `__eq__()`. Проверка на равенство по умолчанию предполагает равенство двух экземпляров, если все соответствующие атрибуты равны друг другу.

Но это не все. Классы данных предлагают множество полезных функций. Они легко совместимы с другими протоколами Python. Предположим, мы хотим, чтобы наши экземпляры класса `Vector` были неизменяемыми. В таком случае они могут быть использованы в качестве ключей словаря или входить во множество. Вы можете сделать это, просто добавив в декоратор аргумент `frozen=True`, как в примере ниже:

```
@dataclass(frozen=True)
class FrozenVector:
    x: int
    y: int
```

Такой замороженный класс `Vector` становится совершенно неизменяемым, и вы не сможете изменить ни один из его атрибутов. Но складывать и вычитать векторы все еще можно, как и показано в примере, поскольку эти операции просто создают новый объект `Vector`.

В завершение разговора о классах данных в этой главе отметим, что вы можете задать значения для определенных атрибутов по умолчанию с помощью конструктора `field()`. Можно использовать и статические значения, и конструкторы других объектов. Рассмотрим следующий пример:

```
>>> @dataclass
... class DataClassWithDefaults:
...     static_default: str = field(default="this is static default value")
...     factory_default: list = field(default_factory=list)
...
>>> DataClassWithDefaults()
DataClassWithDefaults(static_default='this is static default value',
factory_default=[])
```

В следующем разделе мы поговорим о подклассах встроенных типов.

Создание подклассов встроенных типов

Создать подклассы встроенных типов в Python довольно просто. Встроенный тип `object` — общий предок для всех встроенных типов, а также всех пользовательских классов, не имеющих явно указанного родительского класса. Благодаря этому каждый раз, когда вам нужно реализовать класс, который ведет себя почти как один из встроенных типов, лучше всего сделать его подтипом.

Теперь рассмотрим код класса под названием `distinctdict`, где используется именно такой метод. Это будет подкласс обычного типа `dict`. Этот новый класс будет вести себя в основном так же, как обычный тип Python `dict`. Но вместо того, чтобы допускать наличие нескольких ключей с одним значением, при добавлении значения он вызывает подкласс `ValueError` со справочным сообщением.

Как уже было сказано, встроенный тип `dict` является объектом подкласса:

```
>>> isinstance(dict(), object)
True
>>> issubclass(dict, object)
True
```

Это значит, что мы могли бы легко определить собственный словарь в виде подкласса:

```
class distinctdict(dict):
    ...
```

Описанный ранее подход будет работать как надо, поскольку подклассы из типов `dict`, `list` и `str` были разрешены, начиная с версии Python 2.2. Но, как правило, лучше всего создавать подкласс с помощью модулей `collections`:

- `collections.UserDict`;
- `collections.UserList`;
- `collections.UserString`.

С этими классами, как правило, легче работать, поскольку обычные объекты `dict`, `list` и `str` сохраняются в виде атрибутов данных этих классов.

Ниже приведен пример реализации типа `distinctdict`, который отменяет часть свойств словаря, чтобы он теперь мог содержать только уникальные значения:

```
from collections import UserDict

class DistinctError(ValueError):
    """Выдается, когда в distinctdict добавляется дубликат"""

class distinctdict(UserDict):
    """Словарь, в который нельзя добавлять дублирующиеся значения"""
    def __setitem__(self, key, value):
        if value in self.values():
            if (
                (key in self and self[key] != value) or
                key not in self
            ):
                raise DistinctError(
                    "This value already exists for different key"
                )
        super().__setitem__(key, value)
```

Ниже приведен пример использования класса `distinctdict` в интерактивной сессии:

```
>>> my = distinctdict()
>>> my['key'] = 'value'
>>> my['other_key'] = 'value'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 10, in __setitem__
DistinctError: This value already exists for different key
>>> my['other_key'] = 'value2'
>>> my
{'key': 'value', 'other_key': 'value2'}
>>> my.data
{'key': 'value', 'other_key': 'value2'}
```

Если вы посмотрите на имеющийся у вас код, то найдете много классов, которые частично реализуют протоколы или функции встроенных типов. Данные классы стали бы работать быстрее и чище, будь они реализованы как подтипы этих типов. Тип `list`, например, управляет последовательностями любого типа, и вы можете использовать его всякий раз, когда ваш класс обрабатывает последовательности или коллекции.

Ниже приведен простой пример класса `Folder`, который является подклассом `list` для вывода содержимого каталогов в виде древовидной структуры и управления им:

```
from collections import UserList

class Folder(UserList):
    def __init__(self, name):
        self.name = name

    def dir(self, nesting=0):
        offset = "  " * nesting
        print('%s%s/' % (offset, self.name))

        for element in self:
            if hasattr(element, 'dir'):
                element.dir(nesting + 1)
            else:
                print("%s  %s" % (offset, element))
```

Обратите внимание: на самом деле мы создали подкласс класса `UserList` из модуля `collections`, а не чистого `list`. Вы можете создавать подклассы чистых встроенных типов, например строк, словарей и множеств, но желательно использовать их двойники из модуля `collections`, поскольку подклассы в этом случае создавать немного легче.

Ниже приведен пример использования нашего класса `Folder` в интерактивном режиме:

```
>>> tree = Folder('project')
>>> tree.append('README.md')
>>> tree.dir()
project/
  README.md
>>> src = Folder('src')
>>> src.append('script.py')
>>> tree.append(src)
>>> tree.dir()
project/
  README.md
  src/
    script.py
>>> tree.remove(src)
>>> tree.dir()
project/
  README.md
```

Встроенных типов достаточно для большинства задач

Собираясь создать новый класс, который действует как последовательность или отображение, подумайте о его особенностях и просмотрите существующие встроенные типы. Модуль collections расширяет основные списки встроенных типов с помощью множества полезных контейнеров. Вы часто будете использовать один из них, что избавит вас от необходимости создавать собственные подклассы.

В следующем разделе поговорим о порядке разрешения методов (ПРМ).

ПРМ и доступ к методам из суперклассов

Класс `super` — это встроенный класс, который можно использовать для доступа к атрибуту, принадлежащему родительскому объекту.

В официальной документации super — встроенная функция, однако на самом деле это встроенный класс, даже если используется как функция:

```
>>> super
<class 'super'>
>>> isinstance(super, type)
```

Его сложновато использовать, если вы привыкли обращаться к атрибуту или методу класса через вызов родительского класса и передачу `self` в качестве первого

аргумента. Это старый паттерн, но его все еще можно найти в некоторых кодовых базах (особенно в устаревших проектах). Смотрите следующий код:

```
class Mama:  # Старый способ
    def says(self):
        print('do your homework')

class Sister(Mama):
    def says(self):
        Mama.says(self)
        print('and clean your bedroom')
```

Обратите внимание на строку `Mama.says(self)`. Здесь явно используется родительский класс. Это значит, что будет вызван метод `says()`, принадлежащий `Mama`. Но экземпляр, на котором он будет вызываться, указывается в аргументе `self`, который в данном случае является экземпляром `Sister`.

А если использовать `super`, то код будет выглядеть следующим образом:

```
class Sister(Mama):
    def says(self):
        super(Sister, self).says()
        print('and clean your bedroom')
```

Кроме того, можно также использовать более короткую форму вызова `super()`:

```
class Sister(Mama):
    def says(self):
        super().says()
        print('and clean your bedroom')
```

Внутри методов допускается сокращенная форма `super` (без каких-либо аргументов), но использование этого класса не ограничивается телом методов. Его можно задействовать в любой части кода, где требуется явный вызов метода суперкласса. Однако если `super` не используется внутри тела метода, то все его аргументы являются обязательными:

```
>>> anita = Sister()
>>> super(anita.__class__, anita).says()
do your homework
```

Последнее и самое важное, что следует отметить, — второй аргумент функции класса `super` является необязательным. Когда указан только первый аргумент, `super` возвращает неограниченный тип. Это особенно полезно при работе с `classmethod`:

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    def __repr__(self):
        return "Pizza with " + " and ".join(self.toppings)
```

```
    @classmethod
    def recommend(cls):
        """Здесь не помешала бы пицца, причем с начинкой"""
        return cls(['spam', 'ham', 'eggs'])

class VikingPizza(Pizza):
    @classmethod
    def recommend(cls):
        """Используем ту же рекомендацию, что и для super,
           но добавляем немного «воды»"""
        recommended = super(VikingPizza).recommend()
        recommended.toppings += ['spam'] * 5
        return recommended
```

Обратите внимание: вариация `super()` без аргументов также допускается для методов, обработанных декоратором `classmethod`. Если `super()` вызывается без аргументов в таких методах, то считается, что определен только первый аргумент.

Показанные случаи использования просты и понятны, но, когда вы сталкиваетесь со сложными схемами наследования, применять `super` становится трудно. Прежде чем объяснять эту проблему, нужно понять, когда следует избегать `super` и как в Python работает *порядок разрешения методов* (ПРМ).

Обсудим классы старого стиля и суперклассы в Python 2.

Классы старого стиля и суперклассы в Python 2

`super()` в Python 2 работает почти точно так же, как и в Python 3. Единственное отличие заключается в том, что название короче, форма без аргументов недоступна, поэтому всегда должен использоваться по меньшей мере один из ожидаемых аргументов.

Еще один важный нюанс для программистов, которым приходится писать совместимый между версиями код, заключается в том, что `super` в Python 2 работает только для новых классов. Более ранние версии Python не имеют общего предка `object` для всех классов. Старое поведение сохранено во всех версиях Python 2.x для обратной совместимости, поэтому в данных версиях, если определение класса не имеет явно указанного предка, он интерпретируется как класс старого стиля и `super` в нем использоваться не может:

```
class OldStyle1:
    pass

class OldStyle2(OldStyle1):
    pass
```

Классы нового стиля в Python 2 должны явно наследовать от типа `object` или другого класса нового стиля:

```
class NewStyleClass(object):
    pass

class NewStyleClassToo(NewStyleClass):
    pass
```

Python 3 больше не поддерживает концепцию классов старого стиля, поэтому любой класс, который явно не наследует от любого другого класса, неявно наследует от `object`. Это значит, что явное указание наследования от `object` может показаться излишним. Обычно принято не включать в программу избыточный код, но удаление такой избыточности на самом деле подходит только для проектов, которые не планируется запускать на Python 2. Код, в котором должна быть совместимость, должен включать в себя `object` как предка базовых классов, даже если это является излишним в Python 3. В противном случае такие классы будут интерпретироваться по старому стилю, и это в конечном итоге приведет к проблемам, которые очень трудно диагностировать.

Разберемся с ПРМ в Python в следующем подразделе.

Понимание ПРМ в Python

ПРМ в Python основан на C3, созданном для языка программирования Dylan (opendylan.org). Справочный документ, написанный Микеле Симионато, можно найти по адресу www.python.org/download/releases/2.3/mro. В документе показано, как C3 строит линеаризацию класса или приоритет, который представляет собой упорядоченный список предков. Этот список используется для поиска атрибута. Алгоритм C3 более подробно описан ниже.

Изменение ПРМ позволяет устранить проблему, появившуюся после создания общего базового типа (то есть типа `object`). До перехода на C3 если класс имел два предка (рис. 4.1), то порядок разрешения методов было довольно легко вычислить и отследить только в простых случаях, когда не использовалось несколько моделей наследования.

Ниже приведен пример кода, который в соответствии с Python 2 не будет использовать C3:

```
class Base1:
    pass

class Base2:
    def method(self):
        print('Base2')

class MyClass(Base1, Base2):
    pass
```

```
>>> MyClass().method()
Base2
```

Когда вызывается `MyClass().method()`, интерпретатор ищет метод в `MyClass`, затем в `Base1` и в конечном итоге находит его в `Base2`.

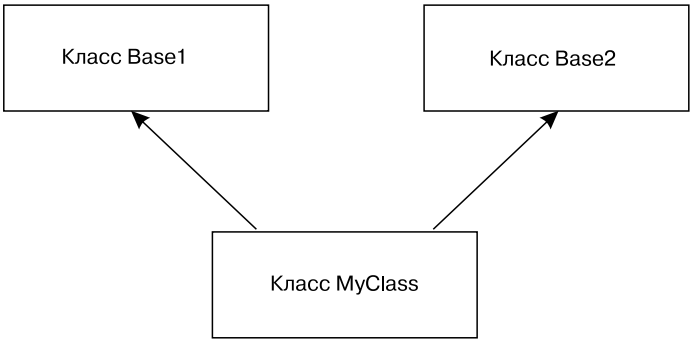


Рис. 4.1. Классическая иерархия

Когда мы вводим класс `CommonBase` на вершине нашей иерархии классов (`Base1` и `Base2` унаследуют от него; рис. 4.2), все усложнится. В результате вместо того, чтобы следовать простому порядку разрешения, работающему по правилу *«слева направо от нижних к верхним»*, мы вернемся к вершине через класс `Base1`, прежде чем заглянуть в `Base2`. Этот алгоритм дает результат, противоречащий интуитивному. Выполняемый метод — не обязательно ближайший метод в дереве наследования.

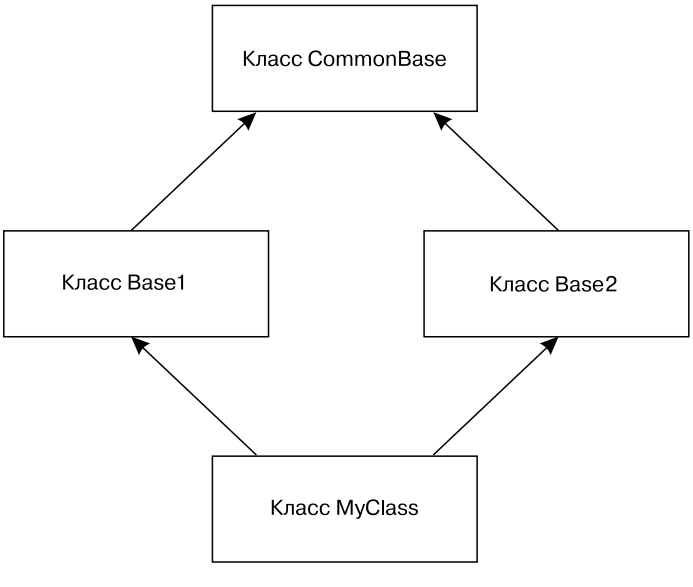


Рис. 4.2. Иерархия классов типа Diamond

Этот алгоритм по-прежнему доступен в Python 2 для классов старого стиля. Ниже представлен пример старого разрешения метода в Python 2 с помощью таких классов:

```
class CommonBase:
    def method(self):
        print('CommonBase')

class Base1(CommonBase):
    pass

class Base2(CommonBase):
    def method(self):
        print('Base2')

class MyClass(Base1, Base2):
    pass
```

Приведенный ниже текст интерактивной сессии показывает, что `Base2.method()` не будет вызываться, несмотря на то что класс `Base2` находится ближе в иерархии классов к `MyClass`, чем `CommonBase`:

```
>>> MyClass().method()
CommonBase
```

Такой сценарий наследования используется крайне редко, поэтому данная проблема скорее теоретическая, чем практическая. Стандартная библиотека не структурирует иерархию наследования подобным образом, и многие разработчики считают, что так делать не стоит. Но с введением `object` как родителя всех типов проблема множественного наследования всплывает на стороне языка C, что приводит к конфликтам при выполнении подтипов. Следует также отметить: каждый класс в Python 3 теперь имеет общего предка. Поскольку заставить его работать должным образом с существующим ПРМ слишком сложно, проще было ввести новый ПРМ.

Итак, тот же пример в Python 3 дает другой результат:

```
class CommonBase:
    def method(self):
        print('CommonBase')

class Base1(CommonBase):
    pass

class Base2(CommonBase):
    def method(self):
        print('Base2')
```

```
class MyClass(Base1, Base2):
    pass
```

А этот пример показывает, что при использовании сериализации C3 будет применяться метод ближайшего предка:

```
>>> MyClass().method()
Base2
```

Обратите внимание: такое поведение не может быть воспроизведено в Python 2 без явного наследования CommonBase от object. Как следствие, бывает полезно напрямую указывать такое наследование в Python 3, даже если это является избыточным, — о чем мы и говорили в предыдущем подразделе.

ПРМ в Python основан на рекурсивном вызове через базовые классы. Мы говорили о работе Мишеля Симионато в начале этого подраздела — так вот, символическое обозначение C3 для нашего примера будет выглядеть следующим образом:

```
L[MyClass(Base1, Base2)] = MyClass + merge(L[Base1], L[Base2], Base1, Base2)
```

Здесь `L[MyClass]` — это линеаризация `MyClass`, а `merge` — специфический алгоритм, который объединяет несколько результатов линеаризации.

Таким образом, как говорит в своей статье Симионато, *«линеаризация C является суммой C и слияния линеаризации родителей и списка родителей»*.

Алгоритм `merge` отвечает за удаление дубликатов и сохранение правильного порядка. Вот как он описан в документе (с адаптацией под наш пример): *«Берем первый элемент первого списка, то есть L[Base1][0]; если он не находится в хвосте каких-либо других списков, то добавляем его к линеаризации MyClass и удаляем из списков в merge, в противном случае смотрим на начало следующего списка и берем его, если подходит. Затем повторяем эту операцию, пока все классы не будут удалены или окажется невозможно найти хорошие первые элементы. В таком случае нельзя построить слияние, и Python 2.3 откажется от создания класса MyClass и выбросит исключение»*.

Элемент `head` — первый в списке, а `tail` содержит остальные элементы. Так, например, в `(BASE1, Base2, ..., BaseN)` элемент `Base1` — это `head`, а `(Base2, ..., BaseN)` — хвост `(tail)`.

Иными словами, C3 выполняет рекурсивный поиск на глубину каждого из родителей, чтобы получить последовательность списков. Затем он вычисляет правило «слева направо», чтобы объединить все списки с неоднозначностью иерархии, если класс входит в несколько списков.

Итак, результат выглядит следующим образом:

```
def L(klass):
    return [k.__name__ for k in klass.__mro__]

>>> L(MyClass)
['MyClass', 'Base1', 'Base2', 'CommonBase', 'object']
```

Атрибут класса __mro__ (который доступен только для чтения) хранит результат вычисления линеаризации. Расчет делается в тот момент, когда загружается определение класса.

Кроме того, можно вызвать MyClass.mro(), чтобы вычислить и получить результат. Это еще одна причина, почему классы в Python 2 нужно рассматривать как отдельный кейс. У классов старого стиля в Python 2 есть определенный порядок, в котором разрешаются методы, но в них нет атрибута __mro__. Таким образом, несмотря на порядок разрешения, неправильно говорить, что у них есть ПРМ. В большинстве случаев, когда кто-то говорит о ПРМ в Python, имеется в виду алгоритм C3, описанный в данном разделе.

Теперь обсудим ряд проблем, с которыми сталкиваются программисты.

Ловушки суперкласса

Теперь вернемся к вызову `super()`. Если вы работаете с иерархией множественного наследования, то здесь могут быть проблемы. В основном они связаны с инициализацией классов. В Python методы инициализации базовых классов (то есть методы `__init__`) не вызываются неявно в классах предка, если классы предка переопределяют `__init__`. В таких случаях необходимо вызывать методы суперкласса явным образом, и иногда это может привести к проблемам инициализации.

В этом подразделе мы рассмотрим несколько примеров таких проблемных ситуаций.

Смешивание вызова суперкласса и явного вызова

В следующем примере, взятом с сайта Джеймса Найта (fuhm.net/super-harmful), есть класс `C`, который вызывает методы инициализации своих родительских классов с помощью `super()`.

```
class A:
    def __init__(self):
        print("A", end=" ")
        super().__init__()
```

```
class B:
    def __init__(self):
        print("B", end=" ")
        super().__init__()

class C(A, B):
    def __init__(self):
        print("C", end=" ")
        A.__init__(self)
        B.__init__(self)
```

Результат:

```
>>> print("MRO:", [x.__name__ for x in C.__mro__])
MRO: ['C', 'A', 'B', 'object']
>>> C()
C A B B <__main__.C object at 0x0000000001217C50>
```

Как видно, инициализация класса C вызывает `B.__init()__` дважды. Чтобы избежать подобных проблем, `super` надо использовать в иерархии целого класса. Проблема заключается в том, что иногда часть такой сложной иерархии может быть расположена в стороннем коде, в результате чего она будет недоступна для понимания. О других ловушках, связанных с неясностью иерархии и правил наследования, можно почитать на странице Джеймса.

К сожалению, вы не можете быть уверены в том, что внешние пакеты используют в коде `super()`. Всякий раз, когда нужно создать подкласс какого-то стороннего класса, стоит заглянуть в его код и код других классов в ПРМ. Это может быть утомительно, но в качестве бонуса вы получите некую информацию о качестве кода в пакете и будете его лучше понимать, да и вообще узнаете что-то новое.

Гетерогенные аргументы

Еще одна проблема с `super` возникает в случае, если методы классов в пределах иерархии классов используют несовместимые наборы аргументов. Как класс может вызывать базовый класс `__init__()`, если у него не такая же сигнатура? Это приводит к следующей задаче:

```
class CommonBase:
    def __init__(self):
        print('CommonBase')
        super().__init__()

class Base1(CommonBase):
    def __init__(self):
        print('Base1')
        super().__init__()
```

```
class Base2(CommonBase):
    def __init__(self, arg):
        print('base2')
        super().__init__()

class MyClass(Base1 , Base2):
    def __init__(self, arg):
        print('my base')
        super().__init__(arg)
```

Попытка создать экземпляр `MyClass` вызовет `TypeError` из-за несоответствия сигнатур родительских классов `__init__`:

```
>>> MyClass(10)
my base
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __init__
TypeError: __init__() takes 1 positional argument but 2 were given
```

Одним из решений было бы применить упаковку аргументов и именованных аргументов с помощью `*args` и `**kwargs`, так что все конструкторы будут передавать все параметры, даже если не используют их:

```
class CommonBase:
    def __init__(self, *args, **kwargs):
        print('CommonBase')
        super().__init__()

class Base1(CommonBase):
    def __init__(self, *args, **kwargs):
        print('Base1')
        super().__init__(*args, **kwargs)

class Base2(CommonBase):
    def __init__(self, *args, **kwargs):
        print('base2')
        super().__init__(*args, **kwargs)

class MyClass(Base1 , Base2):
    def __init__(self, arg):
        print('my base')
        super().__init__(arg)
```

При таком подходе сигнатуры родительского класса всегда будут совпадать:

```
>>> _ = MyClass(10)
my base
Base1
base2
CommonBase
```

Но это ужасный костыль, поскольку в таком случае все конструкторы будут принимать вообще любые параметры. То есть любые баги тоже пройдут. Другое решение — использовать явный вызов `__init__()` в `MyClass`, но это привело бы к первой ловушке, о которой мы говорили.

В следующем подразделе мы обсудим практические рекомендации.

Практические рекомендации

Чтобы избежать всех вышеуказанных проблем и до тех пор, пока Python не эволюционировал в этой области, придется принять во внимание следующие моменты.

- *Следует избегать множественного наследования* — вместо него можно задействовать паттерны проектирования, представленные в главе 17.
- *Использовать super нужно последовательно* — в иерархии классов `super` следует применять либо везде, либо нигде. Смешивание `super` и классических классов ведет к путанице. Программисты, как правило, избегают `super`, чтобы сделать код более ясным.
- *Применять явное наследование от объекта в Python 3, если нужна совместимость с Python 2* — классы без какого-либо предка в Python 2 считаются классами старого стиля. Смешения классов старого стиля с новыми классами в Python 2 следует избегать.
- *Нужно просмотреть иерархию классов перед вызовом метода родительского класса* — чтобы избежать каких-либо проблем, каждый раз при вызове метода класса родителя следует обязательно изучить ПРМ (с помощью `__mro__`).

Посмотрим на паттерны доступа для расширенных атрибутов.

Паттерны доступа к расширенным атрибутам

Изучая Python, многие программисты C++ и Java удивляются отсутствию ключевого слова `private`. Наиболее близкая к нему концепция — это *искажение (декорирование) имени* (name mangling). Каждый раз, когда атрибут получает префикс `__`, он динамически переименовывается интерпретатором:

```
class MyClass:
    __secret_value = 1
```

Доступ к атрибуту `__secret_value` по его изначальному имени приведет к выбрасыванию исключения `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'
>>> dir(MyClass)
['_MyClass__secret_value', '__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
>>> instance_of._MyClass__secret_value
1
```

Это сделано специально для того, чтобы избежать конфликта имен по наследованию, так как атрибут переименовывается именем класса в качестве префикса. Это не точный аналог `private`, поскольку атрибут может быть доступен через составленное имя. Данное свойство можно применить для защиты доступа некоторых атрибутов, однако на практике `__` не используется никогда. Если атрибут не является публичным, то принято использовать префикс `_`. Он не вызывает алгоритм декорирования имени, но документирует атрибут как приватный элемент класса и является преобладающим стилем.

В Python есть и другие механизмы, позволяющие отделить публичную часть класса от приватной. Дескрипторы и свойства дают возможность аккуратно оформить такое разделение.

Дескрипторы

Дескриптор позволяет настроить действие, которое происходит, когда вы ссылаетесь на атрибут объекта.

Дескрипторы лежат в основе организации сложного доступа к атрибутам в Python. Они используются для реализации свойств, методов, методов класса, статических методов и надтипов. Это классы, которые определяют, каким образом будет получен доступ к атрибутам другого класса. Иными словами, класс может делегировать управление атрибута другому классу.

Классы дескрипторов основаны на трех специальных методах, которые формируют *протокол дескриптора*:

- `__set__(self, obj, value)` — вызывается всякий раз, когда задается атрибут. В следующих примерах мы будем называть его *«сеттер»*;
- `__get__(self, obj, owner=None)` — вызывается всякий раз, когда считывается атрибут (далее *геттер*);
- `__delete__(self, object)` — вызывается, когда `del` вызывается атрибутом.

Дескриптор, который реализует `__get__` и `__set__`, называется *дескриптором данных*. Если он просто реализует `__get__`, то называется *дескриптором без данных*.

Методы этого протокола фактически вызываются методом `__getattribute__()` (не путать с `__getattr__()`, который имеет другое назначение) при каждом поиске атрибута. Всякий раз, когда такой поиск выполняется с помощью точки или прямого вызова функции, неявно вызывается метод `__getattribute__()`, который ищет атрибут в следующем порядке.

1. Проверяет, является ли атрибут дескриптором данных на объекте класса экземпляра.
2. Если нет, то смотрит, найдется ли атрибут в `__dict__` объекта экземпляра.
3. Наконец, проверяет, является ли атрибут дескриптором без данных на объекте класса экземпляра.

Иными словами, дескрипторы данных имеют приоритет над `__dict__`, который, в свою очередь, имеет приоритет над дескрипторами без данных.

Для ясности приведем пример из официальной документации Python, в котором показано, как дескрипторы работают в реальном коде:

```
class RevealAccess(object):
    """Дескриптор данных, который задает и возвращает значения
       и выводит сообщения о попытках доступа
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

class MyClass(object):
    x = RevealAccess(10, 'var "x"')
    y = 5
```

Вот пример его использования в интерактивном режиме:

```
>>> m = MyClass()
>>> m.x
Retrieving var "x"
```

```
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

Пример ясно показывает, что если класс имеет дескриптор данных для этого атрибута, то вызывается метод `__get__()`, чтобы вернуть значение каждый раз, когда извлекается атрибут экземпляра, а `__set__()` вызывается всякий раз, когда такому атрибуту присваивается значение. Использование метода `__del__` в предыдущем примере не показано, но должно быть очевидно: он вызывается всякий раз, когда атрибут экземпляра удаляется с помощью оператора `del instance.attribute` или `delattr(instance, 'attribute')`.

Разница между дескрипторами с данными и без имеет большое значение по причинам, которые мы упомянули в начале подраздела. В Python используется протокол дескриптора для связывания функций класса с экземплярами через методы. Они также применяются в декораторах `classmethod` и `staticmethod`. Это происходит потому, что функциональные объекты по сути также являются дескрипторами без данных:

```
>>> def function(): pass
>>> hasattr(function, '__get__')
True
>>> hasattr(function, '__set__')
False
```

Это верно и для функций, созданных с помощью лямбда-выражений:

```
>>> hasattr(lambda: None, '__get__')
True
>>> hasattr(lambda: None, '__set__')
False
```

Таким образом, если `__dict__` не будет иметь приоритет над дескрипторами без данных, мы не сможем динамически переопределить конкретные методы уже созданных экземпляров во время выполнения. К счастью, благодаря тому, как дескрипторы работают в Python, это возможно; поэтому разработчики могут выбирать, в каких экземплярах что работает, не используя подклассы.

Пример из реальной жизни: ленивое вычисление атрибутов. Один из примеров использования дескрипторов — задержка инициализации атрибута класса в момент доступа к нему из экземпляра. Это может быть полезно, если инициализация таких атрибутов зависит от глобального контекста приложения. Другой случай — когда такая инициализация слишком затратна, и неизвестно, будет ли атрибут вообще использоваться после импорта класса. Такой дескриптор можно реализовать следующим образом:

```
class InitOnAccess:
    def __init__(self, klass, *args, **kwargs):
        self.klass = klass
        self.args = args
        self.kwargs = kwargs
        self._initialized = None

    def __get__(self, instance, owner):
        if self._initialized is None:
            print('initialized!')
            self._initialized = self.klass(*self.args, **self.kwargs)
        else:
            print('cached!')
        return self._initialized
```

Ниже представлен пример использования:

```
>>> class MyClass:
...     lazily_initialized = InitOnAccess(list, "argument")
...
>>> m = MyClass()
>>> m.lazily_initialized
initialized!
['a', 'r', 'g', 'u', 'm', 'e', 'n', 't']
>>> m.lazily_initialized
cached!
['a', 'r', 'g', 'u', 'm', 'e', 'n', 't']
```

Официальная библиотека OpenGL Python на PyPI под названием PyOpenGL использует такую технику, чтобы реализовать объект `lazy_property`, который является одновременно 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
```

Такая реализация аналогична использованию декоратора `property` (о нем поговорим позже), но функция, которая оборачивается декоратором, выполняется только один раз, а затем атрибут класса заменяется значением, возвращенным этим свойством функции. Данный метод часто бывает полезен, когда необходимо одновременно выполнить два требования:

- экземпляр объекта должен быть сохранен как атрибут класса, который распределяется между его экземплярами (для экономии ресурсов);
- этот объект не может быть инициализирован в момент импорта, поскольку процесс его создания зависит от некоего глобального состояния приложения/контекста.

В случае приложений, написанных с использованием OpenGL, вы будете часто сталкиваться с такой ситуацией. Например, создание шейдеров в OpenGL обходится дорого, поскольку требует компиляции кода, написанного на *OpenGL Shading Language (GLSL)*. Разумно создавать их только один раз и в то же время держать их описание в непосредственной близости от классов, которым они нужны. С другой стороны, шейдерные компиляции не могут быть выполнены без инициализации контекста OpenGL, так что их трудно определить и собрать в глобальном пространстве имен модуля на момент импорта.

В следующем примере показано возможное использование модифицированной версии декоратора `lazy_property` PyOpenGL (здесь `lazy_class_attribute`) в некоем абстрактном приложении OpenGL. Изменения оригинального декоратора `lazy_property` требуются для того, чтобы разрешить совместное использование атрибута различными экземплярами класса:

```
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(obj or cls)
        # Примечание: хранение объекта не-экземпляра класса
        #       независимо от уровня доступа
        setattr(cls, self.fget.__name__, value)
        return value

class ObjectUsingShaderProgram(object):
    # Банальная реализация шейдера-вершины
    VERTEX_CODE = """
        #version 330 core
        layout(location = 0) in vec4 vertexPosition;
        void main(){
            gl_Position =  vertexPosition;
        }
    """
    # Шейдер грани, который закрашивает все белым
    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("compiling!")
        return shaders.compileProgram(
            shaders.compileShader(
                self.VERTEX_CODE, gl.GL_VERTEX_SHADER
            ),
            shaders.compileShader(
                self.FRAGMENT_CODE, gl.GL_FRAGMENT_SHADER
            )
        )
```

Как и все расширенные функции синтаксиса Python, эту также следует использовать с осторожностью и хорошо документировать в коде. Неопытным разработчикам измененное поведение класса может преподнести сюрпризы, поскольку дескрипторы влияют на поведение класса. Поэтому очень важно убедиться, что все члены вашей команды знакомы с дескрипторами и понимают эту концепцию, если она играет важную роль в кодовой базе проекта.

Свойства

Свойства предоставляют встроенный тип дескриптора, который знает, как связать атрибут с набором методов. Свойство принимает четыре необязательных аргумента: `fget`, `fset`, `fdel` и `doc`. Последний из них может быть предусмотрен для определения строки документации, связанной с атрибутом, как если бы это был метод. Ниже приведен пример класса `Rectangle`, которым можно управлять либо путем прямого доступа к атрибутам, хранящим две угловые точки, либо с помощью свойств `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="rectangle width measured from left"
    )
    height = property(
        _height_get, _height_set,
        doc="rectangle height measured from top"
    )

    def __repr__(self):
        return "{}({}, {}, {}, {})".format(
            self.__class__.__name__,
            self.x1, self.y1, self.x2, self.y2
        )
```

В следующем фрагменте кода приведен пример таких свойств, определенных в интерактивной сессии:

```
>>> 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 in module chapter3:
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
 |      rectangle height measured from top
 |
 |  width
 |      rectangle width measured from left
```

Эти свойства облегчают написание дескрипторов, но с ними следует аккуратно обращаться при использовании наследования по классам. Атрибут создается динамически с помощью методов текущего класса и не будет применять методы, которые переопределены в производных классах.

Приведенный в следующем примере код не сможет переопределить реализацию метода `fget` из свойства `width` родительского класса `(Rectangle)`:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} meters".format(self.x2 - self.x1)
...
>>> Rectangle(0, 0, 100, 100).width
100
```

Чтобы решить эту проблему, все свойство следует перезаписать в производном классе:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} meters".format(self.x2 - self.x1)
...     width = property(_width_get, Rectangle.width.fset)
...
>>> MetricRectangle(0, 0, 100, 100).width
'100 meters'
```

К сожалению, код имеет кое-какие проблемы с сопровождением. Может возникнуть путаница, если разработчик решит изменить родительский класс, но забудет обновить вызов свойства. Именно поэтому не рекомендуется переопределять только части поведения свойств. Вместо того чтобы полагаться на реализацию родительского класса, рекомендуется переписать все методы свойств в производных классах, если нужно изменить способ их работы. Обычно других вариантов нет, поскольку изменение свойств поведения `setter` влечет за собой изменение в поведении `getter`.

Лучшим вариантом создания свойств будет использование `property` в качестве декоратора. Это позволит сократить количество сигнатур методов внутри класса и сделать код более читабельным и удобным в сопровождении:

```
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 измеряется слева направо"""
        return self.x2 - self.x1

    @width.setter
    def width(self, value):
        self.x2 = self.x1 + value

    @property
    def height(self):
```

```
        """Высота измеряется сверху вниз"""
        return self.y2 - self.y1

    @height.setter
    def height(self, value):
        self.y2 = self.y1 + value
```

Слоты

Интересная функция, которая очень редко используются разработчиками, — это слоты. Они позволяют установить статический список атрибутов для класса с помощью атрибута `__slots__` и пропустить создание словаря `__dict__` в каждом экземпляре класса. Они были созданы для экономии места в памяти для классов с малочисленными атрибутами, так как `__dict__` создается не в каждом экземпляре.

Кроме того, они могут помочь в создании классов, чьи сигнатуры нужно заморозить. Например, если вам необходимо ограничить динамические свойства языка для конкретного класса, то слоты могут помочь:

```
>>> class Frozen:
...     __slots__ = ['ice', 'cream']
...
>>> '__dict__' in dir(Frozen)
False
>>> 'ice' in dir(Frozen)
True
>>> frozen = Frozen()
>>> frozen.ice = True
>>> frozen.cream = None
>>> frozen.icy = True
Traceback (most recent call last): File "<input>", line 1, in <module>
AttributeError: 'Frozen' object has no attribute 'icy'
```

Эту возможность нужно использовать с осторожностью. Когда набор доступных атрибутов ограничен слотами, намного сложнее добавить что-то к объекту динамически. Некоторые известные приемы, например обезьяний патч (monkey patching), не будут работать с экземплярами классов, которые имеют определенные слоты. К счастью, новые атрибуты можно добавить к производным классам, если они не имеют собственных определенных слотов:

```
>>> class Unfrozen(Frozen):
...     pass
...
>>> unfrozen = Unfrozen()
>>> unfrozen.icy = False
>>> unfrozen.icy
False
```

Резюме

В этой главе мы рассмотрели современные элементы синтаксиса Python, относящиеся к моделям класса и объектно-ориентированному программированию.

Мы начали с объяснения концепции протокола языка, обсуждали подклассы встроенных типов и способы вызова методов из суперклассов. Далее мы перешли к более сложным понятиям объектно-ориентированного программирования на Python. Речь шла о полезных функциях синтаксиса, работающих с доступом к атрибутам экземпляра: дескрипторах и свойствах. Мы показали, как их можно использовать для создания более чистого и удобного в сопровождении кода.

В следующей главе мы рассмотрим обширную тему метапрограммирования в Python. Мы будем повторно использовать некоторые уже известные особенности синтаксиса, чтобы показать различные методы метапрограммирования.

5 Элементы метапрограммирования

Метапрограммирование — один из самых сложных и мощных подходов к программированию в Python. Его инструменты и методы эволюционировали вместе с Python, и прежде, чем углубляться в данную тему, важно хорошо изучить все элементы современного синтаксиса Python. Мы обсуждали их в двух предыдущих главах. Если вы изучили их внимательно, то должны знать достаточно, чтобы полностью понять содержание этой главы.

В данной главе мы объясним, что такое метапрограммирование в Python, и представим несколько практических подходов к нему.

В этой главе:

- что такое метапрограммирование;
- декораторы;
- метаклассы;
- генерация кода.

Технические требования

Ниже приведены пакеты Python, упомянутые в этой главе, которые можно скачать с PyPI:

- `macropy`;
- `falcon`;
- `hy`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по адресу github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter5.

Что такое метапрограммирование

Вероятно, есть какое-то хорошее академическое определение метапрограммирования, которое можно было бы процитировать здесь, но данная книга все же скорее о мастерстве программирования, чем о теории компьютерных наук. Именно поэтому мы будем использовать следующее простое определение: *«Метапрограммирование — это методика написания компьютерных программ, которые могут обрабатываться как данные, что позволяет им просматривать, создавать и/или изменять себя во время работы»*.

Используя это определение, выделим два основных подхода к метапрограммированию в Python.

Первый подход концентрируется на способности языка к самоанализу собственных элементов — функций, классов или типов, а также способности создавать или изменять их динамически. В Python и впрямь немало инструментов для работы в этой области. Данная особенность языка используется в IDE (например, PyCharm), предоставляя возможности анализа кода и предложения имен в режиме реального времени. Самые простые инструменты метапрограммирования в Python, в которых применяется самоанализ языка, — это декораторы, позволяющие добавлять дополнительные функциональные возможности к существующим функциям, методам и классам. Кроме того, существуют специальные методы классов, которые позволяют вмешиваться в процедуру создания экземпляра класса. Самые мощные в этом смысле — метаклассы, благодаря которым программисты могут полностью переделать реализацию объектно-ориентированного программирования в Python.

Второй подход позволяет программистам работать непосредственно с кодом либо в сыром виде (обычный текст), либо в более доступной форме абстрактного синтаксического дерева (abstract syntax tree, AST). Данный подход, разумеется, более сложный и трудный в реализации, зато дает возможность делать действительно экстраординарные вещи, такие как расширение синтаксиса языка Python или даже создание собственного *предметно-ориентированного языка* (domain-specific language, DSL).

В следующем подразделе мы подробнее поговорим о декораторах.

Декораторы как средство метапрограммирования

Мы говорили о синтаксисе декораторов в главе 3. Это синтаксический сахар, который работает по следующей простой схеме:

```
def decorated_function():
    pass
decorated_function = some_decorator(decorated_function)
```

Эта многословная форма декорирования функции ясно показывает, что делает декоратор. Он принимает объект функции и изменяет его во время выполнения.

В результате новая функция (или что-нибудь еще) создается на основе предыдущей функции объекта с тем же именем. Это декорирование может быть сложной операцией, выполняющей некий самоанализ кода или декорированную функцию, чтобы дать разные результаты в зависимости от того, как была реализована оригинальная функция. Все это значит, что декоратор можно рассматривать в качестве инструмента метапрограммирования.

И это хорошая новость. Основные принципы декораторов относительно просты для понимания и в большинстве случаев позволяют сделать код более коротким, читабельным и удобным в сопровождении. Остальные инструменты метапрограммирования в Python труднее и понять, и использовать. Кроме того, код от них только усложняется.

Далее рассмотрим декораторы класса.

Декораторы класса

Одной из менее известных особенностей синтаксиса Python являются декораторы класса. По синтаксису и реализации они аналогичны декораторам функций, как мы уже упоминали в главе 3. Единственное отличие состоит в том, что они возвращают класс, а не функцию. Ниже приведен пример декоратора класса, который изменяет метод `__repr__()`, чтобы тот возвращал печатаемое представление объекта, сокращенное до произвольного количества символов:

```
def short_repr(cls):
    cls.__repr__ = lambda self: super(cls, self).__repr__()[:8]
    return cls

@short_repr
class ClassWithRelativelyLongName:
    pass
```

Вывод будет выглядеть так:

```
>>> ClassWithRelativelyLongName()
<ClassWi
```

Конечно, этот фрагмент никоим образом не является примером хорошего кода. Тем не менее он показывает, как возможности языка, описанные в предыдущей главе, можно использовать совместно, например:

- во время выполнения можно изменять не только экземпляры, но и объекты класса;
- функции тоже являются дескрипторами, поэтому они могут быть добавлены к классу во время выполнения, поскольку фактическое связывание метода выполняется в процессе поиска атрибута как часть протокола дескриптора;

- вызов `super()` можно использовать вне области видимости определения класса, если предоставлены соответствующие аргументы;
- наконец, декораторы классов можно применять при определении классов.

Все остальное, относящееся к декораторам функций, также относится к декораторам классов. Самое главное — это то, что в них доступны замыкания и параметризация. Пользуясь данными фактами, предыдущий пример можно переписать, сделав более читабельным и удобным в сопровождении:

```
def parametrized_short_repr(max_width=8):
    """Параметризованный декоратор, сокращающий представление"""
    def parametrized(cls):
        """Внутренняя функция-оболочка, которая является фактическим декоратором"""
        class ShortlyRepresented(cls):
            """Подкласс, представляющий поведение декоратора"""
            def __repr__(self):
                return super().__repr__()[:max_width]

        return ShortlyRepresented

    return parametrized
```

Основной недостаток использования замыканий в декораторах классов заключается в том, что полученные объекты являются не экземплярами класса, который был задекорирован, а экземплярами подкласса, созданного динамически в функции декоратора. Среди прочего, это повлияет на атрибуты `__name__` и `__doc__`:

```
@parametrized_short_repr(10)
class ClassWithLittleBitLongerLongName:
    pass
```

Такое использование декораторов класса приведет к следующим изменениям в метаданных класса:

```
>>> ClassWithLittleBitLongerLongName().__class__
<class 'ShortlyRepresented'>
>>> ClassWithLittleBitLongerLongName().__doc__
'Subclass that provides decorated behavior'
```

К сожалению, исправить это не так просто, как мы объяснили в главе 3. В декораторах класса нельзя просто брать и использовать дополнительный декоратор `wraps`, чтобы сохранить первоначальный тип класса и метаданные. Поэтому использование декораторов класса в некоторых случаях будет ограничено. Они могут, например, испортить результаты работы средств автоматизированной генерации документации.

Тем не менее, несмотря на этот недостаток, декораторы класса — простая и облегченная альтернатива популярному паттерну «Примесь». Таковым в Python называют класс, который не создает экземпляров, но вместо этого используется

для создания многоразового API или функциональности других существующих классов. Такие классы почти всегда добавляются с помощью множественного наследования. Как правило, это принимает следующий вид:

```
class SomeConcreteClass(MixinClass, SomeBaseClass):
    pass
```

Примеси (миксины, mixin) являются полезным паттерном проектирования, который применяется во многих библиотеках и фреймворках. Например, они широко используются в Django. Несмотря на пользу и популярность, примеси могут стать проблемой, будучи плохо разработанными, поскольку в большинстве случаев требуют от программиста использования множественного наследования. Как мы уже отмечали, множественное наследование в Python реализуется относительно хорошо благодаря ПРМ. Но все равно по возможности старайтесь избегать множественного наследования, так как оно усложняет понимание кода. Декораторы класса могут быть хорошей заменой примесей.

Посмотрим на использование `__new__()` для переопределения процесса создания экземпляра.

Использование __new__() для переопределения процесса создания экземпляра

Специальный метод `__new__()` — это статический метод, который отвечает за создание экземпляров класса. Его нет необходимости объявлять статическим с помощью декоратора `staticmethod`. Этот метод `__new__(cls, [, ...])` вызывается до `__init__()`. Как правило, реализация переопределенного метода `__new__()` вызывает его версию суперкласса, используя `super()`. Метод `__new__()` с соответствующими аргументами модифицирует экземпляр перед его возвращением.

Ниже приведен пример класса с переопределенной реализацией метода `__new__()` для подсчета количества экземпляров класса:

```
class InstanceCountingClass:
    instances_created = 0
    def __new__(cls, *args, **kwargs):
        print('__new__() called with:', cls, args, kwargs)
        instance = super().__new__(cls)
        instance.number = cls.instances_created
        cls.instances_created += 1

        return instance

    def __init__(self, attribute):
        print('__init__() called with:', self, attribute)
        self.attribute = attribute
```

Ниже представлен журнал интерактивной сессии, которая показывает, как работает наша реализация `InstanceCountingClass`:

```
>>> from instance_counting import InstanceCountingClass
>>> instance1 = InstanceCountingClass('abc')
__new__() called with: <class '__main__.InstanceCountingClass'> ('abc',) {}
__init__() called with: <__main__.InstanceCountingClass object at
0x101259e10> abc
>>> instance2 = InstanceCountingClass('xyz')
__new__() called with: <class '__main__.InstanceCountingClass'> ('xyz',) {}
__init__() called with: <__main__.InstanceCountingClass object at
0x101259dd8> xyz
>>> instance1.number, instance1.instances_created
(0, 2)
>>> instance2.number, instance2.instances_created
(1, 2)
```

Метод `__new__()` при нормальной работе должен возвращать экземпляр указанного класса, но может возвращать экземпляры и других классов. В таком случае вызов метода `__init__()` пропускается. Это полезно, когда есть необходимость изменить создание/инициализацию экземпляров неизменяемых классов (например, некоторых встроенных типов Python), как показано в следующем коде:

```
class NonZero(int):
    def __new__(cls, value):
        return super().__new__(cls, value) if value != 0 else None

    def __init__(self, skipped_value):
        # Имплементация __init_ в данном случае может быть пропущена,
        # однако она оставлена, чтобы вы увидели, как не стоит вызывать этот метод
        print("__init__() called")
        super().__init__()
```

Рассмотрим это в следующей интерактивной сессии:

```
>>> type(NonZero(-12))
__init__() called
<class '__main__.NonZero'>
>>> type(NonZero(0))
<class 'NoneType'>
>>> NonZero(-3.123)
__init__() called
-3
```

Когда же использовать `__new__()`? Ответ прост: только если одного `__init__()` недостаточно. Один из таких случаев мы уже упомянули — наследование от неизменяемых встроенных типов Python, например `int`, `str`, `float`, `frozenset` и т. д. Это связано с тем, что невозможно изменить такой экземпляр неизменяемого объекта в методе `__init__()` сразу после его создания.

Отдельные программисты считают, что метод `__new__()` пригоден для инициализации важного объекта, которая может быть пропущена, если пользователь забудет указать `super.__init__()` в переопределенном методе инициализации. Это звучит разумно, однако есть существенный недостаток. При таком подходе программист может явно пропустить предыдущие шаги по инициализации, если нужное поведение уже имеет место быть. Кроме того, это нарушает негласное правило всех инициализаций, выполненных в `__init__()`.

Поскольку `__new__()` не ограничен возвращением экземпляра именно того же класса, им легко злоупотребить. Безответственное использование данного метода может сильно повредить читабельность кода, поэтому его всегда следует применять с осторожностью и прикреплять обширную документацию. Как правило, лучше поискать другие имеющиеся решения данной задачи, в которых создание объекта не будет модифицировано так, что приведет к поломке. Даже переопределенную инициализацию неизменяемых типов можно заменить более предсказуемыми и хорошо зарекомендовавшими себя паттернами проектирования, один из которых мы рассмотрим в главе 17.

Существует по крайней мере один инструмент программирования на Python, в котором обширное использование `__new__()` вполне оправданно. Это метаклассы, и о них мы поговорим в следующем подразделе.

Метаклассы

Метаклассы — это особенность Python, которую многие разработчики считают одной из самых трудных для понимания и поэтому избегают ее. На самом деле все не так сложно, как кажется, стоит усвоить несколько основных понятий. Наградой будет знание того, как использовать метаклассы, и вы сможете делать то, что без них невозможно.

Метакласс — это тип (класс), определяющий другие типы (классы). Самое главное, что нужно знать для их понимания, — классы, которые определяют экземпляры объектов, тоже являются объектами. И поэтому у них есть соответствующий класс. Основной тип каждого класса по умолчанию просто встроенный класс `type`. Простая схема (рис. 5.1) помогает это понять.

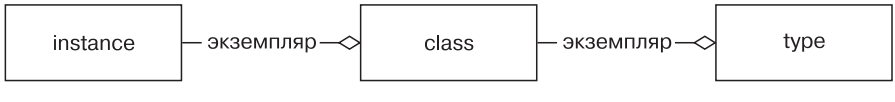


Рис. 5.1. Типизация класса

В Python можно заменить метакласс для объекта класса собственным типом. Как правило, новый метакласс — это все еще подкласс класса `type` (рис. 5.2), по-

скольку в противном случае полученные классы будут несовместимы с другими классами с точки зрения наследования.

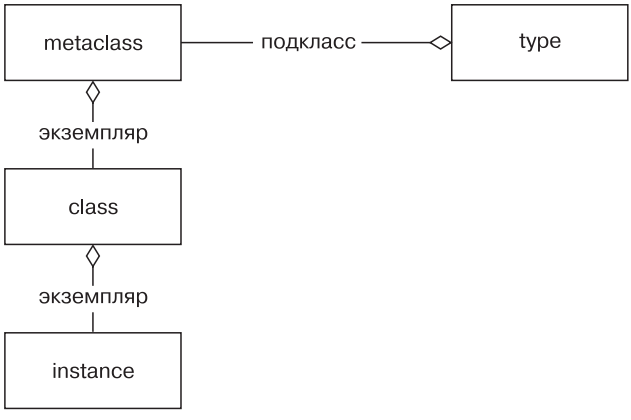


Рис. 5.2. Обычная реализация пользовательского метакласса

Рассмотрим общий синтаксис метаклассов.

Общий синтаксис

Вызов встроенного класса `type()` может использоваться в качестве динамического эквивалента объявления класса. Ниже приведен пример определения класса с вызовом `type()`:

```
def method(self):
    return 1

MyClass = type('MyClass', (object,), {'method': method})
```

Это эквивалентно явному определению класса с ключевым словом `class`:

```
class MyClass:
    def method(self):
        return 1
```

Каждый класс, который явно создается таким образом, имеет метакласс `type`. Такое поведение по умолчанию можно изменить, добавив именованный аргумент `metaclass`:

```
class ClassWithAMetaclass(metaclass=type):
    pass
```

Значение, предоставляемое в качестве аргумента `metaclass`, — это, как правило, еще один объект класса, но может быть любым другим вызываемым объектом,

который принимает те же аргументы, что и класс `type`, и возвращает другой объект класса. Сигнатура вызова такова: `type(name, bases, namespace)`. Значение аргументов выглядит следующим образом:

- `name` — имя класса, которое будет храниться в атрибуте `__name__`;
- `bases` — список родительских классов, которые станут атрибутом `__base__` и будут использоваться для построения ПРМ вновь созданного класса;
- `namespace` — пространство имен (отображение) с определениями для тела класса, который станет атрибутом `__dict__`.

Метаклассы — это своего рода метод `__new__()`, но на более высоком уровне определения класса.

Несмотря на то что вместо метаклассов можно добавить функции, которые явно вызывают `type()`, обычно для этого используется другой класс, наследующий от `type`. Общий шаблон для метакласса выглядит следующим образом:

```
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`, `namespace` имеют такое же значение, как и в `type()`, но все эти четыре метода могут иметь различные цели.

- Метод `__new__(mcs, name, bases, namespace)` отвечает за фактическое создание объекта класса, как и у обычных классов. Первый аргумент является объектом метакласса. В предыдущем примере это был бы просто `Metaclass`. Обратите внимание, что `mcs` — общепринятое имя для данного аргумента.
- Метод `__prepare__(mcs, name, bases, **kwargs)` создает пустой объект пространства имен. По умолчанию возвращает пустой `dict`, но может возвращать и любой другой тип отображения. Обратите внимание: он не принимает `namespace` в качестве аргумента, поскольку до вызова пространство имен еще не существует. Пример использования этого метода будет объяснен позже в пункте на с. 162.
- Метод `__init__(cls, name, bases, namespace, **kwargs)` не особо популярен в реализации метакласса, но имеет тот же смысл, что и в обычных классах.

Он может выполнять дополнительную инициализацию объекта класса, как только тот будет создан с помощью `__new__()`. Первый позиционный аргумент теперь называется `cls` и обозначает уже созданный объект класса (экземпляр метакласса), а не объект метакласса. В момент вызова `__init__()` класс уже был создан, и поэтому данный метод не так полезен, как `__new__()`. Реализация такого метода очень похожа на использование декораторов класса, но основное отличие состоит в том, что `__init__()` будет вызываться для каждого подкласса, а вот декораторы класса для подклассов не вызываются.

- Метод `__call__(cls, *arg, **kwargs)` вызывается, когда вызывается экземпляр метакласса. Последний является объектом класса (см. рис. 5.1), он вызывается при создании новых экземпляров класса. Метод позволяет переопределить способ создания и инициализации экземпляров класса.

Каждый из указанных выше методов может принимать дополнительные именованные аргументы, указанные в `**kwargs`. Эти аргументы могут быть переданы объекту метакласса с помощью дополнительных именованных аргументов в определении класса:

```
class Klass(metaclass=Metaclass, extra="value"):
    pass
```

Столько новой информации без соответствующих примеров — это многовато, так что посмотрим, как метаклассы, классы и экземпляры создаются с помощью вызовов `print()`:

```
class RevealingMeta(type):
    def __new__(mcs, name, bases, namespace, **kwargs):
        print(mcs, "__new__ called")
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print(mcs, "__prepare__ called")
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        print(cls, "__init__ called")
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        print(cls, "__call__ called")
        return super().__call__(*args, **kwargs)
```

Если мы используем `RevealingMeta` как метакласс и создаем новое определение класса, то результат интерактивной сессии Python будет следующим:

```
>>> 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 'RevealingMeta'> __prepare__ called
<class 'RevealingMeta'> __new__ called
<class 'RevealingClass'> __init__ called
>>> instance = RevealingClass()
<class 'RevealingClass'> __call__ called <class 'RevealingClass'> __new__
called <RevealingClass object at 0x1032b9fd0> __init__ called
```

Рассмотрим новый синтаксис Python 3 для работы с метаклассами.

Новый синтаксис метаклассов Python 3

Метаклассы уже не новинка, они были доступны в Python, начиная с версии 2.2. А вот их синтаксис существенно изменился, причем совместимость из-за этих изменений нарушилась в обе стороны. Новый синтаксис выглядит следующим образом:

```
class ClassWithAMetaclass(metaclass=type):
    pass
```

В Python 2 это выглядело так:

```
class ClassWithAMetaclass(object):
    __metaclass__ = type
```

Операторы классов в Python 2 не принимают именованные аргументы, поэтому в синтаксисе Python 3 для определения метаклассов в процессе импорта будет выброшено исключение `SyntaxError`. Метаклассы позволяют писать код, который будет работать на обеих версиях Python, но это требует дополнительных трудозатрат. К счастью, пакеты совместимости, например `six`, предоставляют готовые решения данной проблемы наподобие тех, что показаны в следующем коде:

```
from six import with_metaclass

class Meta(type):
    pass

class Base(object):
    pass

class MyClass(with_metaclass(Meta, Base)):
    pass
```

Другим важным отличием является отсутствие в метаклассах Python 2 хука `__prepare__()`. Реализация такой функции не выбрасывает никаких исключений в Python 2, но в ней нет смысла, поскольку ее не будут использовать для создания

чистого объекта пространства имен. Именно поэтому в пакетах совместимости с Python 2 используются более сложные трюки, результатом которых будет то же, что дает `__prepare__()`. Например, в Django REST Framework версии 3.4.7 (www.django-rest-framework.org), чтобы сохранить порядок, в котором атрибуты добавляются к классу, применяется следующий подход:

```
class SerializerMetaclass(type):
    @classmethod
    def _get_declared_fields(cls, bases, attrs):
        fields = [(field_name, attrs.pop(field_name))
                  for field_name, obj in list(attrs.items())
                  if isinstance(obj, Field)]
        fields.sort(key=lambda x: x[1]._creation_counter)

        # If this class is subclassing another Serializer, add
        # that Serializer's fields.
        # Note that we loop over the bases in *reverse*.
        # This is necessary in order to maintain the
        # correct order of fields.
        for base in reversed(bases):
            if hasattr(base, '_declared_fields'):
                fields = list(base._declared_fields.items()) + fields

        return OrderedDict(fields)

    def __new__(cls, name, bases, attrs):
        attrs['_declared_fields'] = cls._get_declared_fields(
            bases, attrs
        )
        return super(SerializerMetaclass, cls).__new__(
            cls, name, bases, attrs
        )
```

Это позволяет обойти проблему того, что тип пространства имен по умолчанию (`dict`) не гарантирует сохранение порядка кортежей «ключ — значение» в версиях Python старше 3.7 (мы говорили об этом в пункте «Словари» на с. 79). Атрибут `_creation_counter` ожидается в каждом экземпляре класса `Field`. Атрибут `Field.creation_counter` создается так же, как и `InstanceCountingClass.instance_number`, о котором мы говорили выше, в подразделе «Использование __new__() для переопределения процесса создания экземпляра». Это довольно сложное решение, которое нарушает единый принцип ответственности, так как его реализация разделена на два различных класса для отслеживания порядка атрибутов. В Python 3 все гораздо проще, поскольку `__prepare__()` может возвращать другие типы отображения, например `OrderedDict`, как показано в следующем коде:

```
from collections import OrderedDict

class OrderedMeta(type):
    @classmethod
```

```
    def __prepare__(cls, name, bases, **kwargs):
        return OrderedDict()
    def __new__(mcs, name, bases, namespace):
        namespace['order_of_attributes'] = list(namespace.keys())
        return super().__new__(mcs, name, bases, namespace)

class ClassWithOrder(metaclass=OrderedMeta):
    first = 8
    second = 2
```

Если вы проверите `ClassWithOrder` в интерактивной сессии, то увидите следующий вывод:

```
>>> ClassWithOrder.order_of_attributes
['__module__', '__qualname__', 'first', 'second']
>>> ClassWithOrder.__dict__.keys()
dict_keys(['__dict__', 'first', '__weakref__', 'second',
'order_of_attributes', '__module__', '__doc__'])
```

В следующем пункте мы поговорим об использовании метаклассов.

Использование метаклассов

Освоив работу с метаклассами, вы получите мощный инструмент, который, впрочем, всегда будет усложнять ваш код. Кроме того, они плохо объединяются, и вы быстро столкнетесь с проблемами, когда попробуете смешать несколько метаклассов через наследование.

Для простых задач вроде изменения атрибутов чтения/записи или добавления новых атрибутов можно отказаться от метаклассов в пользу более простых решений, таких как свойства, дескрипторы или декораторы класса.

Но бывают ситуации, когда без метаклассов не обойтись. Например, трудно представить себе реализацию ORM в Django без широкого использования метаклассов. Это возможно, но маловероятно, что результат вообще будет пригоден для использования. А вот в фреймворках метаклассы действительно работают прекрасно. Как правило, в них много сложного для понимания внутреннего кода, но в конечном счете это позволит другим программистам писать более концентрированный и читабельный код, работающий на более высоком уровне абстракции.

Рассмотрим кое-какие ограничения, связанные с применением метаклассов.

Ловушки метаклассов

Как и некоторые другие расширенные функции Python, метаклассы очень эластичны и с ними легко переборщить. Синтаксис вызова класса достаточно строгий, но Python не задает тип возвращаемого параметра. Он может быть каким угодно, пока класс принимает заданные при вызове аргументы и имеет нужные атрибуты, когда это необходимо.

Одним из таких объектов, который может быть *чем угодно и где угодно*, является экземпляр класса `Mock`, предоставляемый в модуле `unittest.mock`. `Mock` — это не метакласс, он не наследует от `class`. Он также не возвращает объект класса при создании экземпляра. Тем не менее он может быть включен в качестве именованного аргумента метакласса при его определении, и это не вызывает никаких синтаксических ошибок. Использование `Mock` как метакласса — это полная ерунда, но рассмотрим следующий пример:

```
>>> from unittest.mock import Mock
>>> class Nonsense(metaclass=Mock): # pointless, but illustrative
...     pass
...
>>> Nonsense
<Mock spec='str' id='4327214664'>
```

Нетрудно предвидеть, что любая попытка создать экземпляр придуманного нами псевдокласса `Nonsense` потерпит неудачу. Любопытно, что результат получится таким:

```
>>> Nonsense()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/unittest/
    mock.py", line 917, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/unittest/
    mock.py", line 976, in _mock_call
    result = next(effect)
StopIteration
```

Дает ли исключение `StopIteration` хоть малый шанс понять, что проблема возникла именно в определении класса на уровне метакласса? Очевидно, нет. Этот пример показывает, как трудно отлаживать код метакласса, если вы не знаете, где искать ошибки.

В следующем подразделе поговорим о генерации кода.

Генерация кода

Как мы уже упоминали, динамическая генерация кода — это наиболее сложный подход к метапрограммированию. В Python есть инструменты, которые позволяют создавать и выполнять код или даже вносить изменения в уже скомпилированные части кода.

По проектам наподобие *Hy* (о нем позже поговорим отдельно) видно, что с помощью методов генерации кода Python можно переопределять целые языки. То есть возможности этого инструментария практически безграничны. Мы знаем,

насколько сложна данная тема и сколько в ней скрытых подвохов, поэтому даже не будем пытаться приводить примеры кода или давать вам рекомендации о том, как с этим работать.

Но вам будет полезно знать, что это в принципе возможно, если вы планируете самостоятельно заняться данной работой позже. Считайте этот подраздел указанием к действию для дальнейшего обучения.

Посмотрим, как использовать функции `exec`, `eval` и `compile`.

exec, eval и compile

В Python есть три встроенные функции, позволяющие вручную выполнить, вычислить и скомпилировать произвольный код Python.

- `exec(object, global, locals)` — позволяет динамически выполнять код Python. Элемент `object` должен быть строкой или объектом кода (см. функцию `compile()`), представляющим один оператор или последовательность нескольких. Аргументы `global` и `local` — это глобальные и локальные пространства имен для исполняемого кода, которые не являются обязательными. Если они не указаны, то код выполняется в текущем пространстве. Если указаны, то `global` должен быть словарем, а `local` может быть любым объектом отображения, он всегда возвращает `None`.
- `eval(expression, global, locals)` — используется для вычисления данного выражения и возвращает его значение. Похоже на `exec()`, но `expression` — это всего одно выражение Python, а не последовательность операторов. Возвращает значение вычисленного выражения.
- `compile(source, filename, mode)` — компилирует источник в объект кода или AST. Исходный код предоставляется в качестве строкового значения в аргументе `source`. `filename` — это файл, из которого читается код. Если связанного файла нет (например, потому что он был создан динамически), обычно используется значение `<string>`. Режим — `exec` (последовательность операторов), `eval` (одно выражение) или `single` (один интерактивный оператор, например, в интерактивной сессии Python).

Начать работу с функциями `exec()` и `eval()` легче всего с динамической генерации кода, поскольку функции работают со строками. Если вы уже знаете, как программировать на Python, то знаете и то, как правильно сформировать рабочий исходный код из программы.

Наиболее полезной в контексте метапрограммирования функцией будет, очевидно, `exec()`, поскольку она позволяет выполнять любую последовательность операторов Python. Здесь вас должно тревожить слово «любую». Даже функция `eval()`, которая в руках умелого программиста позволяет вычислять выражения (при подаче на вход пользовательского ввода), может привести к проблемам с без-

опасностью. И сбоя интерпретатора Python здесь стоит бояться меньше всего. Вводя уязвимость в удаленное выполнение в виде этих функций, вы рискуете своей репутацией и карьерой.

Даже если вы доверяете входным данным, список мелких проблем с `exec()` и `eval()` все еще слишком велик, а результаты работы вашей программы будут весьма неожиданными. Армин Ронакер написал хорошую статью *Be careful with exec and eval in Python*, в которой перечислены наиболее важные из них (см. lucumr.pocoo.org/2011/2/1/exec-in-python/).

Несмотря на все эти пугающие предупреждения, существуют естественные ситуации, когда использование `exec()` и `eval()` действительно оправданно. Тем не менее, в случае даже малейших сомнений лучше всего отказаться от этих функций и попытаться найти другое решение.

eval() и ненадежный ввод

Сигнатура функции eval() наводит на мысль о том, что если вы дадите ей пустые globals и locals и обернете ее оператором try...except, то все будет достаточно безопасно. Но это огромная ошибка. Нед Батчелер написал очень хорошую статью, в которой показывает, как вызвать ошибку сегментации интерпретатора с помощью вызова eval() (см. nedbatchelder.com/blog/201206/eval_really_is_dangerous.html). Это доказательство того, что exec() и eval() никогда не должны использоваться с ненадежными входными данными.

В следующем пункте рассмотрим абстрактное синтаксическое дерево.

Абстрактное синтаксическое дерево

Синтаксис Python преобразуется в AST до компиляции в байт-код. Это представление абстрактной синтаксической структуры исходного кода в виде дерева. Обработка грамматики в Python реализуется через встроенный модуль `ast`. Сырые AST кода Python создаются с помощью функции `compile()` с флагом `ast.PyCF_ONLY_AST` или с использованием помощника `ast.parse()`. Трансляция в обратном направлении — сложная задача, и в стандартной библиотеке нет функций, которые позволяют ее реализовать. Но в некоторых проектах, например в PyPy, такие функции есть.

Модуль `ast` предоставляет некоторые вспомогательные функции, позволяющие работать с AST, например:

```
>>> tree = ast.parse('def hello_world(): print("hello world!")')
>>> tree
<_ast.Module object at 0x00000000038E9588>
>>> ast.dump(tree)
"Module(
```

```
    body=[
        FunctionDef(
            name='hello_world',
            args=arguments(
                args=[],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[]
            ),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[Str(s='hello world!')],
                        keywords=[]
                    )
                )
            ],
            decorator_list=[],
            returns=None
        )
    ]
)"
```

Выход `ast.dump()` в предыдущем примере был переформатирован с целью улучшить читабельность и лучше показать древовидную структуру AST. Важно знать, что AST можно изменить до передачи в функцию `compile()`. Это дает много новых возможностей. Так, новые узлы синтаксиса могут быть использованы для, например, измерения области охвата теста. Кроме того, можно изменить имеющееся дерево кода, чтобы добавить новую семантику к существующему синтаксису. Такой метод используется в рамках проекта MacroPy (github.com/lihaoyi/macropy) для добавления в Python синтаксических макросов с помощью уже существующего синтаксиса (рис. 5.3).

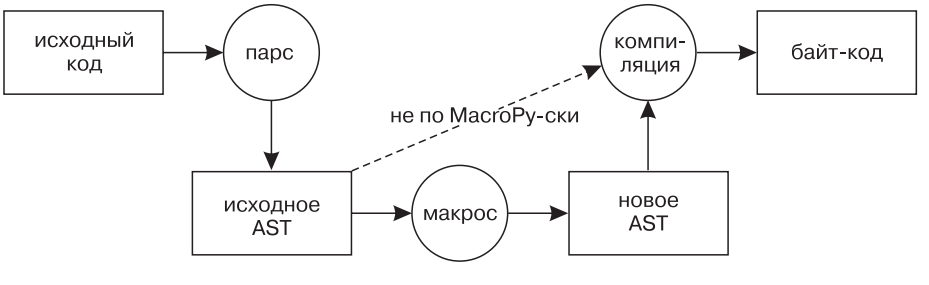


Рис. 5.3. Как MacroPy добавляет синтаксические макросы в модуль Python при импорте

AST также можно создавать чисто искусственным образом, и нет необходимости парсить какой-либо источник на всех. Это дает Python-программистам возможность создавать байт-код Python для пользовательских предметно-ориентированных языков или даже полностью реализовать другие языки программирования поверх Python.

Хуки импорта. Воспользоваться способностью MacroPy изменять оригинальную AST позволит оператор `import macropy.activate`, если каким-то образом сможет переопределить импорт Python. К счастью, в Python есть способ перехвата импорта с помощью следующих двух видов хуков.

- *Метахуки* вызываются до обработки `import`. Использование метахуков позволяет изменить способ обработки `sys.path` даже для замороженных и встроенных модулей. Чтобы добавить новый метахук, нужно добавить в список `sys.meta_path` новый объект *meta path finder*.
- *Хуки пути импорта* вызываются как часть обработки `sys.path`. Они используются, если встречается элемент пути, связанный с данным хуком. Хуки пути импорта добавляются путем расширения списка `sys.path_hooks` новым объектом *path finder*.

Подробности реализации path finders детально описаны в официальной документации Python (см. docs.python.org/3/reference/import.html). Она должна быть вашим основным ресурсом, если вы хотите взаимодействовать с импортом на данном уровне. Это объясняется тем, что механизм импорта в Python довольно сложен и любая попытка обобщить его неизбежно терпит неудачу. Здесь мы лишь отметили, что такие вещи в принципе возможны.

Ниже рассмотрим проекты, в которых используются паттерны генерации кода.

Проекты, в которых используются паттерны генерации кода

Трудно найти действительно полезную реализацию библиотеки, которая основана на паттернах генерации кода и при этом будет реально рабочей, а не экспериментом фанатов. Причины этой ситуации достаточно очевидны:

- обоснованный страх перед функциями `exec()` и `eval()`, поскольку их неправильное использование может привести к катастрофе;
- успешную генерацию кода очень трудно разрабатывать и поддерживать, так как это требует глубокого понимания языка и серьезных навыков программирования в целом.

Несмотря на эти трудности, есть ряд проектов, в которых данный подход успешно используется либо для повышения производительности, либо для достижения того, что было бы невозможно получить с помощью других средств.

Маршрутизация на Falcon. Falcon (falconframework.org) — это минималистский веб-фреймворк Python WSGI для создания быстрых и облегченных API. В нем используется архитектурный стиль REST, который сегодня весьма популярен во всем Интернете. Это хорошая альтернатива другим достаточно тяжелым движкам наподобие Django или Pyramid. Он также составляет сильную конкуренцию другим микрофреймворкам, стремящимся к простоте, таким как Flask, Bottle или web2py.

Одна из особенностей этого фреймворка — очень простой механизм маршрутизации. Она будет проще, чем маршрутизация на `urlconf` Django, хотя и менее функциональна. Однако в большинстве случаев и этого функционала достаточно для любого API, построенного на архитектуре REST. Самое интересное в маршрутизации Falcon — это внутреннее устройство маршрутизатора. Он реализован с помощью кода, сгенерированного из списка маршрутов, и код меняется каждый раз, когда регистрируется новый маршрут. Именно этот трюк позволяет ускорить маршрутизацию.

Рассмотрим короткий пример API, взятый из веб-документации Falcon:

```
# sample.py
import falcon
import json
class QuoteResource:
    def on_get(self, req, resp):
        """Обработка GET-запросов"""
        quote = {
            'quote': 'I\'ve always been more interested in '
                     'the future than in the past.',
            'author': 'Grace Hopper'
        }
        resp.body = json.dumps(quote)
api = falcon.API()
api.add_route('/quote', QuoteResource())
```

Выделенный вызов метода `api.add_route()` динамически обновляет все сгенерированное дерево кода по запросу маршрутизатора Falcon. Он также собирает его с помощью функции `compile()` и генерирует новую функцию поиска маршрута посредством `eval()`. Внимательнее посмотрим на атрибут `__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>
```

Результат показывает, что код этой функции был сгенерирован из строки, а не из реального исходного кода (файла `"<string>"`). Кроме того, видно, что факти-

ческий объект кода изменяется с каждым вызовом метода `api.add_route()` (адрес объекта в памяти изменяется).

Hy. Hy (docs.hylang.org) — это диалект Lisp, полностью написанный на Python. Многие подобные проекты, которые реализуют другой код в Python, обычно стремятся к простой форме кода, хранимого в виде файлоподобного объекта или строки и интерпретируемого как последовательность явных вызовов на Python. В отличие от других, Hy — это язык, который полностью работает в среде выполнения Python, как и собственно Python. Код, написанный на Hy, может использовать существующие встроенные модули и внешние пакеты, и, наоборот, код, написанный на Hy, можно импортировать обратно в Python.

Чтобы вставить Lisp в Python, Hy переводит код Lisp непосредственно в Python AST. Совместимость на этапе импорта достигается с помощью хука импорта, который регистрируется в тот момент, когда модуль Hy импортируется в Python. Любой модуль с расширением `.hy` рассматривается как модуль Hy и может быть импортирован как обычный модуль Python. Ниже показан стандартный *Hello World* на этом диалекте Lisp:

```
;; hyllo.hy
(defn hello [] (print "hello world!"))
```

Этот код можно импортировать и выполнить с помощью следующего кода Python:

```
>>> import hy
>>> import hyllo
>>> hyllo.hello()
    hello world!
```

Если копнуть глубже и попытаться разобрать `hyllo.hello` с помощью встроенного модуля `dis`, то мы заметим, что байт-код функции Hy не слишком отличается от своего чистого аналога Python, как показано в следующем коде:

```
>>> 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
```

Резюме

В этой главе мы рассмотрели обширную тему метапрограммирования в Python. Мы подробно обсудили особенности синтаксиса, с помощью которых можно реализовать различные паттерны метапрограммирования. В основном это декораторы и метаклассы.

Кроме того, мы рассмотрели еще один важный аспект метапрограммирования — динамическое генерирование кода. Эта тема слишком обширна, чтобы ее можно было тщательно обсудить на страницах данной книги, поэтому мы затронули ее лишь слегка. Тем не менее предоставленная информация может стать для вас хорошей отправной точкой для дальнейшего изучения этой области.

В следующей главе мы немного отдохнем от сложных вещей и поговорим об общепринятых методах присвоения имен.

6 Как выбирать имена

Большая часть стандартной библиотеки была построена с учетом требований к юзабилити. В этом смысле Python можно сравнить с псевдокодом, который возникает в вашей голове, когда вы работаете над программой. Почти весь код можно прочитать вслух. Например, следующий фрагмент будет понятен даже тем, кто далек от программирования:

```
my_list = []
if 'd' not in my_list:
    my_list.append('d')
```

Код на Python близок к естественному языку, и это — одна из причин того, почему Python настолько прост в освоении и использовании. Когда вы пишете программу, поток ваших мыслей быстро превращается в строки кода.

В данной главе мы сосредоточимся на практических рекомендациях по написанию кода, который легко понять и применять, а именно:

- использование соглашения об именовании, описанного в PEP 8.
- выдача рекомендаций по присвоению имен;
- краткий обзор популярных инструментов, которые позволяют проверить ваш код на соответствие требованиям стиля.

В этой главе:

- PEP 8 и практические рекомендации по именованию;
- стили именования;
- руководство по именованию;
- рекомендации для аргументов;
- имена классов;
- имена модулей и пакетов;
- полезные инструменты.

Технические требования

Ниже приведены пакеты Python, упомянутые в этой главе, которые можно скачать с PyPI:

- `pylint`;
- `pycodestyle`;
- `flake8`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter6.

PEP 8 и практические рекомендации по именованию

В документе PEP 8 (www.python.org/dev/peps/pep-0008) приведено руководство по стилю написания кода на Python. Помимо базовых правил, например, касающихся отступов, максимальной длины строки, и других правил размещения кода, в PEP 8 есть раздел, посвященный соглашениям об именовании, которым следует большинство кодовых баз.

Текущий раздел нашей книги содержит лишь краткий обзор PEP 8 и удобный путеводитель по именованию для каждого типа синтаксиса Python. И тем не менее всем Python-программистам нужно в обязательном порядке прочитать документ PEP 8.

Почему и когда надо соблюдать PEP 8

Если вы создаете новый программный пакет, предназначенный для работы с открытым исходным кодом, обязательно нужно следовать PEP 8, поскольку это широко распространенный стандарт, который используется в большинстве проектов с открытым исходным кодом, написанных на Python. Если вы хотите как-либо сотрудничать с другими программистами, то должны обязательно придерживаться PEP 8, даже если ваше мнение относительно оформления кода расходится с тем, что приведено в данном документе. Следование изложенным в нем правилам позволяет другим разработчикам быстро и без проблем разобраться в вашем проекте. Код будет более читабельным для новичков благодаря согласованности его стиля с большинством других пакетов Python с открытым исходным кодом.

Кроме того, сразу начиная полностью соблюдать требования PEP 8, вы сэкономите время и избавитесь от некоторых проблем в будущем. Если вы захотите выпустить свой код в массы, то рано или поздно коллеги-программисты все равно предложат вам перейти на PEP 8. Споры о том, действительно ли это необходимо делать для конкретного проекта, — источник длинных и безрезультатных споров. И, как ни печально, в конечном итоге вам придется уступить, иначе с вами просто не захотят сотрудничать.

Рестайлинг базового кода большого проекта может потребовать огромного количества времени и усилий. В некоторых случаях придется редактировать почти каждую строку кода. Большинство изменений можно автоматизировать (отступы, разрывы строк, замыкающие пробелы). Тем не менее такой капитальный ремонт кода, как правило, приводит к куче багов и конфликтов в каждом рабочем процессе контроля версий. Кроме того, будет очень трудно охватить столько изменений сразу. Именно поэтому во многих проектах с открытым исходным кодом есть правило, согласно которому изменение стиля кода выпускается отдельным патчем, чтобы оно не влияло на функционал и не вызывало багов.

За пределами PEP 8 — правила стиля внутри команды

Несмотря на внушительный набор указаний по стилю, PEP 8 все-таки дает разработчикам немного свободы. Особенно это касается вложенных литералов данных и многопоточных вызовов функций, требующих длинных списков аргументов. Некоторые команды могут решить, что им нужны дополнительные правила оформления, и даже выпускают свой документ, который регламентирует это и доступен для каждого ее участника.

Кроме того, в ряде ситуаций может быть нереально или экономически недостижимо полностью перевести на PEP 8 старые проекты, в которых не были определены четкие правила. Таким проектам не помешает какое-нибудь более строгое оформление кода, пусть это и не будет именно набор правил PEP 8. Помните: согласованность в рамках проекта даже важнее, чем соблюдение PEP 8. Если у каждого программиста есть понятные и четко описанные правила, то будет намного легче поддерживать согласованность в рамках проекта и организации.

В следующем разделе рассмотрим различные стили именования.

Стили именования

Стили именования, используемые в Python:

- ВерблюжийРегистр;
- смешанныйСтиль;

- ВЕРХНИЙРЕГИСТР и ВЕРХНИЙ_РЕГИСТР_С_ПОДЧЕРКИВАНИЯМИ;
- нижнийрегистр и нижний_регистр_с_подчеркиваниями;
- подчеркивание _до и после_, либо __двойное__.

Строчные и прописные элементы часто представляют собой одно слово, а иногда и несколько сочлененных слов. Будучи подчеркнутыми, они обычно являются сокращенными фразами. Лучше использовать одно слово. Подчеркивание до и после служит для обозначения приватности и специальных элементов.

Эти стили применяются к следующим объектам:

- переменные;
- функции и методы;
- свойства;
- классы;
- модули;
- пакеты.

Переменные

В Python существует два вида переменных:

- **константы** содержат значения, которые не должны изменяться во время выполнения программы;
- **публичные и приватные переменные** содержат значения, которые могут изменяться во время выполнения программы.

Константы

Для неизменяемых глобальных переменных используется верхний регистр с подчеркиванием. Такой стиль сообщает разработчику, что данная переменная — константа.

В Python нет констант, как, например, в C++, где можно использовать const. В Python вы можете изменить значение любой переменной. Поэтому в Python для обозначения констант используется другой стиль именования.

Например, модуль `doctest` предоставляет список флагов опций и директив (docs.python.org/lib/doctest-options.html) — это небольшие предложения, четко определяющие, для чего предназначен каждый вариант, допустим:

```
from doctest import IGNORE_EXCEPTION_DETAIL
from doctest import REPORT_ONLY_FIRST_FAILURE
```

Эти имена переменных кажутся длинноватыми, но важно четко описать их. Они используются в основном в разделе инициализации, а не в теле самого кода, так что их многословность не слишком раздражает.

Сокращенные имена обычно только вносят путаницу. Не бойтесь использовать полные слова, если аббревиатура кажется неясной.

Кроме того, имена некоторых констант определяются их базовой технологией. Так, в модуле `os` есть константы, определенные на стороне C, например серия `EX_XXX`, которая определяет номера выходных кодов UNIX. То же кодовое имя можно найти, как в следующем примере, в файлах заголовков `sysexits.h`:

```
import os
import sys
sys.exit(os.EX_SOFTWARE)
```

Еще один хороший прием использования констант: собирать их все в верхней части модуля, в котором они применяются. Кроме того, их часто объединяют под новыми переменными, если они представляют собой флаги или перечисления, которые позволяют выполнить такие операции, как в примере ниже:

```
import doctest
TEST_OPTIONS = (doctest.ELLIPSIS |
                doctest.NORMALIZE_WHITESPACE |
                doctest.REPORT_ONLY_FIRST_FAILURE)
```

Далее обсудим именование и использование констант.

Именование и использование

Константы служат для определения набора значений, которые использует программа, например имени файла конфигурации по умолчанию.

Хорошим приемом будет собрать все константы в одном файле в пакете. Так работает Django, например. Модуль с именем `settings.py` содержит все константы:

```
# config.py
SQL_USER = 'tarek'
SQL_PASSWORD = 'secret'
SQL_URI = 'postgres://%s:%s@localhost/db' % (
    SQL_USER, SQL_PASSWORD
)
MAX_THREADS = 4
```

Другой подход заключается в использовании файла конфигурации, который можно проанализировать с помощью модуля `ConfigParser` или другого инструмента парсинга конфигурации. Некоторые, впрочем, считают излишеством применять другой формат файла в языках, подобных Python, где исходный файл редактируется столь же легко, как текстовый.

Переменные-флаги обычно объединяются с логическими операциями, как это делается в модулях `doctest` и `re`. Паттерн, взятый из `doctest`, довольно прост, и это видно по следующему коду:

```
OPTIONS = {}

def register_option(name):
    return OPTIONS.setdefault(name, 1 << len(OPTIONS))

def has_option(options, name):
 return bool(options & name)

# Определение опций
BLUE = register_option('BLUE')
RED = register_option('RED')
WHITE = register_option('WHITE')
```

Данный код позволяет делать вот так:

```
>>> # Попробуем их
>>> SET = BLUE | RED
>>> has_option(SET, BLUE)
True
>>> has_option(SET, WHITE)
False
```

При определении нового набора констант следует избегать использования общего префикса, если в модуле нет нескольких независимых наборов опций. Само название модуля является общим префиксом.

Еще одно хорошее решение для констант-опций — задействовать класс `Enum` от встроенного модуля `enum` и просто применять множества вместо бинарных операторов. Подробности использования и синтаксис модуля `enum` см. в главе 3.

Использование двоичных побитовых операций для объединения опций — обычное дело в Python. Оператор OR (|) позволит объединить несколько опций в одну, а AND (&) — проверить, присутствует ли опция в целом числе (см. функцию has_option()).

В следующем пункте поговорим о публичных и приватных переменных.

Публичные и приватные переменные

Для глобальных переменных, которые являются изменяемыми и свободно доступны через импорт, следует применять строчные буквы с подчеркиванием в ситуациях, когда не требуется защита. Если переменная не должна использоваться

и модифицироваться за пределами порождающего модуля, то мы считаем ее приватной для него. В этом случае начальное подчеркивание помечает переменную как приватный элемент пакета, что и показано в следующем коде:

```
_observers = []
def add_observer(observer):
    _observers.append(observer)
def get_observers():
    """Убеждаемся, что _observers нельзя изменить"""
    return tuple(_observers)
```

Переменные в функциях и методах следуют тем же правилам, что и публичные переменные, и никогда не помечаются как приватные, поскольку являются локальными в контексте функции.

Для переменных класса или экземпляра нужно использовать маркер приватности (начальное подчеркивание), если включение переменной в состав публичной сигнатуры не несет никакой полезной информации или является излишним. Иными словами, если переменная служит только для внутренних целей метода, который предоставляет публичную функцию, то лучше сделать ее приватной.

Например, атрибуты свойств — это приватные переменные, как показано в следующем коде:

```
class Citizen(object):
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"
```

Другой пример — переменная, сохраняющая некое внутреннее состояние, которое не должно быть раскрыто другим классам. Это значение не является полезным для остального кода, но участвует в поведении класса:

```
class UnforgivingElephant(object):
    def __init__(self, name):
        self.name = name
        self._people_to_stomp_on = []

    def get_slapped_by(self, name):
        self._people_to_stomp_on.append(name)
        print('Ouch!')

    def revenge(self):
        print('10 years later...')
        for person in self._people_to_stomp_on:
            print('%s stomps on %s' % (self.name, person))
```

Вот что вы увидите в интерактивной сессии:

```
>>> joe = UnforgivingElephant('Joe')
>>> joe.get_slapped_by('Tarek')
Ouch!
>>> joe.get_slapped_by('Bill')
Ouch!
>>> joe.revenge()
10 years later...
Joe stomps on Tarek
Joe stomps on Bill
```

Рассмотрим именование функций и методов.

Функции и методы

Функции и методы пишутся в нижнем регистре с подчеркиванием. Но данное правило не всегда выполняется в старых модулях стандартной библиотеки. В стандартной библиотеке Python 3 было сделано немало изменений, поэтому у большинства функций и методов регистр букв переделан на «правильный». Тем не менее в ряде модулей, таких как `threading`, еще присутствуют функции со старыми именами, в которых используется смешанный регистр (например, `currentThread`). Подобные функции были оставлены без изменений, чтобы обеспечить обратную совместимость, но если вам не требуется запускать ваш код в более старых версиях Python, то следует избегать использования этих старых имен.

Именно такое написание было распространено до того, как нижний регистр стал стандартом, и в некоторых фреймворках, например Zope и Twisted, названия методов до сих пор пишутся в *смешанном регистре*. Сообщество программистов, работающих с этими фреймворками, по-прежнему достаточно велико. Таким образом, выбор между данным написанием и строчными буквами с подчеркиванием, безусловно, зависит от набора библиотек, которые вы используете.

Разработчикам Zope трудновато соблюдать общие правила именования, поскольку довольно сложно создать приложение, в котором сочетаются чистый Python и импортированные модули Zope. В ряде классов Zope правила именования смешиваются, поскольку базовый код по-прежнему развивается, и разработчики Zope пытаются перейти к использованию общепринятых соглашений.

Хорошим приемом в таких библиотечных средах является использование смешанного регистра только для элементов, которые применяются в фреймворках, и сохранение остальной части кода в стиле PEP 8.

Отметим, что разработчики проекта Twisted используют совершенно иной подход к этой проблеме. Проект Twisted, так же как Zope, возник еще до документа PEP 8. В те времена еще не было никаких официальных руководящих принципов по стилю кода Python, и в данном проекте были собственные правила. Стилистические правила, касающиеся отступов, строк документации, длины строк и т. д., легко можно адаптировать. А вот обновление всего кода в соответствии с соглашениями

об именованиях PEP 8 приведет к полному нарушению обратной совместимости. Допускать подобное для такого крупного проекта, как Twisted, нельзя. Поэтому в Twisted PEP 8 был принят настолько, насколько это возможно, а смешанный регистр остался для переменных, функций и методов в рамках собственного стандарта проекта. Но это полностью отвечает PEP 8, поскольку соответствие внутри проекта важнее, чем соответствие указаниям PEP 8.

Споры о приватности

Для приватных методов и функций обычно используется одно начальное подчеркивание. Это просто соглашение об именовании, и оно не имеет синтаксического смысла. Но это не значит, что начальные подчеркивания вообще его не имеют. Когда в названии метода есть два начальных подчеркивания, он переименовывается интерпретатором динамически, чтобы предотвратить конфликт имен с методом из любого подкласса. Эта функция Python называется *искажением (декорированием) имен*.

Некоторые разработчики часто используют двойное подчеркивание для именования приватных атрибутов, чтобы избежать конфликтов имен в подклассах, например:

```
class Base(object):
    def __secret(self):
        print("don't tell")

    def public(self):
        self.__secret()

class Derived(Base):
    def __secret(self):
        print("never ever")
```

Вывод кода будет следующим:

```
>>> Base.__secret
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: type object 'Base' has no attribute '__secret'
>>> dir(Base)
['_Base__secret', ..., 'public']
>>> Base().public()
don't tell
>>> Derived().public()
don't tell
```

Причиной введения декорирования имен в Python было не желание создать примитивную изоляцию, как у слова `private` в C++, а желание убедиться, что в базовых классах будут неявно устранены конфликты в подклассах, особенно если они предназначены для использования в различных контекстах наследования (например, в примесях). Применение такого декорирования для каждого атрибута,

который не является публичным, ведет к путанице в коде и делает его чрезвычайно трудным для расширения. А это не в стиле Python.

Дополнительную информацию по этой теме можно почитать в рассылке Python, которая выходила много лет назад, когда люди рассуждали о пользе декорирования имен и его судьбе в языке: mail.python.org/pipermail/python-dev/2005-December/058555.html.

Рассмотрим стили именования специальных методов.

Специальные методы

Имена специальных методов (docs.python.org/3/reference/datamodel.html#specialmethodnames) начинаются и заканчиваются двойным подчеркиванием и образуют так называемые протоколы языка (см. главу 4). Некоторые разработчики привыкли называть их методами *dunder* из-за двойного подчеркивания. Они используются для перегрузки операторов, определения контейнеров и т. д. В целях читабельности их нужно собирать в начале определения класса, как показано в следующем коде:

```
class WeirdInt(int):
    def __add__(self, other):
        return int.__add__(self, other) + 1

    def __repr__(self):
        return '<weirdo %d>' % self

    # Публичный API
    def do_this(self):
        print('this')

    def do_that(self):
        print('that')
```

Данное соглашение не должен применять ни один пользовательский метод, если ему явно не требуется реализовать один из протоколов объектов Python. Так что не придумывайте собственные методы наподобие этого:

```
class BadHabits:
    def __my_method__(self):
        print('ok')
```

Далее поговорим о правилах именования аргументов.

Аргументы

Аргументы именуются в нижнем регистре, с подчеркиванием, если это необходимо. К ним применяются те же правила именования, что и к переменным, поскольку аргументы — это всего лишь локальные переменные, которые получают свое значение при вызове функций. В следующем примере `text` и `separator` являются аргументами функции `one_line()`:

```
def one_line(text, separator=" "):
    """Многострочный текст объединяется в одну строку"""
    return separator.join(text.split())
```

Перейдем к стилю именования свойств.

Свойства

Имена свойств пишутся в нижнем регистре или в нижнем регистре с подчеркиванием. В основном они представляют состояние объекта (это может быть существительное, или прилагательное, или небольшая фраза, когда это необходимо). В следующем примере кода класс `Contatiner` представляет собой простую структуру данных, которая может возвращать копии своего содержимого через свойства `unique_items` и `ordered_items`:

```
class Container:
    _contents = []

    def append(self, item):
        self._contents.append(item)

    @property
    def unique_items(self):
        return set(self._contents)

    @property
    def ordered_items(self):
        return list(self._contents)
```

Рассмотрим стили именования классов.

Классы

Имена классов всегда пишутся в верблюжьем регистре и могут иметь подчеркивание, когда являются приватными внутри модуля.

В объектно-ориентированном программировании классы используются для инкапсуляции состояния приложения. Атрибуты объектов представляют своего рода записи этих состояний. Методы применяются для изменения состояния, их преобразования в понятные по смыслу значения или для побочных действий. Именно поэтому имена классов — это, как правило, существительные или фразы, а логика их использования формируется через имена методов — глагольные конструкции. Следующий пример кода содержит определение класса `Document` с методом `save()`:

```
class Document():
    file_name: str
    contents: str
    ...
```

```
    def save(self):
        with open(self.file_name, 'w') as file:
            file.write(self.contents)
```

Экземпляры классов часто задействуют те же именные конструкции, что и документ, но пишутся в нижнем регистре. Таким образом, использование класса `Document` может выглядеть следующим образом:

```
new_document = Document()
new_document.save()
```

Рассмотрим стили именования для модулей и пакетов.

Модули и пакеты

Все модули, кроме специального модуля `__init__`, именуются в нижнем регистре. Ниже приведены некоторые примеры из стандартной библиотеки:

- `os`;
- `sys`;
- `shutil`.

В стандартной библиотеке Python для разделения слов в именах модулей подчеркивания не используются, чего не скажешь о многих других проектах. Когда модуль является приватным в рамках пакета, добавляется начальное подчеркивание. Модули Compiled C или C++ обычно именуются с подчеркиванием и импортируются в чистые модули Python. Имена пакетов подчиняются тем же правилам, поскольку пакеты работают скорее как структурированные модули.

В следующем разделе поговорим о других правилах именования.

Руководство по именованию

Общий набор правил именования может быть применен к переменным, методам, функциям и свойствам. Имена классов и модулей играют весьма важную роль в построении пространства имен и оказывают сильное влияние на читабельность кода. В этом разделе приведено мини-руководство, которое поможет вам определить значимые и читабельные имена для элементов кода.

Использование префиксов is/has в булевых элементах

Когда элемент содержит булево значение, логично добавить в имя префикс `is` и/или `has`, чтобы сделать переменную более читабельной. В следующем примере идентификаторы `is_connected` и `has_cache` содержат логические состояния экземпляров класса `DB`:

```
class DB:
    is_connected = False
    has_cache = False
```

Использование множественного числа в именах коллекций

Когда элемент содержит последовательность, бывает удобно использовать форму множественного числа. То же самое можно делать и для различных переменных отображения и свойств. В следующем примере `connected_users` и `tables` — это атрибуты класса, которые содержат несколько значений:

```
class DB:
    connected_users = ['Tarek']
    tables = {'Customer':['id', 'first_name', 'last_name']}
```

Использование явных имен для словарей

Когда переменная содержит отображение, по возможности нужно использовать явное имя. Например, если `dict` содержит адрес человека, то его следует назвать `persons_addresses`:

```
persons_addresses = {'Bill': '6565 Monty Road',
                     'Pamela': '45 Python street'}
```

Избегайте встроенных и избыточных имен

Обычно следует избегать использования в именах слов `list`, `dict` и `set`, причем даже для локальных переменных. Python теперь предлагает аннотации функций и переменных, а также иерархию типов, что позволяет явно указывать ожидаемый тип для данной переменной, вследствие чего больше нет необходимости описывать типы объектов в их именах. Это делает код трудным для чтения, понимания и использования. Кроме того, следует избегать применения встроенных имен, чтобы не возникало переопределение в текущем пространстве имен. Общих глаголов тоже следует избегать, если они не имеют значения в пространстве имен.

Лучше использовать термины, связанные с вашей задачей:

```
def compute(data):  # Так примитивно
    for element in data:
        yield element ** 2

def squares(numbers): # Лучше
    for number in numbers:
        yield number ** 2
```

Ниже приведен список префиксов и суффиксов, которых, несмотря на их широкое распространение в программировании, следует избегать в именах функций и классов:

- `Manager`;
- `Object`;
- `Do`, `handle` или `perform`.

Причина в том, что эти слова неконкретные, двусмысленные и не придают никакого значения реальному имени. Джефф Этвуд, соучредитель Discourse и Stack Overflow, написал очень хорошую статью на данную тему в своем блоге blog.codinghorror.com/ishall-call-it-somethingmanager/.

Существует также список имен пакетов, которых следует избегать. Имена, не связанные с содержимым пакета, могут навредить проекту в долгосрочной перспективе. Такие имена, как `misc`, `tools`, `utils`, `common` или `core`, приводят к появлению больших кусков кода очень низкого качества, объем которых потом экспоненциально растет. Чаще всего наличие такого модуля говорит о лени разработчика. Любители таких имен модулей с тем же успехом могли бы называть их `trash` или `dumpster`, поскольку именно так их и будут воспринимать товарищи по команде.

В большинстве случаев почти всегда лучше иметь несколько небольших модулей, пусть даже с малым количеством контента, но с именами, хорошо отражающими содержимое. Честно говоря, в таких именах, как `utils` и `common`, нет ничего плохого и их можно использовать без ущерба проекту. Но реальность показывает: часто они лишь порождают примеры того, как *не надо* делать, которые размножаются очень быстро. Лучше всего просто избегать таких рискованных организационных моделей и пресекать их в зародыше.

Избегайте уже существующих имен

Не принято использовать имена, которые дублируют уже существующие в том же контексте. Это сильно усложняет чтение и отладку кода. Всегда лучше определить заранее уже существующие имена, даже если они являются локальными по отношению к контексту. Если вам все же необходимо повторно использовать существующие имена или ключевые слова, то используйте подчеркивание в конце, чтобы избежать конфликтов, например:

```
def xapian_query(terms, or_=True):
    """если or_ истинно, элементы terms объединяются
       с помощью оператора OR"""
    ...
```

Обратите внимание: ключевое слово `class` часто заменяется на `klass` или `cls`:

```
def factory(klass, *args, **kwargs):
    return klass(*args, **kwargs)
```

Рассмотрим некоторые рекомендации по работе с аргументами.

Практические рекомендации по работе с аргументами

Сигнатуры функций и методов — это столпы, на которых зиждется целостность кода. Они определяют его использование и составляют его API. Помимо правил именования, представленных выше, следует уделить особое внимание аргументам. Это можно сделать с помощью трех простых правил, таких как:

- сборка аргументов по итеративному принципу;
- доверие аргументам и тестам;
- осторожное использование `*args` и `**kwargs`.

Сборка аргументов по итеративному принципу

Наличие постоянного и четко определенного списка аргументов для каждой функции делает ваш код более надежным. Но это не получится воплотить в первой версии, так что аргументы должны быть построены по итеративному принципу. Они должны отражать конкретные случаи использования элемента, для которого он был создан, и развиваться соответственно.

Рассмотрим следующий пример из первых версий класса `Service`:

```
class Service: # Версия 1
    def _query(self, query, type):
        print('done')

    def execute(self, query):
        self._query(query, 'EXECUTE')
```

Если вы хотите расширить сигнатуру метода `execute()` новыми аргументами так, чтобы сохранить обратную совместимость, то должны указать значения по умолчанию для этих аргументов следующим образом:

```
class Service(object): # Версия 2
    def _query(self, query, type, logger):
        logger('done')

    def execute(self, query, logger=logging.info):
        self._query(query, 'EXECUTE', logger)
```

В следующем примере из интерактивной сессии показаны два стиля вызова метода `execute()` обновленного класса `Service`:

```
>>> Service().execute('my query')   # Устаревший вызов
>>> Service().execute('my query', logging.warning)
WARNING:root:done
```

Доверие к аргументам и тестам

Учитывая природу динамического ввода Python, некоторые разработчики используют утверждение в верхней части функций и методов с целью убедиться, что в аргументах заложено правильное содержание, например:

```
def divide(dividend, divisor):
    assert isinstance(dividend, (int, float))
    assert isinstance(divisor, (int, float))
    return dividend / divisor
```

Это часто делают разработчики, которые привыкли к статической типизации и чувствуют, что в Python чего-то не хватает.

Данный способ проверки аргументов является частью стиля контрактного программирования DBC, в котором предварительные условия проверяются перед непосредственным запуском кода.

Две основные проблемы этого подхода заключаются в следующем:

- код DBC объясняет, как его следует использовать, что делает его менее читаемым;
- это может замедлить его, так как оператор условия есть в каждом вызове.

Последнего можно избежать с помощью опции `-O` интерпретатора Python. В этом случае все утверждения удаляются из кода до создания байт-кода и проверка теряется.

В любом случае утверждения надо делать осторожно и их нельзя использовать для насильного превращения Python в статически типизированный язык. Единственный случай использования такого метода — для защиты кода от бессмысленного вызова. Если вам нужна статическая типизация в Python, то обязательно попробуйте MyPy или аналогичную статическую проверку, которая не влияет на время выполнения кода и позволяет обеспечить определение типов в более удобном для восприятия виде с помощью аннотаций функций и переменных.

Осторожность при работе с магическими аргументами *args и **kwargs

Аргументы `*args` и `**kwargs` могут нарушить устойчивость функции или метода к различным ошибкам. Сигнатура становится расплывчатой, а код выполняет лишний парсинг аргументов, хотя и не должен, например:

```
def fuzzy_thing(**kwargs):
    if 'do_this' in kwargs:
        print('ok i did this')

    if 'do_that' in kwargs:
        print('that is done')

    print('ok')

>>> fuzzy_thing(do_this=1)
ok i did this
ok
>>> fuzzy_thing(do_that=1)
that is done
ok
>>> fuzzy_thing(what_about_that=1)
ok
```

Если список аргументов становится длинным и сложным, то появляется соблазн добавить магические аргументы. Однако это говорит лишь о недостатках функции или метода, который придется делить на части или реорганизовывать.

Если `*args` используется для работы с последовательностью элементов, которые обрабатываются так же, как функции, то лучше будет задать в качестве аргумента уникальный контейнер, например `iterator`:

```
def sum(*args):  # Сойдет
    total = 0
    for arg in args:
        total += arg
    return total

def sum(sequence):  # Лучше!
    total = 0
    for arg in sequence:
        total += arg
    return total
```

Для `**kwargs` применяется то же самое правило. Лучше зафиксировать именованные аргументы, чтобы сигнатура стала более осмысленной:

```
def make_sentence(**kwargs):
    noun = kwargs.get('noun', 'Bill')
    verb = kwargs.get('verb', 'is')
    adjective = kwargs.get('adjective', 'happy')
    return f'{noun} {verb} {adjective}'

def make_sentence(noun='Bill', verb='is', adjective='happy'):
    return f'{noun} {verb} {adjective}'
```

Еще один интересный подход заключается в создании контейнера класса, который группирует несколько взаимосвязанных аргументов, чтобы создать контекст

выполнения. Эта структура отличается от `*args` или `**kwargs` тем, что позволяет работать со значениями и ее части могут развиваться независимо друг от друга. Код, в котором используются такие аргументы, не будет работать с его внутренними элементами.

Например, веб-запрос, который передается в функцию, часто является экземпляром класса. Этот класс отвечает за хранение данных, передаваемых веб-сервером, как показано в следующем коде:

```
def log_request(request):  # версия 1
    print(request.get('HTTP_REFERER', 'No referer'))

def log_request(request):  # версия 2
    print(request.get('HTTP_REFERER', 'No referer'))
    print(request.get('HTTP_HOST', 'No host'))
```

Иногда без магических аргументов обойтись нельзя, особенно в метапрограммировании. Например, они незаменимы при создании декораторов, которые работают в функциях с любым видом сигнатуры.

В следующем разделе поговорим об именах классов.

Имена классов

Имя класса должно быть кратким, точным и описательным. Обычно используют суффикс, в котором заложена информация о типе или природе класса, например:

- **SQL**Engine;
- **Mime**types;
- **String**Widget;
- **Test**Case.

Для базовых или абстрактных классов можно использовать префикс **Base** или **Abstract**:

- **Base**Cookie;
- **Abstract**Formatter.

Самое главное — не путаться в атрибутах класса. К примеру, нужно избегать избыточности между именами класса и его атрибутов, как показано ниже:

```
>>> SMTP.smtp_send()   # Избыточная информация в пространстве имен
>>> SMTP.send()        # Более читабельный вариант
```

В следующем разделе поговорим об именах модулей и пакетов.

Имена модулей и пакетов

Имена модулей и пакетов должны нести информацию об их назначении и содержании. Это короткие имена в нижнем регистре и, как правило, без подчеркивания, например:

- `sqlite`;
- `postgres`;
- `sha1`.

Если модуль реализует протокол, то часто используется суффикс `lib`, как в следующем примере:

```
import smtplib
import urllib
import telnetlib
```

При выборе имени для модуля всегда нужно учитывать его содержание и устранять избыточность в пространстве имен, например:

```
from widgets.stringwidgets import TextWidget # Плохо
from widgets.strings import TextWidget       # Лучше
```

Когда модуль становится слишком сложным и обрастает классами, неплохо бы создать пакет и разделить элементы модуля на дополнительные модули.

Модуль `__init__` можно задействовать для возврата некоторых общих API на верхний уровень пакета. Такой подход позволяет разделить код на более мелкие компоненты, причем не во вред удобству использования.

Рассмотрим некоторые полезные инструменты, применяемые при работе с соглашениями по именованию и стилям.

Полезные инструменты

Общие соглашения и приемы, используемые в программном проекте, всегда должны быть задокументированы. Однако наличия документации по руководящим принципам не всегда достаточно, чтобы обеспечить их соблюдение. К счастью, можно применять автоматизированные инструменты, позволяющие проверить источники кода и то, соответствует ли он требованиям конкретных соглашений об именовании и руководящим принципам стиля.

Ниже приведено несколько популярных инструментов:

- `pylint` — очень гибкий анализатор исходного кода;
- `pycodestyle` и `flake8` — инструменты для проверки и обертки кода, которые к тому же добавляют в код некоторые полезные функции, такие как статический анализ и измерение сложности.

Pylint

Помимо некоторых показателей обеспечения качества, Pylint позволяет проверить, соответствует ли данный исходный код соглашению об именовании. Его настройки по умолчанию соответствуют PEP 8, а скрипт Pylint обеспечивает вывод отчета оболочки.

Чтобы установить Pylint, вы можете использовать `pip` следующим образом:

```
$ pip install pylint
```

После этого команда будет доступна и вы сможете работать с одним или несколькими модулями с помощью символов. Попробуем Pylint на скрипте `bootstrap.py` из Buildout, как показано ниже:

```
$ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py -q
$ pylint bootstrap.py
No config file found, using default configuration
************* Module bootstrap
C: 76, 0: Unnecessary parens after 'print' keyword (superfluous-parens)
C: 31, 0: Invalid constant name "tmpeggs" (invalid-name)
C: 33, 0: Invalid constant name "usage" (invalid-name)
C: 45, 0: Invalid constant name "parser" (invalid-name)
C: 74, 0: Invalid constant name "options" (invalid-name)
C: 74, 9: Invalid constant name "args" (invalid-name)
C: 84, 4: Import "from urllib.request import urlopen" should be placed at
the top of the module (wrong-import-position)
...

Global evaluation
-----------------
Your code has been rated at 6.12/10
```

Реальный вывод Pylint будет немного длиннее, а здесь он был усечен для краткости.

Помните, что Pylint часто выдает ложноположительные предупреждения, которые снижают общую оценку качества. Например, оператор импорта, не используемый в коде самого модуля, прекрасно будет работать в некоторых случаях (например, при сборке модулей `__init__` верхнего модуля в пакете). Вывод Pylint — это скорее подсказка, а не что-то стопроцентно верное.

Выполнение вызовов библиотек, в именах методов которых используется смешанный регистр, также может привести к снижению оценки. В любом случае глобальная оценка кода не так уж важна. Pylint — это просто инструмент, который указывает, что можно улучшить.

Всегда рекомендуется подстраивать Pylint под себя. Для этого нужно создать файл конфигурации `.pylinrc` в корневом каталоге вашего проекта. Вы можете сделать это с помощью опции `-generate-rcfile` команды `pylint`:

```
$ pylint --generate-rcfile > .pylintrc
```

Этот файл конфигурации самодокументируется (то есть все опции описаны в комментариях) и уже должен содержать все доступные опции конфигурации Pylint.

Помимо проверки на соответствие некоторым обязательным стандартам кодирования, Pylint также может дать дополнительную информацию об общем качестве кода, например:

- метрики дублирования кода;
- неиспользованные переменные и импорт;
- отсутствующие строки документации в функциях, методах или классах;
- слишком длинные сигнатуры функций.

Список проверок, доступных по умолчанию, довольно велик. Важно понимать, что часть этих правил весьма условна и их не всегда легко применить к каждой кодовой базе. Помните, что последовательность всегда ценнее, чем соблюдение некоторых произвольных правил. К счастью, Pylint довольно гибок, так что если в вашей команде используются некие соглашения по именованию и кодированию, которые отличаются от общепринятых, то вы легко можете настроить Pylint для проверки согласованности именно с вашими соглашениями.

pycodestyle и flake8

Инструмент `pycodestyle` (ранее назывался `pep8`) был создан для выполнения проверки стиля по соглашениям, определенным в PEP 8. Это его основное отличие от Pylint, у которого есть еще много других возможностей. Это лучший вариант для программистов, желающих иметь автоматизированный инструмент проверки стиля кода только для стандарта PEP 8, без каких-либо дополнительных настроек, как в случае с Pylint.

Устанавливается `pycodestyle` через `pip` следующим образом:

```
$ pip install pycodestyle
```

При запуске скрипта `bootstrap.py` из Buildout вы получите следующий краткий перечень нарушений стиля:

```
$ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py -q
$ pycodestyle bootstrap.py
bootstrap.py:118:1: E402 module level import not at top of file
bootstrap.py:119:1: E402 module level import not at top of file
bootstrap.py:190:1: E402 module level import not at top of file
bootstrap.py:200:1: E402 module level import not at top of file
```

Основное отличие этого вывода от Pylint заключается в его длине. Инструмент `pycodestyle` сконцентрирован только на стиле и поэтому не выдает других предупреждений, таких как неиспользуемые переменные, слишком длинные имена

функций или отсутствующие строки документации. Кроме того, он не дает коду оценку. В этом есть смысл, поскольку нет такого понятия, как «код частично написан по правилам». Любое, даже малейшее, нарушение руководящих принципов стиля делает код несоответствующим PEP 8.

Код `pycodestyle` проще, чем у Pylint, и его вывод легче анализировать, так что он может подойти, если вы хотите включить проверку стиля кода в непрерывный процесс интеграции. В случае нехватки каких-нибудь функций статического анализа есть пакет `flake8`, который является оболочкой `pycodestyle`, а также нескольких других легко расширяемых инструментов, имеющих более широкий набор функций. К ним относятся следующие:

- измерение сложности Мак-Кейба;
- статический анализ с помощью `pyflakes`;
- отключение целых файлов или отдельных строк с помощью комментариев.

Резюме

В этой главе мы рассмотрели наиболее распространенные и широко принятые соглашения о написании кода. Мы начали с официального руководства по стилю Python (PEP 8). Затем мы дополнили его кое-какими предложениями по именованию, которые сделают ваш будущий код более явным. Мы также рассмотрели ряд полезных инструментов, необходимых для поддержания согласованности и качества кода.

Теперь вы готовы перейти к первой практической теме книги: написанию и распространению собственных пакетов. В следующей главе вы узнаете, как опубликовать собственный пакет в репозитории PyPI, а также как использовать возможности экосистем упаковки в вашей частной организации.

7 Создаем пакеты

Эта глава посвящена процессу написания и выпуска пакетов на Python. Мы узнаем, как побыстрее установить все, что нужно, прежде чем начать реальную работу. Мы также увидим, как стандартизировать методику написания пакетов и упростить разработку через тестирование. Наконец, мы поговорим о том, как облегчить процесс выпуска.

Глава разделена на следующие четыре части.

- *Общая схема* для всех пакетов, описывающая сходство между всеми пакетами Python, а также то, какую роль играют дистрибутивы и инструменты установки в процессе упаковки.
- Что такое *пакеты пространства имен* и чем они полезны.
- Как зарегистрировать и загрузить пакеты в *каталог пакетов Python* (Python Package Index, PyPI), правила безопасности и распространенные ловушки.
- *Исполняемые файлы* как альтернативный способ упаковки и распространения приложений, написанных на Python.

В этой главе:

- создание пакета;
- пакеты пространства имен;
- загрузка пакета;
- исполняемые файлы.

Технические требования

Ниже перечислены упомянутые в этой главе пакеты, которые можно скачать с PyPI:

- `twine`;
- `wheel`;
- `cx_Freeze`;

- `py2exe`;
- `pyinstaller`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по ссылке github.com/packtpublishing/expert-python-programming-third-edition/tree/master/chapter7.

Создание пакета

Сам процесс упаковки в Python может поначалу показаться странным. В основном это связано с путаницей в инструментах, подходящих для создания пакетов Python. Однако, создав свой первый пакет, вы увидите, что все не так сложно, как выглядит. Кроме того, вам поможет знание передовых инструментов.

Вам стоит уметь создавать пакеты, даже если вы не собираетесь распространять код. Умение создавать собственные пакеты позволит лучше понимать всю эту систему и, следовательно, разбираться со сторонним кодом, который есть на PyPI и который вы, вероятно, уже используете.

Кроме того, наличие проекта с закрытым исходным кодом или его компонентов в виде пакетов может помочь в развертывании вашего кода в различных средах. Преимущества использования packaging-экосистемы Python в процессе развертывания кода будут более подробно описаны в следующей главе. Здесь мы сосредоточимся на правильных инструментах и методах создания таких дистрибутивов.

В следующем разделе мы обсудим, что такого странного в нынешних инструментах создания пакетов в Python.

Странности в нынешних инструментах создания пакетов в Python

Создание пакетов Python переживало долгие странные времена, и потребовалось много лет, чтобы навести в этой теме порядок. Все началось с пакета `distutils`, введенного в 1998 году, который впоследствии был дополнен `setuptools` в 2003-м. Эти два проекта породили великое множество ответвлений, альтернативных проектов и полных переписываний, целью которых было раз и навсегда привести в порядок экосистему упаковки в Python. К сожалению, большинство из этих попыток оказались неудачными. Эффект был совершенно противоположный. Каждый новый проект, пытавшийся заменить собой `setuptools` или `distutils`, только вносил еще больше путаницы. Одни проекты выродились и вернулись к корням

(например, `distribute` — ответвление `setuptools`), а другие остались заброшенными (например, `distutils2`).

К счастью, дела постепенно налаживаются. Организация под названием *Python Packaging Authority (PyPA)* была создана с целью вернуть порядок и организацию в экосистему упаковки. Руководство пользователя по упаковке Python (packaging.python.org), выпускаемое PyPA, является авторитетным источником информации о новейших средствах упаковки и практических рекомендациях. В рамках данной главы этот сайт можно считать лучшим источником информации об упаковке. Руководство также содержит подробную историю изменений и информацию о новых проектах, связанных с упаковкой. Даже если вы уже знаете кое-что об упаковке, его стоит прочесть с целью убедиться, что вы используете правильные инструменты.

Держитесь подальше от других популярных интернет-ресурсов, таких как *Hitchhiker's Guide to Packaging*. Это старый, неподдерживаемый и неактуальный ресурс. Он может быть интересен только любителям истории, а PyPA, по сути, ответвление данного ресурса.

Посмотрим, как PyPA влияет на упаковку Python.

Нынешняя ситуация с созданием пакетов Python благодаря PyPA

Помимо авторитетного руководства по упаковке, PyPA также поддерживает другие подобные проекты и процесс стандартизации новых официальных аспектов упаковки Python. Все проекты PyPA можно найти в одной организации на GitHub: github.com/pypa.

Некоторые из проектов уже были упомянуты в книге. Ниже приведены наиболее видные из них:

- `pip`;
- `virtualenv`;
- `twine`;
- `warehouse`.

Обратите внимание: большинство из этих проектов были запущены еще до PyPA и вошли в него, уже будучи зрелыми и распространенными решениями.

Благодаря PyPA произошел прогрессивный отказ от всякой ерунды в пользу хороших решений. Кроме того, благодаря приверженности сообщества PyPA старая реализация PyPI была наконец полностью переписана в виде проекта *Warehouse*. Теперь у PyPI появился модернизированный пользовательский интерфейс и произошло долгожданное улучшение юзабилити и возможностей.

В следующем подразделе рассмотрим некоторые из инструментов, рекомендуемых для работы с пакетами.

Рекомендации по инструментам

Руководство по упаковке Python содержит несколько рекомендаций по инструментам для работы с пакетами. В целом их можно разделить на следующие две группы:

- инструменты для установки пакетов;
- инструменты для создания и распространения пакетов.

Инструменты из первой группы, рекомендованные PyPA, уже упоминались нами в главе 2, однако вспомним их еще раз в порядке логики повествования:

- использование `pip` для установки пакетов из PyPI;
- использование `virtualenv` или `venv` для изоляции среды выполнения Python на уровне приложения.

Рекомендации руководства по упаковке Python, касающиеся инструментов для создания и распространения пакетов, заключаются в следующем:

- используйте `setuptools` для определения проектов и *распространения исходного кода*;
- используйте подходящие средства для *распространения сборок*;
- используйте `twine` для загрузки пакета на PyPI и его последующего распространения.

Рассмотрим, как настроить ваш проект.

Конфигурация проекта

Очевидно, что самый простой способ организовать код в большом приложении — разделить его на несколько пакетов. Это упрощает понимание кода, его сопровождение и редактирование, а также максимизирует повторное использование кода. Отдельные пакеты выступают в качестве компонентов, которые можно задействовать в различных программах.

setup.py

В корневом каталоге распространяемого пакета есть скрипт `setup.py`. Там определены все метаданные, как описано в модуле `distutils`. Метаданные пакета выражаются в виде аргументов при вызове стандартной функции `setup()`. Несмотря на то что `distutils` — это модуль стандартной библиотеки, предназначенный для упаковки, все же вместо него рекомендуется использовать `setuptools`. В данный пакет внесены несколько усовершенствований по сравнению со стандартным модулем `distutils`.

Минимальное содержание этого файла выглядит следующим образом:

```
from setuptools import setup

setup(
    name='mypackage',
)
```

Элемент `name` — это полное имя пакета. Кроме того, скрипт предоставляет несколько команд, которые можно вывести с помощью опции `--help-commands`, как показано в следующем коде:

```
$ 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
```

На самом деле список команд больше и может варьироваться в зависимости от имеющихся расширений `setuptools`. Мы показали только самые важные и актуальные для данной главы. *Стандартные команды* — это встроенные команды, из `distutils`, а *дополнительные* пришли из сторонних пакетов, таких как `setuptools`, или из любого другого пакета, который определяет и регистрирует новую команду. Здесь примером такой дополнительной команды является `bdist_wheel` из пакета `wheel`.

setup.cfg

Файл `setup.cfg` содержит параметры по умолчанию для команд скрипта `setup.py`. Он чрезвычайно полезен, когда процесс создания и распространения пакета усложняется и требует множества необязательных аргументов, которые передаются командам скрипта `setup.py`. В файле `setup.cfg` можно хранить такие параметры по

умолчанию вместе с исходным кодом для каждого проекта. Это позволит сделать процедуру распространения независимой от проекта, а также обеспечит прозрачность процесса сборки и распространения пакета среди пользователей и других членов команды.

Синтаксис файла `setup.cfg` аналогичен встроенному модулю `configparser`, поэтому он похож на популярные файлы Microsoft Windows INI. Ниже приведен пример файла конфигурации `setup.cfg`, в котором определены параметры по умолчанию `global`, `sdlist` и `bdist_wheel`:

```
[global]
quiet=1

[sdist]
formats=zip,tar

[bdist_wheel]
universal=1
```

Пример конфигурации гарантирует, что исходный код (раздел `sdist`) всегда будет создаваться в двух форматах (ZIP и TAR), а встроенные сборки `wheel` (`bdist_wheel`) создаются как универсальные диски, не зависящие от версии Python. Кроме того, большая часть выходных данных каждой команды будет подавляться глобальным переключателем `--quiet`. Обратите внимание: эта опция включена только в целях демонстрации и подавлять вывод для каждой команды по умолчанию с помощью данной опции — плохое решение.

MANIFEST.in

При создании сборки с помощью команды `sdist` модуль `distutils` просматривает каталог пакета и ищет файлы, которые следует добавить в архив. По умолчанию `distutils` содержит:

- все исходные файлы Python, подключенные в аргументах `py_modules`, `packages` и `scripts`;
- все исходные файлы C, перечисленные в аргументе `ext_modules`;
- файлы, соответствующие стандартной маске `test/test*.py`;
- файлы с именами `README`, `README.txt`, `setup.py` и `setup.cfg`.

Кроме того, если ваш пакет управляется системой контроля версий, например Subversion, Mercurial или Git, то вы можете автоматически включать все версии контролируемых файлов с помощью дополнительных расширений `setuptools`, таких как `setuptools-svn`, `setuptools-hg`, и `setuptools-git`. С помощью других расширений возможна интеграция с другими системами управления версиями. Независимо от того, встроенная это стратегия сбора по умолчанию или опреде-

ляется пользовательским расширением, `sdist` создаст файл `MANIFEST`, в котором перечислены все файлы, и включит их в финальный архив.

Предположим, что вы не используете никаких дополнительных расширений и нужно включить в дистрибутив пакета файлы, которые не были учтены по умолчанию. Вы можете определить шаблон `MANIFEST.in` в корневом каталоге вашего пакета (тот же каталог, что и у файла `setup.py`). Этот шаблон направляет `sdist` команду о том, какие файлы включить.

Шаблон `MANIFEST.in` определяет одно включение или исключение правила в каждой строке:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.txt *.py
```

Полный список команд `MANIFEST.in` можно найти в официальной документации `distutils`.

Наиболее важные метаданные

Помимо имени и версии сборки пакета, можно выделить другие наиболее важные аргументы, которые принимает функция `setup()`:

- `description`: — несколько предложений, описывающих пакет;
- `long_description` — включает в себя полное описание в виде `reStructuredText` (по умолчанию) или на других поддерживаемых языках разметки;
- `long_description_content_type` — определяет тип MIME длинного описания; служит для указания репозитория пакетов, какой язык разметки используется для описания пакета;
- `keywords` — список ключевых слов, которые определяют пакет и дают лучшую индексацию в репозитории пакетов;
- `author` — имя автора пакета или выпустившей его организации;
- `author_email` — адрес электронной почты автора;
- `url` — URL проекта;
- `license` — имя лицензии (GPL, LGPL и т. д.), под которой распространяется пакет;
- `packages` — список всех имен пакетов в сборке; в `setuptools` есть небольшая функция под названием `find_packages`, позволяющая автоматически находить имена пакетов, которые надо включить;
- `namespace_packages` — список пакетов пространства имен в пределах сборки.

Классификаторы коллекций

PyPI и `distutils` — это решения для категоризации приложений с множеством классификаторов под *классификаторы коллекций*. Все классификаторы коллекций образуют древовидную структуру. Каждая строка классификатора определяет список вложенных пространств имен, в котором каждое из них разделено подстрокой `::`. Их перечень приведен в определении пакета в аргументе `classifiers` функции `setup()`.

Ниже приведен пример списка классификаторов, взятых из проекта `solrq`, доступного на 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',
    ],
)
```

Применять такие классификаторы в определении пакета совершенно не обязательно, но это бывает полезно в дополнение к основным метаданным, доступным в интерфейсе `setup()`. Среди прочего классификаторы коллекций предоставляют информацию о поддерживаемых версиях Python, поддерживаемых операционных системах, стадиях разработки проекта или лицензиях кода. Многие пользователи PyPI ищут доступные пакеты по категориям, и правильная классификация в этом случае помогает пакетам достичь их цели.

Классификаторы коллекций играют важную роль во всей экосистеме создания пакетов, и их никогда не следует игнорировать. Нет ни одной организации, которая проверяла бы классификацию пакетов, так что именно вы должны задать классификаторы для ваших пакетов, чтобы не порождать в репозитории хаос.

На момент написания этой книги на PyPI есть 667 классификаторов, разбитых на следующие девять основных категорий:

- состояние разработки;
- среда;
- фреймворк;
- целевая аудитория;
- лицензия;
- естественный язык;
- операционная система;
- язык программирования;
- тема.

Данный список постоянно растет, и время от времени добавляются новые классификаторы. Поэтому их количество может оказаться другим в момент, когда вы читаете эти строки. Полный список доступных в настоящее время классификаторов коллекций можно посмотреть по ссылке pypi.org/classifiers.

Общие шаблоны

Создание сборки пакета — довольно утомительная задача для неопытных разработчиков. Большую часть метаданных, которые подаются `setuptools` или `distuitls` в функциях `setup()`, можно настроить вручную, игнорируя тот факт, что эти метаданные могут быть доступны и в других частях проекта. Ниже показан пример:

```
from setuptools import setup

setup(
    name="myproject",
    version="0.0.1",
    description="mypackage project short description",
    long_description="""
        Longer description of mypackage project
        possibly with some documentation and/or
        usage examples
    """,
    install_requires=[
        'dependency1',
        'dependency2',
        'etc',
    ]
)
```

Некоторые элементы метаданных время от времени встречаются в разных местах в типичном проекте Python. Например, содержание длинного описания обычно включается в файл `README`, а спецификатор версии обычно находится в модуле `__init__` пакета. Жесткая привязка таких метаданных к функции `setup()` будет избыточной и приведет к лишним ошибкам и несоответствиям в дальнейшем. Модули `setuptools` и `distutils` не могут автоматически получать информацию о метаданных из кода проекта, так что это нужно делать самостоятельно. Сообществом Python приняты некоторые общие шаблоны решения наиболее популярных проблем, таких как управление зависимостями, включение версий/`README` и т. д. Следует знать по крайней мере некоторые из них: они настолько популярны, что их можно брать за эталон.

Автоматическое включение строки версии из пакета. В документе PEP 440 определен стандарт спецификации версии и зависимостей. Это длинный документ, в котором описывается схема спецификации версий и определяется, как в Python должно работать сопоставление и сравнение версий в инструментах упаковки. Если вы используете или планируете применять сложную систему нумерации версий проекта, то вам следует внимательно изучить этот документ. В случае использования простой схемы из одного, двух, трех или более чисел, разделенных точками, копаться в PEP 440 вам не придется. Если вы не знаете, как выбрать правильную схему управления версиями, то я настоятельно рекомендую применить семантическую схему, которую мы уже кратко упоминали в главе 1.

Другая проблема, связанная с управлением версиями кода, заключается в том, где именно надо подключать спецификатор версии внутри пакета или модуля. Документ PEP 396 решает именно эту проблему. Данный документ является информационным и имеет отложенный статус и поэтому не входит в перечень официальных стандартов Python. Однако в нем описано то, что де-факто в настоящее время считается стандартом. Согласно PEP 396, если у пакета или модуля есть конкретная определенная версия, то спецификатор версии включается как атрибут `__version__` в файл `__init__.py` или в файл сборки модуля. Еще один общепринятый стандарт — это включать атрибут `VERSION`, содержащий кортеж частей спецификатора версии. Такой способ помогает пользователям писать совместимый код, поскольку кортежи версий удобно сравнивать, если схема управления версиями достаточно проста.

Множество пакетов, выложенных на PyPI, соответствуют обоим соглашениям. В их файлах `__init__.py` указаны атрибуты версий, которые выглядят следующим образом:

```
# Версия в виде кортежа для простоты сравнения
VERSION = (0, 1, 1)
# Строка, созданная из кортежа во избежание ошибок
__version__ = ".".join([str(x) for x in VERSION])
```

Другое предложение PEP 396 состоит в том, чтобы аргумент версии в функции `setup()` из скрипта `setup.py` извлекался из `__version__` или наоборот. В руководстве по упаковке Python есть несколько шаблонов для генерации версии проекта из одного источника, и каждый из них имеет свои преимущества и недостатки. Мой любимый шаблон довольно длинный и не входит в руководство PyPA, но его сложность ограничена лишь скриптом `setup.py`. Этот шаблон предполагает, что спецификатор версии указан в атрибуте `VERSION` модуля `__init__` и извлекает эти данные для включения в вызов `setup()`. Ниже приведен отрывок из скрипта `setup.py` воображаемого пакета, иллюстрирующий этот подход:

```
from setuptools import setup
import os

def get_version(version_tuple):
    # Дополнительная обработка тегов, которая может быть
    # проще в зависимости от схемы версий
    if not isinstance(version_tuple[-1], int):
        return '.'.join(
            map(str, version_tuple[:-1])
        ) + version_tuple[-1]
    return '.'.join(map(str, version_tuple))

# Путь к модулю __init__ в проекте
init = os.path.join(
    os.path.dirname(__file__), 'src', 'some_package',
    '__init__.py'
)

version_line = list(
    filter(lambda l: l.startswith('VERSION'), open(init))
)[0]

# VERSION — это кортеж, поэтому нужно оценить version_line.
# Мы могли бы просто импортировать его из пакета, но не факт,
# что пакет импортируется до того, как установка будет завершена.
PKG_VERSION = get_version(eval(version_line.split('=')[-1]))

setup(
    name='some-package',
    version=PKG_VERSION,
    # ...
)
```

Файл README. Каталог пакетов Python позволяет отображать файл `README` проекта или значение переменной `long_description` на странице пакета на портале PyPI. Портал может интерпретировать разметку, используемую в содержании `long_description`, и выводит ее в виде HTML на странице пакета. Тип языка

разметки задается аргументом `long_description_content_type` функции `setup()`. На данный момент доступны следующие три варианта разметки:

- обычный текст — `long_description_content_type='text/plain'`;
- ReStructuredText — `long_description_content_type='text/x-rst'`;
- Markdown — `long_description_content_type='text/markdown'`.

Markdown и ReStructuredText — наиболее популярный выбор разработчиков на Python, но кто-то по тем или иным причинам по-прежнему может использовать другие языки разметки. Если вы хотите задействовать какой-то другой язык разметки для `README` вашего проекта, то можете указать его в качестве описания проекта на странице PyPI в читабельном виде. Вся соль здесь в применении пакета `pypandoc`, который позволяет превратить другой язык разметки в ReStructuredText (или Markdown) при загрузке пакета в каталог пакетов Python. Нужно также предусмотреть запасной вариант для простого отображения файла `README`, поэтому установка не завершится неудачей, если у пользователя не установлен `pypandoc`. Ниже приведен код скрипта `setup.py`, который считывает содержимое файла `README`, записанного на языке разметки AsciiDoc и переводит его в ReStructuredText перед включением аргумента `long_description`.

```
from setuptools import setup
try:
    from pypandoc import convert

    def read_md(file_path):
        return convert(file_path, to='rst', format='asciidoc')

except ImportError:
    convert = None
    print(
        "warning: pypandoc module not found, "
        "could not convert Asciidoc to RST"
    )

    def read_md(file_path):
        with open(file_path, 'r') as f:
            return f.read()

README = os.path.join(os.path.dirname(__file__), 'README')

setup(
    name='some-package',
    long_description=read_md(README),
    long_description_content_type='text/x-rst',
    # ...
)
```

Управление зависимостями. Многим проектам для нормальной работы требуется установка внешних пакетов. Когда список зависимостей становится длинным,

встает вопрос управления им. Ответ в большинстве случаев прост: не надо перебарщивать со сложностью. Просто явно укажите список зависимостей в скрипте `setup.py` следующим образом:

```
from setuptools import setup
setup(
    name='some-package',
    install_requires=['falcon', 'requests', 'delorean']
    # ...
)
```

Некоторые разработчики на Python любят использовать файлы `requirements.txt` для отслеживания списков зависимостей своих пакетов. Иногда это обоснованно, но в большинстве случаев является пережитком времен, когда код проекта нельзя было нормально упаковать. Во всяком случае, даже такие известные проекты, как Celery, по-прежнему придерживаются этого стиля. Поэтому, если вы не готовы изменить свои привычки или по какой-то причине вынуждены использовать такой файл, то по крайней мере делайте это правильно. Вот одна из популярных идиом для чтения списка зависимостей из файла `requirements.txt`:

```
from setuptools import setup
import os

def strip_comments(l):
    return l.split('#', 1)[0].strip()
def reqs(*f):
    return list(filter(None, [strip_comments(l) for l in open(
        os.path.join(os.getcwd(), *f)).readlines()]))

setup(
    name='some-package',
    install_requires=reqs('requirements.txt')
    # ...
)
```

В следующем подразделе вы узнаете, как добавлять пользовательские команды в скрипт настройки.

Пользовательская команда setup

Модуль `distutils` позволяет создавать новые команды. Новая команда регистрируется с точкой входа, указанной в `setuptools`, что дает простой способ определения пакетов в виде плагинов.

Точка входа является именованной ссылкой на класс или функцию, которая доступна через несколько API в `setuptools`. Любое приложение может просмотреть все зарегистрированные пакеты и использовать связанный код в виде плагина.

Чтобы привязать новую команду, можно применить метаданные `entry_points` в вызове `setup` следующим образом:

```
setup(
    name="my.command",
    entry_points="""
         [distutils.commands]
        my_command = my.command.module.Class
    """
)
```

Все именованные ссылки собираются в именованных разделах. После загрузки модуль `distutils` сканирует ссылки, которые были зарегистрированы в `distutils.commands`.

Этот механизм используется многими приложениями Python, которые обеспечивают расширяемость.

Посмотрим, как работать с пакетами на стадии разработки.

Работа с пакетами в процессе разработки

Работа с `setuptools` в основном связана с созданием и распространением пакетов. Тем не менее `setuptools` также нужно использовать для установки пакетов непосредственно из источника, и причина этому проста. Желательно проверять, правильно ли работает код упаковки, прежде чем отправлять пакет в PyPI. Самый простой способ проверить пакет — это установить его. Если вы отправляете в репозиторий поврежденный пакет, то для его повторной загрузки вам следует увеличить номер версии.

Тестирование на предмет правильной упаковки перед окончательной сборкой спасает вас от ненужных увеличений номера версии и, очевидно, пустой траты времени. Кроме того, установка непосредственно из исходного кода с помощью `setuptools` может иметь значение при одновременной работе с несколькими смежными пакетами.

Установка setup.py

Команда `install` устанавливает пакет в текущей среде Python. Она будет пытаться собрать пакет, если более ранняя сборка не выполнена, а затем вводит результат в каталог файловой системы, где Python ищет установленные пакеты. При наличии архива со сборкой какого-либо пакета вы можете распаковать его во временную папку, а затем установить его с помощью этой команды. Она также установит зависимости, которые определены в аргументе `install_requires`. Зависимости будут установлены из каталога пакетов Python.

Вместо голого скрипта `setup.py` для установки пакета можно использовать `pip`. Так как этот инструмент рекомендован PyPA, его стоит применять даже при

установке пакета в локальной среде в процессе разработки. Чтобы установить пакет из локального источника, выполните следующую команду:

```
pip install <путь-к-проекту>
```

Удаление пакета

Удивительно, но в `setuptools` и `distutils` нет команды `uninstall`. К счастью, любой пакет Python можно удалить с помощью `pip`:

```
pip uninstall <имя-пакета>
```

Удаление — опасная операция для общесистемных пакетов. Это еще одна причина, почему для разработки важно использовать виртуальное окружение.

setup.py или pip -e

Пакеты, установленные командой `setup.py install`, копируются в каталог `site-packages` текущей среды Python. Это значит, что всякий раз, когда вы вносите изменения в исходный код пакета, его придется переустановить. Об этом часто забывают, вследствие чего возникают проблемы. Поэтому в `setuptools` есть дополнительная команда `develop`, которая позволяет устанавливать пакеты в *режиме разработки*. Эта команда создает специальную ссылку, которая проецирует исходный код в каталоге развертывания (`site-packages`) вместо копирования туда пакета. Исходный код пакетов можно редактировать, не прибегая к необходимости переустановки, и они доступны в `sys.path`, как если бы были установлены в обычном режиме.

`pip` тоже позволяет устанавливать пакеты в таком режиме. Этот вариант установки называется *редактируемым режимом* и включается параметром `-e` в команде `install` следующим образом:

```
pip install -e <путь-к-проекту>
```

После установки пакета в среду в режиме редактирования вы можете спокойно редактировать установленный пакет на месте, все изменения будут видны сразу, и необходимость повторной установки пакета не будет возникать.

В следующем разделе рассмотрим пакеты пространства имен.

Пакеты пространства имен

В *«Дзене Пайтона»*, который можно прочитать после `import this` в сессии интерпретатора, сказано следующее: *«Пространства имен — отличная идея, давайте использовать их почаще!»*

Это можно интерпретировать по меньшей мере двумя способами. Первый: пространство имен в контексте языка. Сами того не зная, мы используем следующие пространства имен:

- глобальное пространство имен модуля;
- локальное пространство имен функции или вызова метода;
- пространство имен класса.

Еще один вид пространства имен существует на уровне упаковки. Это *пакеты пространства имен*. Эту функцию упаковки Python часто упускают из виду, однако она весьма полезна для структурирования экосистемы пакетов в пределах организации или в очень крупном проекте.

Почему это полезно

Пространство имен можно расценивать как способ группировки связанных пакетов, где каждый из этих пакетов устанавливается независимо друг от друга.

Пакеты пространства имен особенно полезны, если у вас есть обособленно разработанные, упакованные и пронумерованные компоненты, но вам нужно иметь доступ к ним из одного пространства имен. Это также помогает четко определить, к какой организации или проекту относится каждый пакет. Например, для вымышленной компании Acme общее пространство имен может называться `acme`. В такой организации можно было бы создать общий пакет пространства имен `acme`, который выступает в качестве контейнера для других пакетов из данной организации. Например, если кто-то из Acme захочет внести изменения в пространство имен, например библиотеку SQL-запросов, то может создать новый пакет `acme.sql`, который регистрирует себя в пространстве имен `acme`.

Важно понимать, чем отличаются обычные пакеты и пакеты пространства имен и какие задачи они решают. В нормальной ситуации (без пакетов пространства имен) вы можете создать пакет под названием `acme` с подпакетом/подмодулем `sql` со следующей структурой файла:

```
$ tree acme/
acme/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 3 files
```

Всякий раз, когда вы захотите добавить новый подпакет, его нужно будет включить в исходное дерево `acme` следующим образом:

```
$ tree acme/
acme/
├──  acme
│   ├──  __init__.py
│   ├──  sql
│   │   └── __init__.py
│   └──  templating
│       └──  __init__.py
└──  setup.py

3 directories, 4 files
```

Такой подход делает независимую разработку `acme.sql` и `acme.templating` практически невозможной. В скрипте `setup.py` нужно указать все зависимости для каждого подпакета. В связи с этим будет невозможно (или по крайней мере очень сложно) настроить опциональную установку отдельных компонентов `acme`. Кроме того, при наличии большого количества подпакетов практически нереально избежать конфликтов зависимостей.

Пакеты пространства имен позволяют хранить независимое исходное дерево для каждого из этих подпакетов:

```
$ tree acme.sql/
acme.sql/
├──  acme
│   └──  sql
│       └──  __init__.py
└──  setup.py

2 directories, 2 files

$ tree acme.templating/
acme.templating/
├──  acme
│   └──  templating
│       └──  __init__.py
└── setup.py

2 directories, 2 files
```

Их можно также зарегистрировать в PyPI или другом каталоге пакетов независимо друг от друга. Пользователи могут выбрать, какие подпакеты хотят установить из пространства имен `acme`, однако никогда не устанавливают весь пакет `acme` (его может даже не существовать):

```
$ pip install acme.sql acme.templating
```

Обратите внимание: независимости деревьев исходного кода недостаточно, чтобы создавать пакеты пространства имен в Python. Потребуется принять дополнительные меры, чтобы ваши пакеты не перезаписывали друг друга. Кроме того, эти самые меры могут отличаться в зависимости от версии языка Python, с которой вы работаете. Подробнее об этом поговорим в следующих двух пунктах.

PEP 420 — неявные пакеты пространства имен

Если вы планируете работать только с Python 3, то у меня для вас хорошая новость. В документе PEP 420 задан новый способ определения пространства имен пакетов. Этот документ является частью стандартов и стал официальной частью языка, начиная с версии 3.3. Если коротко, то каждый каталог, в котором содержатся пакеты или модули Python (включая пакеты пространства имен), считается пакетом пространства имен при условии, что в нем нет файла `__init__`. Ниже приведены примеры файловых структур, представленных в предыдущем подразделе:

```
$ tree acme.sql/
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 2 files

$ tree acme.templating/
acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 2 files
```

Такой структуры достаточно, чтобы определить пакет пространства имен `acme` в Python 3.3 и более поздней версии. Минимальный файл `setup.py` для пакета `acme.templating` будет выглядеть следующим образом:

```
from setuptools import setup
setup(
    name='acme.templating',
    packages=['acme.templating'],
)
```

К сожалению, функция `setuptools.find_packages()` на момент написания данной книги не поддерживает PEP 420. Но в будущем это может измениться. Кроме того, необходимость явно определять список пакетов — не такая большая цена за легкую интеграцию пакетов пространства имен.

Пакеты пространства имен в предыдущих версиях Python

Вы не можете задействовать неявные пакеты пространства имен (по PEP 420) в версиях Python старше 3.3. Однако концепция пакетов пространства имен очень стара и давно и широко применяется в таких зрелых проектах, как Zope. Это значит, что вы можете использовать пакеты пространства имен в старых версиях Python.

Существует несколько способов, позволяющих определить, что пакет должен рассматриваться как пространство имен.

Самый простой — создать файловую структуру для каждого компонента, напоминающую обычный макет пакета без неявных пакетов пространства имен. Вся работа выполняется в `setuptools`. Так, например, макет для `acme.sql` и `acme.templating` может выглядеть следующим образом:

```
$ tree acme.sql/
acme.sql/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 3 files

$ tree acme.templating/
acme.templating/
├── acme
│   ├── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 3 files
```

Обратите внимание: и у `acme.sql`, и у `acme.templating` есть дополнительный файл исходного кода `acme/__init__.py`. Этот файл должен быть пустым. Пакет пространства имен `acme` будет создан, если мы передадим его имя в качестве значения именованного аргумента `namespace_packages` функции `setuptools.setup()` следующим образом:

```
from setuptools import setup

setup(
    name='acme.templating',
    packages=['acme.templating'],
    namespace_packages=['acme'],
)
```

Но проще не значит лучше. Для регистрации нового пространства имен модуль `setuptools` будет вызывать функцию `pkg_resources.declare_namespace()` в вашем файле `__init__.py`. Это сработает, даже если данный файл пуст. Во всяком случае, в официальной документации говорится, что именно вы должны объявить пространство имен в вашем файле `__init__.py` и это неявное поведение `setuptools` в будущем может быть отброшено. Чтобы избавиться от проблем в дальнейшем, вам нужно добавить следующую строку в файл `__init__.py`:

```
__import__('pkg_resources').declare_namespace(__name__)
```

Эта строка позволит обезопасить ваш пакет пространства имен от возможных будущих изменений в пакете пространства имен в модуле `setuptools`.

В следующем разделе рассмотрим, как загрузить пакет.

Загрузка пакета

Пакеты будут бесполезны, если их нельзя хранить, загружать и скачивать. PyPI — основное хранилище пакетов с открытым исходным кодом в сообществе Python. Любой пользователь может свободно загружать новые пакеты, и для этого достаточно всего лишь зарегистрироваться на сайте PyPI: pypi.python.org/pypi.

Разумеется, вы можете задействовать и другие хранилища, репозитории и инструменты. Это особенно полезно для распространения пакетов с закрытым исходным кодом внутри организации или для целей развертывания. Подробнее о таком использовании пакета рассказывается в следующей главе, а также приводятся инструкции по созданию собственного индекса пакетов. Здесь мы сосредоточимся на загрузке пакетов с открытым исходным кодом PyPI и скажем пару слов о том, как указать альтернативные репозитории.

PyPI — каталог пакетов Python

Как уже упоминалось, PyPI — официальный репозиторий пакетов с открытым исходным кодом. Для скачивания не требуется никакая учетная запись или разрешение. Единственное, что вам нужно, — это менеджер пакетов, с помощью которого можно скачивать новые дистрибутивы из PyPI. Предпочтительный вариант менеджера — `pip`.

В следующем разделе посмотрим, как загрузить пакет.

Скачивание пакета из PyPI или другого индекса

Любой желающий может зарегистрироваться и загрузить пакеты PyPI при условии наличия учетной записи на сайте. Пакеты привязываются к пользователю, поэтому только зарегистрировавший пакет пользователь по умолчанию является его администратором и может загружать новые дистрибутивы. Это может стать проблемой в более крупных проектах, поэтому предусмотрена возможность давать право загружать новые дистрибутивы и другим пользователям.

Самый простой способ загрузить пакет — это использовать команду `upload` из скрипта `setup.py`:

```
$ python setup.py <команды-дистрибутива> upload
```

Здесь *<команды-дистрибутива>* — это список команд, который создает дистрибутив для загрузки. В хранилище будут загружены только сборки, созданные в одном

и том же выполнении `setup.py`. Таким образом, если вы одновременно загружаете исходный код, сборку и пакет `wheel`, то вам необходимо выполнить следующую команду:

```
$ python setup.py sdist bdist bdist_wheel upload
```

При загрузке с использованием `setup.py` вы не можете повторно использовать дистрибутивы, уже собранные в предыдущих вызовах команды, поэтому вам нужно заново выполнять сборку при каждой загрузке. Это может быть неудобно для больших и сложных проектов, в которых на создание сборки может уйти довольно много времени. Еще одна проблема применения `setup.py` заключается в том, что такой способ в некоторых версиях Python позволяет использовать простой текстовый HTTP или непроверенные соединения HTTPS. Поэтому в качестве безопасной замены для команды `setup.py` рекомендуют задействовать Twine.

Twine — это утилита для взаимодействия с PyPI, предназначенная для одной цели — безопасной загрузки пакетов в репозиторий. Утилита поддерживает любой формат пакетов и всегда гарантирует безопасность соединения. Она также позволяет загружать уже созданные файлы, поэтому вы можете проверить сборку перед выпуском. В следующем примере использования `twine` для создания сборки требуется вызов скрипта `setup.py`:

```
$ python setup.py sdist bdist_wheel
$ twine upload dist/*
```

Далее поговорим о том, что такое `.pypirc`.

.pypirc

Файл `.pypirc` — это файл конфигурации, в котором хранится информация о репозиториях пакетов Python. Он должен быть размещен в вашем домашнем каталоге. Формат данного файла выглядит следующим образом:

```
 [distutils]
index-servers =
    pypi
    other

 [pypi]
repository: <repository-url>
username: <username>
password: <password>

 [other]
repository: https://example.com/pypi
username: <username>
password: <password>
```

В разделе `distutils` должна быть переменная `index-servers`, в которой перечислены секции, описывающие все доступные репозитории и учетные данные для них. В каждой секции для каждого хранилища есть следующие три переменные:

- `repository` — URL репозитория пакетов (по умолчанию pypi.org);
- `username` — имя пользователя для авторизации в данном репозитории;
- `password` — пароль пользователя для авторизации в данном репозитории (в виде обычного текста).

Обратите внимание: хранение пароля от репозитория в виде простого текста — рискованное решение с точки зрения безопасности. Вы всегда можете оставить это поле пустым, и тогда у вас при необходимости будет появляться предложение ввести пароль.

Файл `.pypirc` должен поддерживаться каждым инструментом упаковки, созданным в Python. Это не всегда выполняется для всех утилит работы с пакетами, однако данный файл поддерживается наиболее важными из них: `pip`, `twine`, `distutils` и `setuptools`.

Сравним пакеты исходного кода и сборки.

Пакеты с исходным кодом и пакеты сборок

В целом можно выделить два типа создания пакетов Python:

- исходный код;
- дистрибутивы (бинарные файлы).

Пакеты исходного кода проще и не зависят от платформы использования. Для чистых пакетов Python это не проблема. В таком пакете есть только исходный код Python, что само по себе означает высокую портируемость.

Все немного усложняется, если к вашему пакету подключаются расширения, например, в пакетах на C. Исходный код отлично подходит при условии, что у пользователя пакета есть надлежащий набор инструментов разработки в своей среде. В основном это компилятор и соответствующие заголовочные файлы C. Для таких случаев лучше подходит именно формат сборки, так как в нее входят уже собранные расширения для конкретных платформ.

Рассмотрим команду `sdist`.

sdist

Команда `sdist` — самая простая из доступных команд. Она создает дерево релиза, в которое копируется все, что нужно для запуска пакета. Оно архивируется в один или несколько файлов (обычно создается один архив). В целом архив — это просто копия дерева исходного кода.

Эта команда — самый простой способ распространения пакета, независимого от целевой системы. Она создает каталог `dist/` для хранения архивов, подлежащих распространению. Прежде чем создавать первый дистрибутив, следует вызвать функцию `setup()` с номером версии. Если вы этого не сделаете, то модуль `setuptools` будет принимать значение по умолчанию `version = '0.0.0'`:

```
from setuptools import setup

setup(name='acme.sql', version='0.1.1')
```

При каждом выпуске пакета номер версии нужно увеличивать, чтобы система знала, что пакет изменился.

Рассмотрим следующую команду `sdist` для пакета `acme.sql` версии `0.1.1`:

```
$ python setup.py sdist
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)

$ ls dist/
acme.sql-0.1.1.tar.gz
```

В Windows тип архива по умолчанию будет ZIP.

Версия используется для того, чтобы задать имя архива, который в дальнейшем можно будет скачивать/распространять и устанавливать на любую систему с Python. В дистрибутиве `sdist`, если пакет содержит библиотеки или расширения C, за их компиляцию будет отвечать система, в которой выполняется установка. Это очень характерно для систем на основе macOS и Linux, поскольку в них обычно есть компилятор. В Windows такая ситуация встречается реже. Именно поэтому у пакета должен быть грамотно продуманный дистрибутив, если пакет предназначен для запуска на нескольких платформах.

Далее рассмотрим команды `bdist` и `wheels`.

bdist и wheels

Распространить скомпилированный дистрибутив в модуле `distutils` помогает команда `build`. Она компилирует пакет в следующие четыре этапа:

- `build_py` — создает чистые модули Python с помощью байтовой компиляции и копирует их в папку сборки;

- `build_clib` — создает библиотеки C, если таковые входят в пакет, используя компилятор Python, и создает статическую библиотеку в папке сборки;
- `build_ext` — создает расширения C и помещает результат в папку сборки, например, `build_clib`;
- `build_scripts` — создает модули, которые помечаются как скрипты. Кроме того, изменяет путь интерпретатора после установки первой строки (с использованием префикса `!#`) и устанавливает режим файла так, чтобы он был исполняемым.

Каждый из этих этапов — это команда, которую можно вызвать независимо от других. Результатом процесса компиляции является папка `build`, в которой лежит все необходимое для установки пакета. В пакете `distutils` пока еще нет возможности кросс-компиляции. Это значит, что результат выполнения команды всегда зависит от системы, для которой делается сборка.

Если нужно создать расширение C, то в процессе сборки можно будет использовать компилятор по умолчанию и заголовочный файл Python (`Python.h`). Этот *включаемый* файл доступен с момента, когда Python только-только появился. Для упакованного дистрибутива вам, вероятно, потребуется дополнительный пакет, зависящий от вашей системы. По крайней мере, в популярных дистрибутивах Linux его часто называют `python-dev`. Он содержит все необходимые файлы заголовков для создания расширений Python.

Компилятор C, используемый в процессе сборки, — это компилятор по умолчанию для вашей операционной системы. Для систем на основе Linux или macOS это *gcc* или *clang* соответственно. Для Windows можно задействовать Microsoft Visual C++ (есть бесплатная версия в формате командной строки). Кроме того, можно взять проект с открытым исходным кодом MinGW. Это настраивается в `distutils`.

Команда `build` используется командой `bdist` для сборки бинарного дистрибутива. Он вызывает `build` и все зависимые команды, а затем создает архив таким же образом, как и `sdist`.

Создадим бинарный дистрибутив для `acme.sql` на macOS следующим образом:

```
$ python setup.py bdist
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)

$ ls dist/
acme.sql-0.1.1.macOSx-10.3-fat.tar.gz acme.sql-0.1.1.tar.gz
```

Обратите внимание: имя вновь созданного архива содержит имя системы и дистрибутива, на котором она была построена (macOS 10.3).

Та же команда, будучи выполненной на Windows, создаст другой архив:

```
C:\acme.sql> python.exe setup.py bdist
...

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)   22 239 752 192 bytes free
```

Если, помимо исходного дистрибутива, пакет содержит код C, то важно выпустить как можно больше бинарных дистрибутивов. По крайней мере, бинарный дистрибутив на Windows важен для тех, у кого нет установленного компилятора C.

Бинарный релиз содержит дерево, которое можно скопировать непосредственно в дерево Python. В нем содержится папка, копируемая в папку `site-packages` Python. Кроме того, он может включать кэшированные файлы байт-кода (`*.pyc` в Python 2 и `__pycache__/*.pyc` в Python 3).

Другой вид дистрибутивов — это *wheels*, которые предоставляются пакетом `wheel`. Будучи установленным (например, с помощью `pip`), пакет `wheel` добавляет к `distutils` новую команду `bdist_wheel`. Она позволяет создавать специфичные для платформы дистрибутивы (пока что только для Windows, macOS и Linux), которые работают лучше, чем обычные дистрибутивы `bdist`. Их цель — заменить более старый формат дистрибутива, введенный в `setuptools`, — `eggs`, в настоящее время уже неактуальный, и в этой книге мы о нем говорить не будем. Перечень преимуществ использования `wheels` довольно длинный. Вот те из них, которые упомянуты на странице *Python Wheels* (pythonwheels.com):

- более быстрая установка пакетов чистого Python и нативных расширений C;
- позволяет избежать выполнения произвольного кода для установки (файлов `setup.py`);
- установка расширения C не требует компилятора на Windows, macOS или Linux;
- обеспечивает лучшее кэширование для тестирования и непрерывной интеграции;
- создает файлы `.pyc` как часть установки, чтобы обеспечить их соответствие используемому интерпретатору Python;
- более последовательная установка на разных платформах и машинах.

В соответствии с рекомендациями PyPA формат `wheels` следует использовать по умолчанию. В течение очень долгого времени бинарные `wheels` не поддерживались на Linux, но, к счастью, это уже в прошлом. Бинарные `wheels` на Linux называются *manylinux wheels*. Процесс их создания, к сожалению, не так прост, как на Windows и macOS. Для таких `wheels` в PyPA поддерживаются специальные образы Docker,

которые служат в качестве готовой среды сборки. Код этих образов и дополнительная информация есть в официальном репозитории на GitHub: github.com/pypa/manylinux.

В следующем разделе рассмотрим исполняемые файлы.

Исполняемые файлы

Создание отдельных исполняемых файлов — тема, которую часто упускают в материалах, посвященных упаковке кода Python. Главным образом это связано с тем, что в стандартной библиотеке Python мало подходящих инструментов, позволяющих программистам создавать простые исполняемые файлы, которые пользователи могли бы запустить, не прибегая к установке интерпретатора Python.

Компилируемые языки имеют большое преимущество над Python: они позволяют создать исполняемое приложение для данной архитектуры системы, которое пользователи могут запустить, не имея каких-либо знаний о базовой технологии. Для запуска кода Python, распространяемого в виде пакета, требуется наличие интерпретатора Python. Это создает большие неудобства для пользователей, которые не имеют достаточных технических навыков.

Удобные для разработчиков операционные системы, такие как macOS или большинство дистрибутивов Linux, поставляются с предустановленным интерпретатором Python. Для пользователей приложения Python вполне можно распространять в качестве исходного пакета, зависящего от конкретной *директивы интерпретатора* в главном файле скрипта, который в народе называется *shebang*. У большинства приложений Python он имеет следующий вид:

```
#!/usr/bin/env python
```

Такая директива, указанная в первой строке скрипта, говорит о том, как его следует интерпретировать в версии Python по умолчанию для данной среды. Директива может быть написана в более подробной форме, которая требует определенной версии Python, например `python3.4`, `python3`, `python2` и т. д. Обратите внимание: это будет работать в самых популярных системах POSIX, но портируемость при этом полностью теряется. Данное решение основано на существовании определенных версий Python, а также доступности переменной `env` в `/usr/bin/env`. Оба эти предположения не всегда срабатывают на некоторых операционных системах. Кроме того, shebang вообще не работает на Windows. Вдобавок установка и настройка Python-окружения на Windows бывает сложной даже для опытных разработчиков, и не стоит ожидать, что неопытные пользователи смогут сделать это сами.

Стоит понять и принять, что обычный пользователь привык к простоте в работе на компьютере. Пользователи обычно ожидают, что приложения можно запускать с рабочего стола, просто дважды щелкнув на их ярлыке. Не каждое рабочее окружение сможет «понять»/поддержать написанных на Python приложений, если оно распространяется в виде исходного кода.

Поэтому будет лучше, если мы сможем создать бинарный дистрибутив, который станет работать так же, как и любой другой скомпилированный исполняемый файл. К счастью, можно создать исполняемый файл, в состав которого входит и интерпретатор Python, и наш проект. Это позволяет пользователям открывать наше приложение, не заботясь о Python или любой другой зависимости.

Посмотрим, когда бывают полезны исполняемые файлы.

Когда бывают полезны исполняемые файлы

Исполняемые файлы удобны в ситуациях, когда простота для пользователя важнее, чем возможность копаться в коде. Обратите внимание: распространение приложения в виде исполняемого файла усложняет чтение или изменение кода приложения, однако не делает это полностью невозможным. Такое распространение — способ не защитить код приложения, а упростить взаимодействие с приложением.

Исполняемые файлы должны быть предпочтительным способом распространения приложений для технически не подкованных конечных пользователей, а также единственным разумным способом распространения любого приложения Python для Windows.

Исполняемые файлы, как правило, хороши в таких случаях, как:

- приложения, зависящие от конкретной версии Python, которой может и не быть в целевой операционной системе;
- приложения, использующие модифицированный скомпилированный код CPython;
- приложения с графическим интерфейсом;
- проекты, в которых много бинарных расширений, написанных на разных языках;
- игры.

В следующем подразделе рассмотрим некоторые из популярных инструментов.

Популярные инструменты

В Python нет встроенной поддержки создания исполняемых файлов. К счастью, сообщество создало несколько проектов, решающих эту проблему с различной степенью успеха. Наиболее известны следующие четыре:

- PyInstaller;
- cx_Freeze;
- py2exe;
- py2app.

Они немного отличаются друг от друга в эксплуатации, и у каждого из них есть свои недостатки. Прежде чем выбрать свой инструмент, вы должны решить, какая у вас целевая платформа, поскольку каждый инструмент упаковки поддерживает определенный набор операционных систем.

Такое решение лучше всего принимать в самом начале жизненного цикла проекта. Ни один из этих инструментов не требует глубокого взаимодействия с вашим кодом, но если вы начнете создавать автономные пакеты своевременно, то сможете автоматизировать весь процесс и сэкономить время и средства на будущую интеграцию. Оставив решение на потом, вы можете оказаться в ситуации, когда проект усложнится настолько, что ни один инструмент вашу задачу не решит. Создание отдельного исполняемого файла в этом случае будет проблематичным и займет много времени.

В следующем пункте рассмотрим PyInstaller.

PyInstaller

PyInstaller (www.pyinstaller.org) на сегодняшний день является самой продвинутой программой для превращения пакетов Python в исполняемые файлы. В ней предусмотрена наиболее широкая мультиплатформенная совместимость среди всех прочих решений, и мы рекомендуем использовать именно ее. PyInstaller поддерживает следующие платформы:

- Windows (32 и 64 бита);
- Linux (32 и 64 бита);
- macOS (32 и 64 бита);
- FreeBSD, Solaris и AIX.

Поддерживаемые версии Python — Python 2.7, 3.3–3.5. Программа доступна на PyPI, поэтому ее можно установить в рабочей среде с помощью `pip`. При возникновении проблем с установкой вы всегда можете скачать инсталлятор со страницы проекта.

К сожалению, кросс-платформенная сборка (кросс-компиляция) не поддерживается, поэтому если вы хотите создать исполняемый файл для конкретной платформы, то выполнять сборку вам необходимо на данной платформе. Сегодня это не такая большая проблема, поскольку появилось много разных инструментов виртуализации. При отсутствии на вашем компьютере нужной системы вы всегда можете использовать Vagrant, который позволяет запустить желаемую операционную систему на виртуальной машине.

Применять PyInstaller для простых приложений очень легко. Предположим, что наше приложение содержится в скрипте с названием `myscript.py`. Это будет стандартный *hello world*. Мы хотим создать исполняемый файл для пользователей

Windows, и файлы исходного кода находятся в папке D://dev/app. Наше приложение можно собрать следующей короткой командой:

```
$ pyinstaller myscript.py
2121 INFO: PyInstaller: 3.1
2121 INFO: Python: 2.7.10
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 достаточно велик, даже для простых приложений, так что в предыдущем примере мы его сократили. Если работать на Windows, то полученная структура каталогов и файлов будет выглядеть следующим образом:

```
$ tree /0066
|   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
          python27.dll
          select.pyd
          unicodedata.pyd
          _hashlib.pyd
```

В каталоге `dist/myscript` находится собранное приложение, которое теперь можно распространять среди пользователей. Обратите внимание: распространять нужно весь каталог. В нем находятся все дополнительные файлы, необходимые для запуска нашего приложения (DLL, скомпилированные библиотеки расширения и т. д.). Более компактный дистрибутив можно получить с помощью переключателя `--onefile` команды `pyinstaller` следующим образом:

```
$ pyinstaller --onefile myscript.py
(...)

$ tree /f
├───build
│   └───myscript
│         myscript.exe.manifest
│         out00-Analysis.toc
│         out00-EXE.toc
│         out00-PKG.pkg
│         out00-PKG.toc
│         out00-PYZ.pyz
│         out00-PYZ.toc
│         warnmyscript.txt
│
└───dist
       myscript.exe
```

При сборке с опцией `--onefile` распространять нужно только один исполняемый файл из каталога `dist` (здесь `myscript.exe`). Для небольших приложений это, вероятно, предпочтительный вариант.

Одним из побочных эффектов выполнения команды `pyinstaller` является создание файла `*.spec`. Это автогенерируемый модуль Python, содержащий спецификацию того, как создавать исполняемые файлы из исходного кода. Ниже показан пример файла спецификации, созданного автоматически 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 полезно, если в вашей сборке было задано много настроек. Далее вы сможете применять его в качестве аргумента команды `pyinstaller` вместо вашего скрипта Python следующим образом:

```
$ pyinstaller.exe myscript.spec
```

Обратите внимание: это реальный модуль Python, так что вы можете расширить его и выполнять более сложные настройки в процессе сборки. Настройка файла `.spec` особенно полезна, когда у вас много целевых платформ. Кроме того, некоторые опции `pyinstaller` недоступны через интерфейс командной строки и могут применяться только при изменении файла `.spec`.

PyInstaller — это серьезный инструмент, к тому же простой в использовании для большинства программ. В любом случае, если вы хотите применять его для распространения приложений, рекомендуется внимательно ознакомиться с документацией.

В следующем пункте рассмотрим `cx_Freeze`.

cx_Freeze

Инструмент cx_Freeze (cx-freeze.sourceforge.net) также служит для создания исполняемых файлов. Это более простое решение, чем PyInstaller, однако поддерживает следующие три основные платформы:

- Windows;
- Linux;
- macOS.

Как и PyInstaller, этот инструмент не позволяет выполнять кросс-платформенную сборку, поэтому вам придется создавать исполняемые файлы в той же

операционной системе, которая является целевой. Основной недостаток cx_Freeze — он не позволяет создавать однофайловые реализации. Приложения, созданные с помощью cx_Freeze, должны поставляться с соответствующими DLL-файлами и библиотеками. Предположим, у нас есть такое же приложение, какое мы показали в пункте PyInstaller. Тогда пример использования тоже будет простым:

```
$ cxfreeze myscript.py
copying C:\Python27\lib\site-packages\cx_Freeze\bases\Console.exe ->
D:\dev\app\dist\myscript.exe
copying C:\Windows\system32\python27.dll ->
D:\dev\app\dist\python27.dll
writing zip file D:\dev\app\dist\myscript.exe
(...)
copying C:\Python27\DLLs\bz2.pyd -> D:\dev\app\dist\bz2.pyd
copying C:\Python27\DLLs\unicodedata.pyd -> D:\dev\app\dist\unicodedata.pyd
Resulting structure of files is as follows:

$ tree /f
|  myscript.py
|
└──dist
      bz2.pyd
      myscript.exe
      python27.dll
      unicodedata.pyd
```

Вместо создания собственного формата для сборки (как это делает PyInstaller) cx_Freeze расширяет пакет `distutils`. Это значит, что вы можете настроить сборку вашего исполняемого файла с помощью скрипта `setup.py`. Это делает cx_Freeze очень удобным инструментом, если вы уже распространяете пакет, используя `setuptools` или `distutils`, поскольку дополнительная интеграция требует лишь небольших изменений в `setup.py`. Вот пример такого скрипта `setup.py` с применением `cx_Freeze.setup()` для создания отдельных исполняемых файлов в Windows:

```
import sys
from cx_Freeze import setup, Executable

# Зависимости обнаруживаются автоматически, но могут требовать донастройки
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")]
)
```

С таким файлом новый исполняемый файл можно создать, добавив новую команду `build_exe` в скрипт `setup.py` следующим образом:

```
$ python setup.py build_exe
```

Применять cx_Freeze немного проще, чем PyInstaller, и интеграция с пакетом `distutils` — это очень полезная функция. К сожалению, у неопытных разработчиков могут возникнуть кое-какие трудности по следующим причинам:

- установка с помощью `pip` под Windows может быть проблематична;
- официальная документация местами довольно скудна.

В следующем пункте рассмотрим py2exe и py2app.

py2exe и py2app

py2exe (www.py2exe.org/) и py2app (py2app.readthedocs.io/en/latest) — две взаимодополняющие программы, которые интегрируются с упаковкой Python через `distutils` или `setuptools` для создания исполняемых файлов. Мы говорим о них обеих, поскольку они очень похожи в использовании и имеют общие недостатки. Основным из них является то, что обе программы ориентированы только на одну платформу:

- py2exe позволяет создавать исполняемые файлы для Windows;
- py2app позволяет создавать приложения для macOS.

Поскольку в использовании они очень похожи и для работы требуется только модификация скрипта `setup.py`, эти пакеты дополняют друг друга. В документации проекта py2app приведен следующий пример скрипта `setup.py`, который позволяет создавать исполняемые файлы с помощью правильного инструмента (py2exe или py2app), в зависимости от применяемой платформы:

```
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 sys.argv
        # будет использоваться для открытия файлов
        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 команду setup.py install
        # и устанавливают основной скрипт как таковой
        scripts=[mainscript],
    )

setup(
    name="MyApplication",
    **extra_options
)
```

Используя такой скрипт, вы можете создать исполняемый файл Windows с помощью команды `python setup.py py2exe`, а для macOS — с помощью `python setup.py py2app`. Кросс-компиляция, разумеется, невозможна.

Несмотря на очевидные недостатки py2app и py2exe и меньшую гибкость по сравнению с PyInstaller или cx_Freeze, познакомиться с этими инструментами полезно. В некоторых случаях PyInstaller или cx_Freeze неправильно создают исполняемый файл. В таких ситуациях всегда стоит проверить, могут ли другие решения работать с вашим кодом.

Безопасность кода Python в исполняемых пакетах

Важно знать, что исполняемые файлы никак не защищают код приложения. Декомпилировать встроенный код из таких исполняемых файлов довольно сложно, но все-таки реально. Еще более важно то, что результаты подобной декомпиляции (выполненной надлежащими инструментами) могут выглядеть поразительно похожими на исходный код.

Данный факт делает исполняемые файлы Python неподходящим решением для проектов с закрытым кодом, в которых утечка кода приложения может нанести вред организации. Таким образом, если весь ваш бизнес зависит от кода этого приложения, то вам нужен другой способ распространения приложения. Возможно, распространение программного обеспечения в качестве сервиса будет более удачным выбором.

Усложнение декомпиляции. Как уже говорилось, нет надежного способа защитить приложение от декомпиляции с помощью доступных на данный момент инструментов. Тем не менее есть несколько способов, позволяющих усложнить этот процесс. Однако «труднее» не означает «менее вероятно». Для некоторых из нас самые сложные проблемы заманчивее всего. И в конечном итоге цена решения данной задачи — код, который вы пытались защитить.

Обычно процесс декомпиляции состоит из следующих этапов.

1. Извлечение двоичного представления байт-кода проекта из исполняемого файла.
2. Отображение двоичного представления в байт-код конкретной версии Python.
3. Перевод байт-кода в AST.
4. Воссоздание кода непосредственно из AST.

Предоставлять точные решения для сдерживания разработчиков от обратного проектирования исполняемых файлов было бы бессмысленно по понятным причинам. Ниже представлены несколько идей о том, как усложнить процесс декомпиляции или обесценить результаты.

- Удаление любых метаданных кода, доступных во время выполнения (строки документации), чтобы конечные результаты были немного менее читабельными.
- Изменение значений байт-кода, используемых интерпретатором CPython. В результате преобразование из двоичного кода в байт-код, а затем в AST потребует больших усилий.
- Использование версии исходного кода CPython, модифицированной таким сложным образом, что даже при наличии исходного кода приложения вы ничего не сможете сделать без декомпиляции модифицированного CPython.
- Применение скриптов обфускации перед сборкой в исполняемый файл. Такие скрипты делают исходный код менее ценным после декомпиляции.

Такие решения значительно усложняют процесс разработки. Некоторые из них требуют очень глубокого понимания времени выполнения Python, и в каждом есть ловушки и недостатки. В целом все это послужит лишь отсрочкой неизбежного. Когда ваш трюк раскусят, все ваши затраты времени и ресурсов будут напрасны.

Единственный надежный способ не допустить утечки вашего кода за пределы вашего приложения — это не отправлять приложение пользователю в какой-либо форме. А это возможно, только когда работа вашей организации в целом герметична.

Резюме

В данной главе мы обсудили детали экосистемы упаковки Python. Теперь вы должны знать, какие инструменты соответствуют вашей задаче упаковки, а также какие типы дистрибутивов нужны проекту. Вы также должны знать популярные методы решения общих проблем и способы предоставления полезных метаданных вашему проекту.

Мы также обсудили тему исполняемых файлов, которые очень полезны при распространении приложений для ПК.

В следующей главе мы будем опираться на то, что узнали в этой, и покажем, как эффективно справляться с развертываниями кода надежным и автоматизированным способом.

8 Развертывание кода

Даже совершенный код (при условии, что он вообще существует) бесполезен, если неработоспособен. Итак, чтобы служить какой-либо цели, код должен быть установлен на целевой машине (компьютере) и выполнен. Процесс создания определенной версии приложения или сервиса, доступного для конечных пользователей, называется развертыванием.

В случае с приложениями для ПК все кажется простым: вы должны предоставить скачиваемый пакет с дополнительным установщиком, если необходимо. Задача пользователя — скачать и установить пакет в своей среде. Ваша ответственность — сделать процесс максимально простым и удобным для пользователя. Правильная упаковка — непростая задача, но мы уже описали некоторые инструменты в предыдущей главе.

Удивительно, но все становится гораздо сложнее, когда ваш код не является автономным продуктом. Если ваше приложение — продаваемый пользователям сервис, то это вы должны запустить его в своей инфраструктуре. Данный сценарий типичен для веб-приложения или любого другого сервиса. В таком случае код разворачивается для работы на удаленных машинах, физический доступ к которым есть у разработчиков. Это особенно актуально, если вы уже являетесь пользователем облачных сервисов, таких как *Amazon Web Services (AWS)* или Heroku.

В этой главе мы сосредоточимся на развертывании кода на удаленных хостах из-за очень высокой популярности Python в области построения различных веб-сервисов и продуктов. Несмотря на высокую портируемость данного языка, у него нет какого-то конкретного качества, которое делает его код легким для развертывания. Важнее всего то, как создается ваше приложение и какие процессы используются для развертывания в целевых средах. Таким образом, в этой главе:

- каковы основные проблемы развертывания кода в удаленных средах;
- как создавать на Python легкие в развертывании приложения;
- как перезагружать веб-сервисы без простоев;
- как использовать экосистему Python с использованием пакетов при развертывании кода;
- как правильно управлять кодом, работающим удаленно.

Технические требования

Скачать различные инструменты мониторинга и обработки журналов, упомянутые в этой главе, можно со следующих сайтов:

- Munin: munin-monitoring.org;
- Logstash, Elasticsearch и Kibana: www.elastic.co;
- Fluentd: www.fluentd.org.

Ниже приведены упомянутые в этой главе пакеты Python, которые можно скачать с PyPI:

- `fabric`;
- `devpi`;
- `circus`;
- `uwsgi`;
- `gunicorn`;
- `sentry_sdk`;
- `statsd`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по ссылке github.com/packtpublishing/expert-python-programming-third-edition/tree/master/chapter8.

Двенадцатифакторное приложение

Главное правило безболезненного развертывания — создание приложения таким, чтобы процесс был максимально простым и оптимизированным. Речь идет об устранении препятствий и поощрении устоявшихся приемов. Соблюдение общепринятых методов особенно важно в организациях, где за разработку отвечают одни люди (команда разработки, или Dev), а за развертывание и сопровождение среды выполнения — другие (команда эксплуатации, или Ops).

Все задачи, связанные с сопровождением сервера, мониторингом, развертыванием, настройкой и т. д., часто объединяют в одно понятие *«эксплуатация»*. Даже в организациях, не имеющих отдельных команд для решения таких задач, лишь некоторые из разработчиков имеют право заниматься задачами развертывания и сопровождения удаленных серверов. Общее название такой должности — DevOps. Кроме того, часто бывает так, что каждый член команды разработчиков отвечает за эксплуатацию, поэтому всех участников такой команды можно назвать DevOps.

Независимо от структуры вашей организация и обязанностей каждого разработчика все должны знать, как устроена эксплуатационная деятельность и как код разворачивается на удаленном сервере, поскольку в конечном счете среда исполнения и ее конфигурация являются скрытой частью создаваемого вами продукта.

Следующие общепринятые приемы и соглашения имеют значение в основном по двум причинам.

- В каждой компании существует текучка кадров. Используя лучшие подходы, вы облегчите вхождение в работу новых членов команды. Конечно, нельзя быть уверенными на сто процентов, что новые сотрудники уже знакомы с общими методами конфигурирования системы и запуска приложений надежным способом, но по крайней мере вы можете сделать более вероятной их быструю адаптацию.
- В организациях, где ответственность за развертывание несут определенные люди, это позволяет уменьшить разногласия между членами команд эксплуатации и разработки.

Пример практики, которая поощряет создание легко развертываемых приложений, — манифест под названием *«Двенадцатифакторное приложение»*. Это общая языковая методология создания приложений в виде сервисов. Одна из целей этой практики — упростить развертывание приложений, а также выдвинуть на первый план другие темы, такие как удобство сопровождения и масштабируемость.

Из названия ясно, что «Двенадцатифакторное приложение» состоит из 12 правил:

- *кодовая база* — одна кодовая база отслеживается и системой контроля версий, и при развертывании;
- *зависимости* — явное объявление и изоляция зависимостей;
- *конфигурация* — хранение конфигурации в среде разработки;
- *резервирование* — работа со службами резервирования как с прикрепленными ресурсами;
- *сборка, релиз, запуск* — строгое разделение этапов сборки и запуска;
- *процессы* — выполнение приложения в качестве одного процесса или более без состояния;
- *привязка портов* — экспорт через привязку портов;
- *параллелизм* — масштабирование через модель процесса;
- *простота* — максимальная устойчивость, быстрый запуск и корректная остановка;
- *паритет разработки и производства* — старайтесь держать develop-, stage- и production-ветки максимально похожими;
- *журналы* — журналы обрабатываются как потоки событий;
- *администрирование* — выполнение задач администрирования/управления в одинарных процессах.

Останавливаться на каждом из этих правил мы не будем, а вместо этого приведем ссылку на страницу «Двенадцатифакторное приложение» (12factor.net), где дано обширное обоснование каждого фактора с примерами инструментов для различных фреймворков и сред.

В этой главе мы будем пытаться соблюдать эти правила, поэтому некоторые из них обсудим более подробно по мере необходимости. Показанные методы и примеры иногда могут слегка отличаться от этих 12 факторов, так как они не являются истиной в последней инстанции. Эти правила хороши до тех пор, пока выполняют свою задачу. Но самое важное — это рабочее приложение (продукт), а не совместимость с некоторой методологией.

В следующем разделе рассмотрим различные подходы к автоматизации развертывания.

Различные подходы к автоматизации развертывания

С появлением контейнеризации приложений (Docker и подобных технологий), современных инструментов поставки ПО (например, Puppet, Chef, Ansible и Salt) и систем управления инфраструктурой (например, Terraform и SaltStack) команды разработки и эксплуатации получили много инструментов для организации и управления кодом и настройки удаленных систем. Каждое решение имеет свои плюсы и минусы, поэтому средства автоматизации нужно выбирать разумно и с учетом особенностей выстроенных процессов и методологий разработки.

Быстроработающие команды, которые используют микросервисную архитектуру и часто развертывают код (возможно, даже одновременно в параллельных версиях), наверняка любят контейнерные системы, такие как Kubernetes, или используют выделенные сервисы, предоставляемые их облачным провайдером (например, AWS). Команды, по старинке пишущие монолитные приложения и запускающие их на своих собственных серверах, вероятно, захотят использовать автоматизацию низкого уровня и системы поставки ПО. На самом деле четких правил здесь нет, и всегда найдутся команды любого размера, в которых используется тот или иной подход к резервированию ПО, развертыванию кода и оркестровке приложений. Ограничениями здесь выступают лишь ресурсы и знания.

Именно поэтому очень трудно кратко перечислить универсальный набор инструментов и решений, который подошел бы под потребности и возможности каждого разработчика и каждой команды. Поэтому в данной главе мы поговорим о простом подходе к автоматизации с помощью Fabric. Можно было бы сказать, что такой подход устарел, и это, наверное, правда. *Наиболее современными* сейчас считаются контейнерные системы оркестровки в стиле Kubernetes, позволя-

ющие использовать контейнеры Docker для создания быстрых, удобных в сопровождении, масштабируемых и воспроизводимых программ. Но эти системы характеризуются довольно крутой кривой обучения, и в одной главе о них рассказать не получится. А вот Fabric, с другой стороны, очень прост для понимания и является действительно отличным инструментом для начинающих в вопросах автоматизации.

В следующем подразделе рассмотрим использование Fabric в области автоматизации развертывания.

Использование Fabric для автоматизации развертывания

В очень маленьких проектах можно разворачивать код *вручную*, то есть вручную вводить последовательность команд через удаленные оболочки, каждый раз внедряя новую версию кода и выполняя его в такой оболочке. Однако даже для проекта средних размеров подобный подход чреват ошибками, утомителен и будет пустой тратой большей части вашего самого дорогого ресурса — времени.

Для этого и нужна автоматизация. Простое правило: *если вам необходимо выполнить некую задачу вручную по меньшей мере дважды, то ее нужно автоматизировать, чтобы не выполнять в третий раз.*

Существуют различные инструменты, которые позволяют автоматизировать те или иные задачи.

- Средства удаленного выполнения, такие как Fabric, используются для автоматизированного выполнения кода на нескольких удаленных хостах по требованию.
- Средства управления конфигурацией, такие как Chef, Puppet, Cfengine, Salt и Ansible, предназначены для автоматизированной конфигурации удаленных хостов (сред выполнения). Их можно задействовать для создания служб резервирования (базы данных, кэши и т. д.), системных разрешений, пользователей и т. д. Большинство из них применимы и в качестве инструмента для удаленного выполнения (как, например, Fabric), но, в зависимости от их архитектуры, это может быть более или менее удобно.

Решения для управления конфигурациями — сложная тема, которая заслуживает отдельной книги. Правда в том, что у простейших фреймворков удаленного выполнения низкий порог вхождения и для небольших проектов они более предпочтительны. На самом деле каждый инструмент управления конфигурацией, позволяющий декларативно определить конфигурацию ваших машин, где-то глубоко внутри содержит слой удаленного выполнения.

Кроме того, отдельные инструменты управления конфигурацией не очень хорошо подходят для реального автоматизированного развертывания кода. Один из таких примеров — инструмент Puppet, в котором неудобно выполнять явный запуск каких-либо команд оболочки. Именно поэтому многие предпочитают использовать оба решения, дополняющие друг друга: управление конфигурацией для настройки среды на системном уровне и удаленное выполнение по требованию.

Fabric (www.fabfile.org) до сих пор наиболее популярное решение, которое разработчики на Python задействуют в целях автоматизации удаленного выполнения. Это библиотека и инструмент командной строки Python для оптимизации использования SSH в рамках задач управления развертыванием или приложениями. Остановимся на нем, поскольку для старта это самое оно. Имейте в виду, что в зависимости от ваших потребностей оно может быть не лучшим решением проблем. Во всяком случае, это отличный пример инструмента, позволяющего автоматизировать некоторые ваши действия.

Конечно, можно автоматизировать всю работу, применяя только скрипты Bash, но это очень утомительно и порождает ошибки. В Python есть более удобные способы обработки строк, поощряющие модульность кода. Fabric — лишь инструмент для объединения выполняемых команд через SSH. То есть вам все равно нужно будет знать, как использовать интерфейс командной строки и его утилиты в удаленной среде.

Итак, если вы хотите строго следовать методологии «Двенадцатифакторного приложения», то не должны поддерживать код в исходном дереве развернутого приложения.

Сложные проекты очень часто строятся из различных компонентов, поддерживаемых из отдельных кодовых баз, так что это еще одна причина, почему хорошо бы иметь один отдельный репозиторий для всех конфигураций компонентов проекта и скриптов Fabric. Это делает развертывание различных сервисов более последовательным и поощряет повторное использование кода.

Чтобы начать работать с Fabric, вам необходимо установить пакет `fabric` (используя `pip`) и создать скрипт под названием `fabfile.py`. Этот скрипт, как правило, находится в корневом каталоге вашего проекта. Обратите внимание: `fabfile.py` можно считать частью вашей конфигурации проекта.

Но прежде, чем мы создадим наш `fabfile`, определим некие начальные утилиты, призванные помочь настроить проект удаленно. Вот модуль, который мы будем называть `fabutils`:

```
import os

# Предположим, у нас есть private-репозиторий,
# созданный с devpi
PYPI_URL = 'http://devpi.webxample.example.com'
```

```
# Это обязательное расположение для хранения релизов.
# Каждый релиз расположен в отдельном каталоге виртуального окружения,
# названном как версия проекта. Есть также символический
# указатель «текущая», указывающий на самую новую версию.
# Данный указатель — реальный путь, используемый для настройки:
# .
# ├── 0.0.1
# ├── 0.0.2
# ├── 0.0.3
# ├── 0.1.0
# └── current -> 0.1.0/

REMOTE_PROJECT_LOCATION = "/var/projects/webxample"

def prepare_release(c):
    """Подготавливаем новый выпуск, создав исходный код и загрузив его
       в приватный репозиторий пакетов
    """
    c.local(f'python setup.py build sdist')
    c.local(f'twine upload --repository-url {PYPI_URL}')

def get_version(c):
    """Получаем текущую версию проекта из setuptools"""
    return c.local('python setup.py --version').stdout.strip()

def switch_versions(c, version):
    """Переключаемся между версиями, атомарно заменяя символические ссылки"""
    new_version_path = os.path.join(REMOTE_PROJECT_LOCATION, version)
    temporary = os.path.join(REMOTE_PROJECT_LOCATION, 'next')
    desired = os.path.join(REMOTE_PROJECT_LOCATION, 'current')

    # Принудительная символическая ссылка (-f), поскольку, возможно, она уже есть
    c.run(f"ln -fsT {new_version_path} {temporary}")
    # mv -T обеспечивает атомарность этой операции
    c.run(f"mv -Tf {temporary} {desired}" )
```

Пример окончательного `fabfile`, который определяет простую процедуру развертывания, будет выглядеть следующим образом:

```
from fabric import task
from .fabutils import *

@task
def uptime(c):
    """
    Запускаем команду uptime на удаленном хосте — для тестирования подключения
    """
    c.run("uptime")
```

```
@task
def deploy(c):
    """Развертывание приложения с учетом упаковки"""
    version = get_version(c)

    pip_path = os.path.join(
        REMOTE_PROJECT_LOCATION, version, 'bin', 'pip'
    )

    if not c.run(f"test -d {REMOTE_PROJECT_LOCATION}", warn=True):
        # Проект может не существовать при первом развертывании на новом хосте
        c.run(f"mkdir -p {REMOTE_PROJECT_LOCATION}")

    with c.cd(REMOTE_PROJECT_LOCATION):
        # Создать новое виртуальное окружение, используя venv
        c.run(f'python3 -m venv {version}')

        c.run(f"{pip_path} install webxample=={version} --index-url{PYPI_URL}")

    switch_versions(c, version)
    # Предположим, что Circus — наш инструмент наблюдения
    c.run('circusctl restart webxample')
```

Каждая функция, декорированная с помощью `@task`, теперь рассматривается как доступная подкоманда утилиты `fab` с пакетом `fabric`. Вы можете перечислить все имеющиеся подкоманды с помощью переключателя `-l` или `--list`. Код показан в следующем фрагменте:

```
$ fab --list
Available commands:
    deploy Deploy application with packaging in mind
    uptime Run uptime command on remote host — for testing connection.
```

Теперь вы можете развернуть приложение в среде данного типа одной командой оболочки:

```
$ fab -H myhost.example.com deploy
```

Обратите внимание: предыдущий `fabfile` служит только для иллюстративных целей. В своем коде вы можете захотеть внедрить обработку ошибок и попробовать перезагрузить приложение, не прибегая к необходимости перезагрузки веб-сервиса. Кроме того, некоторые из представленных здесь методов могут быть неочевидны прямо сейчас, но будут объяснены позже в этой главе. К ним относятся следующие:

- развертывание приложения с кодом из private-репозитория;
- использование Circus для наблюдения на удаленном хосте.

В следующем разделе мы рассмотрим зеркальное отображение каталогов в Python.

Ваш собственный каталог пакетов или зеркало каталогов

Есть три основные причины, по которым вам может понадобиться запустить свой собственный каталог пакетов Python.

- Официальный PyPI может оказаться недоступен. Он находится в ведении Python Software Foundation и работает благодаря многочисленным пожертвованиям. Это значит, что сайт может оказаться недоступен в самое неподходящее время. Вы вряд ли захотите останавливать развертывание или упаковку в середине процесса из-за сбоя на PyPI.
- Полезно иметь многократно применяемые написанные на Python компоненты в упакованном виде даже для закрытого кода, который никогда не будет опубликован. Это упрощает кодовую базу, поскольку пакеты, используемые для различных проектов во всей компании, не нужно распространять. Вы можете просто установить их из репозитория. Это упрощает сопровождение кода и позволяет сократить затраты на разработку для всей компании, особенно если в ней много команд, работающих над различными проектами.
- Хороший прием — упаковка проекта с помощью `setuptools`. Тогда развернуть новую версию приложения будет проще некуда: `pip install --update my-application`.

> **Поставка кода**
>
> Поставка кода — практика включения кода из внешнего пакета в исходный код (репозиторий) других проектов. Это обычно делается в случаях, когда код проекта зависит от конкретной версии какого-либо внешнего пакета, который также может потребоваться в других пакетах (и в совершенно иной версии).
>
> Например, в популярном пакете requests используется некая версия urllib3 в исходном дереве, поскольку он очень тесно связан с ней и вряд ли будет работать с любой другой версией urllib3. Пример модуля, который особенно часто используется другими, — модуль six. Его можно найти в коде многочисленных популярных проектов, таких как «Джанго» (django.utils.six), Boto (boto.vendored.six) или Matplotlib (matplotlib.externals.six).
>
> Хотя поставка практикуется даже в некоторых крупных и успешных проектах с открытым исходным кодом, ее следует по возможности избегать. Такое использование оправданно лишь при определенных обстоятельствах и не должно рассматриваться в качестве заменителя управления пакетом зависимостей.

В следующем подразделе мы обсудим зеркала PyPI.

Зеркала PyPI

Проблему неполадок на PyPI можно сгладить, если скачивать пакеты через одно из зеркал. На самом деле официальный каталог пакетов Python уже поставляется через *сеть доставки контента* (Content Delivery Network, CDN), которая и является, по сути, зеркалом. Это не отменяет того факта, что время от времени все работает не очень хорошо. Использование неофициальных зеркал — тоже не решение, поскольку в данном случае могут возникнуть некоторые проблемы безопасности.

Лучше всего иметь собственное зеркало PyPI, на котором будут все нужные вам пакеты. Использовать такое зеркало будете только вы, благодаря чему гораздо легче обеспечить доступность. Еще одно преимущество — если ваш сервер обвалится, то вы сможете заняться его починкой, не полагаясь на кого-то другого. Инструмент создания зеркал, сопровождаемый и рекомендованный PyPA, — это *Bandersnatch* (pypi.python.org/pypi/bandersnatch). Он отражает все содержимое пакетов Python и может указываться в опции `index-url` для секции репозитория в файле `.pypirc` (как описано в предыдущей главе). Данное зеркало не поддерживает загрузку со стороны клиента и не имеет веб-части PyPI. Но будьте осторожны! Для полноценного зеркала нужны сотни гигабайт дискового пространства, и его размер будет продолжать расти в течение долгого времени.

Но зачем останавливаться на простом зеркале при наличии гораздо лучшей альтернативы? Весьма маловероятно, что вам потребуется зеркало всего каталога. Даже если в вашем проекте сотни зависимостей, это будет лишь незначительная часть всех имеющихся пакетов. Кроме того, огромный недостаток такого простого зеркала — невозможность загрузить туда свой собственный пакет. Может показаться, что добавленная стоимость использования Bandersnatch очень мала для такой высокой цены. И в большинстве случаев это верно. Если зеркало требуется всего лишь для одного или нескольких проектов, то гораздо лучше будет задействовать Devpi (doc.devpi.net). Это PyPI-совместимая реализация каталога пакетов, которая обеспечивает:

- частный каталог для загрузки непубличных пакетов;
- создание каталогов зеркал.

Главное преимущество Devpi по сравнению Bandersnatch — подход к созданию зеркала. Он позволяет сделать полноценное зеркало других каталогов, как и Bandersnatch, но главное не это. Вместо того чтобы делать огромную резервную копию всего репозитория, можно делать зеркала только для пакетов, которые уже запрашивались клиентами. Таким образом, всякий раз, когда инструмент для установки запрашивает пакет (`pip`, `setuptools` и `easy_install`), при отсутствии того на локальном зеркале сервер Devpi попытается скачать его с зеркала (обычно PyPI). После того как пакет скачан, Devpi будет периодически проверять его обновления, тем самым поддерживая актуальность зеркала.

В таком подходе есть небольшой риск сбоя в случае, если вы запрашиваете новый пакет, которого в зеркале еще нет, а вышестоящий репозиторий внезапно «падает». Но подобная ситуация встречается нечасто, и чаще всего вы будете использовать уже «отзеркаленные» пакеты. В остальном применение зеркал в конечном итоге гарантирует целостность, и новые версии будут скачиваться автоматически. Это кажется очень разумным компромиссом.

Теперь посмотрим, как правильно объединять и создавать дополнительные «непитоновские» ресурсы в приложении Python.

Объединение дополнительных ресурсов с пакетом Python

У современных веб-приложений много зависимостей, и их часто бывает сложно правильно установить на удаленном хосте. Например, типичный процесс самонастройки для новой версии приложения на удаленном хосте состоит из следующих этапов.

1. Создание нового виртуального окружения для изоляции.
2. Перемещение кода проекта в среду выполнения.
3. Установка последней версии требований проекта (как правило, из файла `requirements.txt`).
4. Синхронизация или перенос схемы базы данных.
5. Сборка статических файлов из кода проекта и внешних пакетов в нужном месте.
6. Компиляция файлов локализации для приложений, доступных на разных языках.

Более сложные сайты могут иметь еще больше задач, и трудности связаны в основном с фронтенд-кодом, который не зависит от ранее определенных задач, как в следующем примере.

1. Генерация файлов CSS с помощью препроцессора, например SASS или LESS.
2. Выполнение минификации, обфускации и/или конкатенации статических файлов (JavaScript и CSS-файлы).
3. Компиляция кода на языках семейства JavaScript (CoffeeScript, TypeScript и т. д.) в нативный JS.
4. Препроцессинг шаблонов файлов ответов (минификация, встраивание стиля и т. д.).

Сегодня для приложений, требующих много дополнительных средств, большинство разработчиков, вероятно, используют образы Docker. В Docker-файлах можно легко определить все шаги, необходимые для объединения всех ассетов с образом вашего приложения. Но если вы не используете Docker, то можно автоматизировать все это с помощью других инструментов, таких как Make, Bash,

Fabric или Ansible. Однако нежелательно выполнять все шаги непосредственно на удаленных хостах, на которых устанавливается приложение, и вот почему.

- Некоторые из популярных инструментов для обработки статических ассетов забирают на себя много ресурсов процессора или памяти. Их запуск в production-среде может привести к дестабилизации выполнения вашего приложения.
- Для этих инструментов очень часто требуются дополнительные зависимости, которые для нормальной работы ваших проектов не нужны. Обычно это дополнительные среды выполнения, такие как JVM, Node или Ruby. Данное обстоятельство повышает сложность управления конфигурацией и увеличивает общие затраты на техническое обслуживание.
- При развертывании приложения на нескольких серверах (десятках, сотнях или тысячах) многократно выполняется одна и та же работа, которую можно было бы сделать один раз. Если у вас есть собственная инфраструктура, то это не выльется в особые затраты, особенно при развертывании в периоды низкого трафика. Но в случае использования облачных сервисов, которые берут с вас дополнительную плату за скачки нагрузки или процессорное время, дополнительные расходы могут оказаться существенными.
- Большинство из указанных шагов занимает много времени. Вы устанавливаете код на удаленный сервер, поэтому точно не хотите, чтобы соединение сопровождалось какими-либо проблемами. Чем быстрее идет развертывание, тем меньше шанс прервать процесс.

Очевидно, что результаты этих шагов нельзя включить в репозиторий кода приложения. Есть вещи, которые все равно следует делать при выпуске каждой версии, и тут ничего не поделать. Именно здесь, очевидно, требуется правильная автоматизация, которая будет работать в нужном месте в нужное время.

Большинство действий, таких как статический сбор и предварительная обработка ассетов/кода, можно выполнять локально или в специальной среде, и фактический код, который будет развернут на удаленном сервере, потребуется лишь немножко обработать на месте. Ниже приведены наиболее существенные из таких этапов развертывания в процессе сборки дистрибутива или установки пакета.

1. Установка зависимостей Python и передача статических ассетов (файлов CSS и JavaScript) в нужное место может выполняться как часть команды `install` скрипта `setup.py`.
2. Предварительная обработка кода (обработка надмножеств JavaScript, минификация/затемнение/конкатенация ассетов и запуск SASS или LESS) и такие действия, как локализованная компиляция текста (например, `compilemessages` в Django), может быть частью команды `sdist/bdist` скрипта `setup.py`.

Включение предобработанного кода не из Python легко реализуется с помощью правильного файла `MANIFEST.in`. Зависимости, конечно, лучше всего указывать в аргументе `install_requires` функции `setup()` из пакета `setuptools`.

Для упаковки всего приложения, конечно, потребуется поработать, например указать свои собственные команды `setuptools` или переписывать существующие, но это дает вам много преимуществ и позволяет значительно ускорить развертывание проекта и сделать его более надежным.

В качестве примера воспользуемся проектом на основе Django (версии Django 1.9). Мы выбрали эту структуру, поскольку она кажется нам наиболее популярным проектом Python такого типа, и с ней вы можете быть уже знакомы. Типичная структура файлов в подобном проекте может выглядеть следующим образом:

```
$ tree . -I __pycache__ --dirsfirst
.
├── webxample
│   ├── conf
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── locale
│   │   ├── de
│   │   │   └── LC_MESSAGES
│   │   │       └── django.po
│   │   ├── en
│   │   │   └── LC_MESSAGES
│   │   │       └── django.po
│   │   └── pl
│   │       └── LC_MESSAGES
│   │           └── django.po
│   ├── myapp
│   │   ├── migrations
│   │   │   └── __init__.py
│   │   ├── static
│   │   │   ├── js
│   │   │   │   └── myapp.js
│   │   │   └── sass
│   │   │       └── myapp.scss
│   │   ├── templates
│   │   │   ├── index.html
│   │   │   └── some_view.html
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   ├── __init__.py
│   └── manage.py
├── MANIFEST.in
├── README.md
└── setup.py
15 directories, 23 files
```

Обратите внимание: данная структура немного отличается от обычной структуры Django-проекта. По умолчанию пакет, который содержит приложение WSGI, модуль настройки и конфигурацию URL, имеет то же имя, что и проект. Поскольку мы решили использовать подход с использованием пакетов, это все называлось бы `webxample`. Но тут возможна путаница, так что лучше переименовать его в `conf`. Не углубляясь в детали реализации, просто сделаем несколько простых предположений:

- в нашем примере у приложения есть несколько внешних зависимостей. Здесь будет два популярных пакета Django — `djangorestframework` и `django-allauth`, плюс один пакет не из Django — `gunicorn`;
- `djangorestframework` и `django-allauth` предоставляются в `INSTALLED_APPS` в модуле `webexample.webexample.settings`;
- приложение локализовано на трех языках (немецком, английском и польском), но мы не хотим хранить скомпилированные сообщения `gettext` в репозитории;
- мы устали от оригинального синтаксиса CSS, поэтому решили использовать более эффективный язык SCSS, который превратим в CSS с помощью SASS.

Зная структуру проекта, мы можем написать скрипт `setup.py` таким образом, чтобы `setuptools` выполнял следующие действия:

- компиляцию файлов SCSS в `webxample/myapp/статический/scss`;
- компиляцию сообщений `gettext` под `webexample/locale` из формата `.po` в `.mo`;
- установку требований;
- выполнение нового скрипта, который обеспечивает точку входа в пакет, так что возьмем пользовательскую команду вместо скрипта `manage.py`.

Здесь нам немного повезло: в Python для привязки `libsass` (портированного на C/C++ движка SASS) есть некая интеграция с `setuptools` и `distutils`. Немного поменяв настройки, мы можем добавить пользовательскую команду `setup.py` для запуска компиляции SASS. Это показано в следующем коде:

```
from setuptools import setup

setup(
    name='webxample',
    setup_requires=['libsass == 0.6.0'],
    sass_manifests={
        'webxample.myapp': ('static/sass', 'static/css')
    },
)
```

Таким образом, вместо выполнения команды `sass` вручную или подпроцесса в скрипте `setup.py` мы можем ввести команду `python setup.py build_scss` и превратить файлы SCSS в CSS. Но и этого мало. Описанные действия уже облегчили нам

жизнь, но мы хотим, чтобы все распространение было полностью автоматизировано и новые версии выпускались в одно касание. Для достижения данной цели нужно будет переписать некоторые из существующих команд `setuptools`.

Пример файла `setup.py`, выполняющего некоторые из этапов подготовки проекта через упаковку, может выглядеть следующим образом:

```
import os

from setuptools import setup
from setuptools import find_packages
from distutils.cmd import Command
from distutils.command.build import build as _build

try:
    from django.core.management.commands.compilemessages \
        import Command as CompileCommand
except ImportError:
    # Примечание: во время установки django может быть недоступен
    CompileCommand = None

# Требуется эта среда
os.environ.setdefault(
    "DJANGO_SETTINGS_MODULE", "webxample.conf.settings"
)
class build_messages(Command):
    """Пользовательская команда для создания сообщений в Django
    """
    description = """сборка сообщений gettext"""
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):

        pass

    def run(self):
        if CompileCommand:
            CompileCommand().handle(
                verbosity=2, locales=[], exclude=[]
            )
        else:
            raise RuntimeError("could not build translations")

class build(_build):
    """Переопределенная команда build с парой новых действий
    """
    sub_commands = [
```

```
        ('build_messages', None),
        ('build_sass', None),
    ] + _build.sub_commands

setup(
    name='webxample',
    setup_requires=[
        'libsass == 0.6.0',
        'django == 1.9.2',
    ],
    install_requires=[
        'django == 1.9.2',
        'gunicorn == 19.4.5',
        'djangorestframework == 3.3.2',
        'django-allauth == 0.24.1',
    ],
    packages=find_packages('.'),
    sass_manifests={
        'webxample.myapp': ('static/sass', 'static/css')
    },
    cmdclass={
        'build_messages': build_messages,
        'build': build,
    },
    entry_points={
        'console_scripts': {
            'webxample = webxample.manage:main',
        }
    }
)
```

При такой реализации мы можем собрать все ассеты и создать дистрибутив исходного пакета для проекта `webxample`, используя одну команду консоли:

```
$ python setup.py build sdist
```

При наличии собственного каталога пакетов (созданный с помощью Devpi) вы можете добавить подкоманду `install` или использовать `twine`, и этот пакет будет доступен для установки через `pip`. Если мы посмотрим на структуру исходного дистрибутива, созданного в скрипте `setup.py`, то увидим, что он содержит скомпилированные сообщения `gettext` и таблицы стилей CSS, сгенерированные из файлов SCSS:

```
$ tar -xvzf dist/webxample-0.0.0.tar.gz 2> /dev/null
$ tree webxample-0.0.0/ -I __pycache__ --dirsfirst
webxample-0.0.0/
├── webxample
│   ├── conf
│   │   ├── __init__.py
```

```
|   |   ├── settings.py
|   |   ├── urls.py
|   |   └── wsgi.py
|   ├── locale
|   |   ├── de
|   |   |   └── LC_MESSAGES
|   |   |       ├── django.mo
|   |   |       └── django.po
|   |   ├── en
|   |   |   └── LC_MESSAGES
|   |   |       ├── django.mo
|   |   |       └── django.po
|   |   └── pl
|   |       └── LC_MESSAGES
|   |           ├── django.mo
|   |           └── django.po
|   ├── myapp
|   |   ├── migrations
|   |   |   └── __init__.py
|   |   ├── static
|   |   |   ├── css
|   |   |   |   └── myapp.scss.css
|   |   |   └── js
|   |   |       └── myapp.js
|   |   ├── templates
|   |   |   ├── index.html
|   |   |   └── some_view.html
|   |   ├── __init__.py
|   |   ├── admin.py
|   |   ├── apps.py
|   |   ├── models.py
|   |   ├── tests.py
|   |   └── views.py
|   ├── __init__.py
|   └── manage.py
├── webxample.egg-info
|   ├── PKG-INFO
|   ├── SOURCES.txt
|   ├── dependency_links.txt
|   ├── requires.txt
|   └── top_level.txt
├── MANIFEST.in
├── PKG-INFO
├── README.md
├── setup.cfg
└── setup.py
16 directories, 33 files
```

Еще одно преимущество использования такого подхода заключается в том, что мы создали собственную точку входа для проекта вместо скрипта по умолчанию

`manage.py`. Теперь мы можем выполнить любую команду управления Django, задействуя эту точку входа, например:

```
$ webxample migrate
$ webxample collectstatic
$ webxample runserver
```

Для этого пришлось внести небольшое изменение в скрипт `manage.py` для совместимости с `entry_points` в `setup()`, так что основная часть своего кода обернута вызовом функции `main()`. Это показано в следующем коде:

```
#!/usr/bin/env python3
import os
import sys

def main():
    os.environ.setdefault(
        "DJANGO_SETTINGS_MODULE", "webxample.conf.settings"
    )

    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)

if __name__ == "__main__":
    main()
```

К сожалению, во многих фреймворках (включая Django) не заложена идея именно такой упаковки ваших проектов. То есть в зависимости от дальнейшего развития вашего приложения для его превращения в пакет может потребоваться много изменений. В Django это часто ведет к переписыванию неявных импортов и обновлению переменных конфигурации в файле настройки.

Еще одна проблема — последовательность релизов, созданных с помощью упаковки Python. Если разные члены команды имеют право создавать дистрибутивы приложений, то важно, чтобы этот процесс происходил в одной и той же воспроизводимой среде. При многочисленном выполнении предварительной обработки ассетов вероятна ситуация, когда пакет, созданный в двух различных средах, будет выглядеть по-разному, даже если создается из одного и того же кода. Это может быть связано с различными версиями инструментов, используемых в процессе сборки. Лучше сделать так, чтобы дистрибутивы создавались в некоторой непрерывной системе интеграции/доставки, такой как Jenkins, BuildBot, Travis CI и т. д. Еще одно преимущество состоит в том, что в этом случае вы будете уверены: пакет пройдет все необходимые испытания.

Имейте в виду: несмотря на то что распространение кода в виде пакетов Python с использованием `setuptools` может показаться элегантным, все не так просто. Потенциал есть возможность значительно упростить развертывание, и поэтому,

безусловно, такой метод стоит попробовать, но ценой ему будет повышение сложности. Если предварительная обработка вашего приложения становится слишком сложной, то вы обязательно должны подумать о создании образа Docker и развертывании приложения в контейнере.

Развертывание с помощью Docker требует дополнительной настройки и оркестровки, но в долгосрочной перспективе сэкономит вам много времени и ресурсов, которые в противном случае потребовались бы на содержание сред сборки и сложную предварительную обработку.

В следующем разделе мы рассмотрим общие соглашения и практики в отношении развертывания приложений Python.

Общие соглашения и практики

Существует ряд общих соглашений и практик развертывания, которые может знать не каждый разработчик, но они будут очевидны для тех, кто занимается эксплуатационной деятельностью. Как объяснялось во введении к данной главе, вам важно знать по крайней мере некоторые из них, даже если вы не занимаетесь разверткой кода и эксплуатацией, поскольку это позволит вам принимать более конструктивные решения в процессе разработки.

В следующем подразделе рассмотрим иерархию файловой системы.

Иерархия файловой системы

Наиболее очевидные соглашения, которые вы можете вспомнить, вероятно, касаются иерархии файловой системы и именования пользователей. Но здесь мы об этом говорить не будем. Существует, конечно, стандарт иерархий *Filesystem Hierarchy Standard (FHS)*, определяющий структуру каталогов и их содержимого в Unix и Unix-подобных операционных системах, однако на самом деле будет трудно найти реальную ОС, которая полностью совместима с FHS. Если системные дизайнеры и программисты не соблюдают эти стандарты, то трудно ожидать того же от администраторов. Мы на своем опыте видели развертку кода приложения почти везде, где это возможно, в том числе в нестандартных пользовательских каталогах на уровне корневой файловой системы. Почти всегда люди, принимавшие такие решения, приводили очень сильные аргументы:

- выбирайте с умом и избегайте сюрпризов;
- будьте последовательны по всей инфраструктуре вашего проекта;
- постарайтесь быть последовательными в организации, в которой вы работаете.

А вот что реально полезно, так это документирование соглашений для вашего проекта. Только не забудьте убедиться, что данная документация доступна каждому

заинтересованному члену команды и все знают о существовании такого документа в принципе.

В следующем подразделе мы поговорим об изоляции.

Изоляция

О необходимости изоляции и о рекомендуемых инструментах мы уже поговорили в главе 2. Причины для введения изоляции таковы: лучшая воспроизводимость среды и решение неизбежных проблем конфликтов зависимостей. Что касается развертывания, осталось добавить одну важную вещь. Вы всегда должны изолировать зависимости проекта для каждой версии приложения. На практике это означает, что всякий раз, когда вы разворачиваете новую версию приложения, необходимо создать для нее новую изолированную среду (с помощью `virtualenv` или `venv`). Старые среды тоже следует на некоторое время оставить, чтобы в случае возникновения проблем вы могли легко выполнить откат к одной из старых версий приложения.

Создание свежей среды для каждой версии помогает управлять чистотой состояния и соответствием перечню предоставляемых зависимостей. Под свежей средой мы имеем в виду создание нового дерева каталогов в файловой системе вместо обновления уже существующих файлов. К сожалению, это может усложнить процедуру корректного перезапуска сервиса, поскольку обновление файлов «на месте» — гораздо более элегантный метод.

В следующем подразделе мы рассмотрим, как применять инструменты мониторинга процессов.

Использование инструментов мониторинга процессов

Приложения на удаленных серверах обычно работают в непрерывном режиме. Если это веб-приложение, то его HTTP-сервер будет ожидать новых соединений и запросов и завершать работу только в случае возникновения какой-то неисправимой ошибки.

Конечно, не представляется возможным запустить сервис вручную в оболочке и поддерживать бесконечное соединение по SSH. Использовать `nohup`, `screen` или `tmux` для полудемонизации процесса — тоже не вариант. Такой подход ведет к провалу.

Вам необходим некий надзорный инструмент, который позволит запускать процесс приложения и управлять им. Прежде чем выбрать правильный инструмент, вы должны убедиться, что он делает следующее:

- перезапускает сервис, если тот завершает работу;
- правильно отслеживает его состояние;

- захватывает потоки `stdout/stderr` для записи в журнал;
- запускает процесс с разрешениями, зависящими от конкретного пользователя/группы;
- настраивает системные переменные среды.

Большинство дистрибутивов Unix и Linux имеют встроенные инструменты/подсистемы для наблюдения за процессами, например скрипты `initd`, `upstart` и `runit`. К сожалению, в большинстве случаев они не очень хорошо подходят для запуска программного кода на уровне пользователя и сложны в сопровождении. В частности, написать надежный скрипт `init.d` довольно трудно, поскольку для этого требуется много скриптов Bash, с которыми тяжело работать. В некоторых дистрибутивах Linux, таких как Gentoo, подход к скриптам `init.d` был изменен, вследствие чего писать их намного проще. Во всяком случае, зацикливаться на одной ОС исключительно из-за удобного инструмента мониторинга — плохая идея.

Существует два популярных в сообществе Python инструмента для управления процессами приложений — Supervisor (supervisord.org) и Circus (circus.readthedocs.org/en/latest/). Оба инструмента очень похожи в настройке и использовании. Circus немного моложе, чем Supervisor, поскольку был создан с целью устранить некоторые его недостатки. Оба инструмента конфигурируются через простой INI-подобный формат. Они не ограничены запуском процесса Python и могут быть настроены на управление любым приложением. Трудно сказать, какой из них лучше, поскольку их функциональность невероятно схожа. Однако Supervisor не работает на Python 3, поэтому не подходит нам автоматически. Python 3 все же можно запустить под Supervisor, но мы тем не менее сосредоточимся на Circus.

Предположим, что мы хотим запустить приложение `webxample` (представленное ранее в этой главе), используя веб-сервер `gunicorn` под управлением Circus. В продакшене мы бы, вероятно, запустили Circus под подходящим инструментом управления процессом на системном уровне (`initd`, `upstart`, и `runit`), особенно если он был установлен из репозитория системных пакетов. Для простоты мы будем запускать его локально внутри виртуального окружения. Минимальный файл конфигурации (названный `circus.ini`), который позволяет нам запускать наше приложение в Circus, выглядит следующим образом:

```
[watcher:webxample]
cmd = /path/to/venv/dir/bin/gunicorn webxample.conf.wsgi:application
numprocesses = 1
```

Теперь процесс `circus` может работать с этим файлом конфигурации в качестве аргумента выполнения:

```
$ circusd circus.ini
2016-02-15 08:34:34 circus[1776] [INFO] Starting master on pid 1776
2016-02-15 08:34:34 circus[1776] [INFO] Arbiter now waiting for commands
```

```
2016-02-15 08:34:34 circus[1776] [INFO] webxample started
[2016-02-15 08:34:34 +0100] [1778] [INFO] Starting gunicorn 19.4.5
[2016-02-15 08:34:34 +0100] [1778] [INFO] Listening at:
http://127.0.0.1:8000 (1778)
[2016-02-15 08:34:34 +0100] [1778] [INFO] Using worker: sync
[2016-02-15 08:34:34 +0100] [1781] [INFO] Booting worker with pid: 1781
```

Теперь вы можете использовать команду `circusctl`, чтобы запустить интерактивную сессию и управлять всеми процессами с помощью простых команд. Вот пример такой сессии:

```
$ circusctl
circusctl 0.13.0
webxample: active
(circusctl) stop webxample
ok
(circusctl) status
webxample: stopped
(circusctl) start webxample
ok
(circusctl) status
webxample: active
```

Конечно, у обоих упомянутых инструментов намного больше возможностей. Все они перечислены в их документации, поэтому, прежде чем сделать свой выбор, вы должны внимательно изучить ее.

В следующем подразделе рассматривается важность запуска кода приложения в пространстве пользователя.

Запуск кода приложения в пространстве пользователя

Код приложения всегда должен работать в пространстве пользователя. Это значит, что он не должен выполняться с правами суперпользователя. Если вы разрабатываете приложение с соблюдением правил «Двенадцатифакторного приложения», то можно запустить приложение под пользователем, не имеющим почти никаких привилегий. Общепринятое имя для пользователя, который не владеет файлами и не находится ни в одной из привилегированных групп, — `nobody`. Однако мы рекомендуем создавать отдельного пользователя для каждого демона. Это обусловлено системой безопасности и делается с целью ограничить ущерб, который может причинить злоумышленник, если получает контроль над процессом. В Linux процессы одного и того же пользователя могут взаимодействовать, поэтому важно, чтобы различные приложения разделялись на уровне пользователя.

В следующем разделе показано, как задействовать обратный HTTP-прокси.

Использование обратного HTTP-прокси

Несколько WSGI-совместимых серверов могут легко обслуживать HTTP-трафик самостоятельно, не прибегая к необходимости иметь какой-либо другой веб-сервер на верхнем уровне. Все еще часто многие прячут их за обратным прокси-сервером наподобие NGINX или Apache. Обратный прокси-сервер создает дополнительный слой HTTP-сервера, который перенаправляет запросы и ответы между клиентами и приложением и появляется на сервере Python так, словно это запрашивающий клиент. Обратные прокси-серверы можно использовать по следующим различным причинам.

- Разрыв TLS/SSL, как правило, лучше обрабатывается веб-серверами верхнего уровня, например NGINX и Apache. Это позволяет приложению Python говорить только на языке HTTP-протокола (вместо HTTPS), вследствие чего вопросы сложности и конфигурации защищенных каналов связи ложатся на обратный прокси-сервер.
- Непривилегированные пользователи не могут связывать порты нижнего уровня (в диапазоне 0–1000), но протокол HTTP идет к пользователям на порт 80, а HTTPS — на порт 443. Для этого необходимо запустить процесс с привилегиями суперпользователя. Как правило, более безопасно сделать так, чтобы ваше приложение работало на порте верхнего уровня или на сокете домена Unix и использовало его в качестве восходящего потока для обратного прокси-сервера, который выполняется под более привилегированным пользователем.
- Обычно NGINX работает со статическими ассетами (изображения, JS, CSS и другие мультимедиа) эффективнее, чем код Python. Если настроить его в качестве обратного прокси-сервера, то потребуется всего несколько строк конфигурации для работы со статическими файлами.
- Когда один хост обслуживает несколько приложений от различных доменов, для создания виртуальных хостов для различных доменов, обслуживаемых на одном порте, нужно использовать Apache или NGINX.
- Обратные прокси-серверы могут повысить производительность за счет добавления дополнительных слоев кэширования или быть сконфигурированными как простые балансировщики нагрузки. Кроме того, на обратных прокси-серверах может применяться сжатие (например, GZIP) для ответов с целью ограничить количество требуемой пропускной способности сети.

Некоторые из веб-серверов, такие как NGINX, на самом деле даже рекомендуется запускать за прокси-сервером. Например, `gunicorn` — это очень надежный WSGI-сервер, показывающий великолепные результаты, если его клиенты функционируют достаточно быстро. С другой стороны, он плохо работает с медленными клиентами и легко восприимчив к атакам, основанным на медленном соединении

с клиентом. Использование прокси-сервера для буферизации медленных клиентов — лучший способ решить данную проблему.

Помните: имея соответствующую инфраструктуру, можно почти полностью избавиться от обратного прокси. Сегодня такие проблемы, как разрыв SSL и сжатие, можно легко решить с помощью служб балансировки нагрузки сервисов, например, AWS Load Balancer. Статические и мультимедийные ассеты лучше подавать через *сеть доставки контента* (Content Delivery Networks, CDN), которые также могут быть использованы для кэширования других откликов вашего сервиса.

Упомянутое выше требование передавать HTTP/HTTPS-трафик на порты 80/433 (которые не могут быть связаны непривилегированными пользователями) тоже больше не проблема, если точки входа ваших клиентов общаются с вашими балансировщиками нагрузки и CDN. Тем не менее даже наличие такой архитектуры не обязательно означает, что ваша система не работает с обратными прокси-серверами. Например, многие балансировщики нагрузки поддерживают протокол прокси. Это значит, балансировщик нагрузки может появиться в приложении, притворившись запрашивающим клиентом. В подобных случаях балансировщик нагрузки действует как обратный прокси-сервер.

В следующем подразделе поговорим о перезагрузке процессов.

Корректная перезагрузка процессов

Девятое правило методологии «Двенадцатифакторного приложения» касается одноразовости процессов и говорит о том, что вы должны максимизировать надежность через быстрый запуск и корректную остановку. С быстрым запуском все понятно, а вот о корректной остановке нужно поговорить отдельно.

Если в веб-приложении процесс сервера завершается ненадлежащим образом, то просто возьмет и закроется, не закончив обработку запросов и выдачу правильных ответов подключенным клиентам. В лучшем случае при использовании какого-либо обратного прокси-сервера он сможет ответить подключенным клиентам с некоторой реакцией на ошибку (например, 502 Bad Gateway), даже если это не самый удачный способ уведомить пользователей о перезагрузке приложения и развертывании новой версии.

В соответствии с «Двенадцатифакторным приложением» процесс веб-сервера должен корректно завершаться при получении сигнала Unix `SIGTERM`. Это значит, что сервер должен прекратить создание новых подключений, завершить обработку всех ожидающих запросов, а затем выйти с неким кодом, как только завершит текущие задачи.

Очевидно, при запуске процедуры отключения сервер больше не может обрабатывать новые запросы. Это означает нарушение в работе сервиса. Таким образом, вам нужно сделать еще кое-что: создать новых исполнителей, которые смогли бы

принимать новые соединения, пока старые отключаются. Существуют WSGI-совместимые реализации веб-сервера Python, позволяющие должным образом перезагрузить сервис, не прибегая к каким-либо простоям.

Самые популярные веб-серверы Python — это Gunicorn и uWSGI, имеющие такие функции:

- мастер-процесс Gunicorn при получении сигнала `SIGHUP` (`kill -HUP <process-pid>`) создает новых исполнителей (с новым кодом и конфигурацией) и корректно завершает работу на старых;
- uWSGI имеет по меньшей мере три независимых схемы корректной перезагрузки. Все они сложные, и в двух словах их не описать, но полная информация о них есть в официальной документации.

На сегодняшний день корректная перезагрузка — стандарт развертывания веб-приложений. Подход Gunicorn выглядит самым простым в использовании, но в то же время обладает наименьшей гибкостью. В противовес этому корректная перезагрузка в uWSGI позволяет значительно лучше управлять процессом перезагрузки, но более сложна для автоматизации и настройки. Кроме того, применяемый подход к перезагрузке в автоматизированном развертывании зависит и от того, какие инструменты контроля используются и как настроены. Например, в Gunicorn все просто:

```
kill -HUP <gunicorn-master-process-pid>
```

Но если вы хотите правильно изолировать дистрибутивы проектов путем создания отдельных виртуальных окружений для каждой версии и настроить контроль за процессом с помощью символических ссылок (как это представлено в примере `fabfile`), то вскоре заметите, что данная функция Gunicorn работает не вполне ожидаемым образом. При более сложном развертывании до сих пор не существует решения на уровне систем, работающего сразу «из коробки». Вам всегда придется залезать внутрь, а для этого иногда требуется иметь немало знаний о низкоуровневых моментах реализации системы.

В таких сложных скриптах, как правило, лучше решать проблему на более высоком уровне абстракции. Если вы наконец решили запускать приложения в качестве контейнеров и распространяете новые версии в виде образов контейнеров (что настоятельно рекомендуется), то можете передать ответственность за корректную перезагрузку вашей системе оркестровки контейнера (например, Kubernetes), которая, как правило, позволяет обрабатывать различные стратегии перезагрузки «из коробки».

Даже не имея современных систем оркестровки, вы можете реализовать перезагрузку на уровне инфраструктуры. Например, AWS Elastic Load Balancer позволяет корректно перенаправлять трафик от старых экземпляров приложения (например,

хостов EC2) на новые. Когда старые экземпляры приложения перестают получать трафик и обрабатывать запросы, их можно просто отключить без видимых перебоев сервиса. У других облачных провайдеров тоже, как правило, есть аналогичные сервисы в портфеле услуг.

В следующем разделе поговорим о контрольно-проверочном коде и мониторинге.

Контрольно-проверочный код и мониторинг

Наша работа не заканчивается на написании приложений и их развертывании в целевой среде выполнения. Можно написать приложение, которому после развертывания не нужно будет дальнейшее сопровождение, хотя это весьма маловероятно. Вместо этого мы должны убедиться, что приложение правильно контролируется на предмет наличия ошибок и производительности.

Чтобы быть уверенными в должной и ожидаемой работе продукта, вам следует правильно обрабатывать журналы приложений и контролировать необходимые показатели. В эту задачу входит следующее:

- мониторинг журналов доступа веб-приложений для различных кодов состояния HTTP;
- сбор журналов процессов, которые могут содержать информацию об ошибках во время выполнения и различные предупреждения;
- использование системных ресурсов (загрузка процессора, задействование памяти, сетевой трафик, производительность ввода/вывода, использование диска и т. д.) на удаленных хостах, на которых запускается приложение;
- мониторинг производительности на уровне приложений и показателей эффективности бизнеса (привлечение клиентов, доход, коэффициент конверсии и т. д.).

К счастью, для выполнения этой задачи существует много инструментов, и большинство из них очень легко интегрировать.

В следующем подразделе поговорим об ошибках журнала и Sentry/Raven.

Ошибки журнала — Sentry/Raven

Истина ужасна. Независимо от того, насколько хорошо будет протестировано ваше приложение, в какой-то момент ваш код все равно даст сбой. Это может быть что угодно: неожиданное исключение, истощение ресурсов, сбой некоего основного сервиса или сети либо просто проблема во внешней библиотеке. Некоторые из возможных проблем (например, истощение ресурсов) можно предсказать и предот-

вратить заранее при наличии надлежащего контроля. Но, к сожалению, проблемы все равно возникают всегда, независимо от ваших усилий.

Но зато вы можете подготовиться к таким сценариям и убедиться, что ошибка не останется незамеченной. В большинстве случаев любые неожиданные результаты приводят к выбрасыванию исключения, которое попадает в журнал. Это может быть `stdout`, `stderr`, файл журнала или любой другой настроенный вывод. В зависимости от реализации, это может привести или не привести к закрытию приложения с неким кодом выхода.

Можно, конечно, в целях обнаружения и мониторинга ошибок приложения положиться исключительно на файлы журналов, хранящиеся в файловой системе. К сожалению, поиск ошибки в простом текстовом виде мучительно труден и не масштабируется за пределы чего-либо более сложного, чем выполнение кода в процессе разработки. Все равно вам придется использовать специальные сервисы, предназначенные для сбора и анализа журналов. Правильная обработка журналов очень важна и по другим причинам (об этом чуть позже), однако не слишком хорошо работает для отслеживания и отладки ошибок. Причина проста. Наиболее распространенная форма журналов ошибок — просто стек вызовов Python. Если вы останавливаетесь лишь на этом, то вскоре поймете, что таких действий недостаточно для обнаружения причины ваших проблем. Это особенно верно, когда ошибки возникают в неизвестных ситуациях или в определенных условиях нагрузки.

Что вам действительно нужно — это как можно больше контекстной информации о возникновении ошибки. Кроме того, очень полезно иметь полную историю ошибок, которые возникали в production-среде, и удобный способ их поиска.

Один из наиболее распространенных инструментов, который дает такие возможности, — Sentry (getsentry.com). Это проверенный сервис для отслеживания исключений и сбора отчетов об ошибках. Он доступен в виде открытого исходного кода, написан на Python и изначально возник как инструмент для бэкенд-веб-разработчиков. Сейчас он перерос изначальные амбиции и имеет поддержку многих других языков, в том числе PHP, Ruby и JavaScript, но по-прежнему остается самым популярным инструментом выбора для многих веб-разработчиков на Python.

Исключение стека вызовов в веб-приложениях

Обычно веб-приложения не завершаются в случае необработанных исключений, поскольку HTTP-серверы обязаны вернуть ответ об ошибке с кодом состояния из группы 5XX, если произошла ошибка сервера. В большинстве веб-фреймворков Python такие вещи реализованы по умолчанию. В таких случаях исключение, по сути, обрабатывается либо на внутреннем уровне веб-фреймворка, либо на промежуточном WSGI-сервере. Во всяком случае, это, как правило, приводит к выбрасыванию исключения стека вызовов (обычно в стандартном выводе).

Sentry доступен в качестве платного «ПО как услуга», но с открытым исходным кодом, вследствие чего его можно бесплатно разместить в вашей собственной инфраструктуре. Библиотека, которая обеспечивает интеграцию с Sentry, — `sentry-sdk` (доступна на PyPI). Если вы еще не работали с ней и хотите попробовать, но у вас нет доступа к собственному серверу Sentry, то можете легко подписаться на бесплатную пробную версию на сайте Sentry. Имея доступ к серверу Sentry и созданный новый проект, вы получите строку с именем *Data Source Name (DSN)*. Эта строка — минимальная настройка конфигурации, необходимая для интеграции приложения с Sentry. Она содержит протокол, учетные данные, местоположение сервера и ваш идентификатор организации/проекта в следующем виде:

```
'{PROTOCOL}://{PUBLIC_KEY}:{SECRET_KEY}@{HOST}/{PATH}{PROJECT_ID}'
```

Получив DSN, вы можете легко выполнить интеграцию, как показано в следующем коде:

```
import sentry_sdk

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>'
)

try:
    1 / 0
except Exception as e:
    sentry_sdk.capture_exception(e)
```

Sentry и Raven

Старая библиотека для интеграции Sentry — это Raven. Она по-прежнему поддерживается и доступна на PyPI, но ее век уже подходит к концу, вследствие чего лучше всего начать интеграцию Sentry с помощью пакета python-sdk. Однако вполне возможно, что некоторые фреймворки или расширения Raven не были портируемы на новый SDK, поэтому в таких ситуациях интеграция с помощью Raven пока более предпочтительна.

Sentry SDK имеет множество интеграций с большинством популярных структур, таких как Python Django, Flask, Celery или Pyramid, упрощающих интеграцию. Эти интеграции автоматически дают дополнительный контекст, специфичный для данной структуры. Если у веб-фреймворка нет поддержки, то пакет `sentry-sdk` содержит общее промежуточное ПО WSGI, которое делает его совместимым с любыми WSGI-серверами, как показано в следующем коде:

```
from sentry_sdk.integrations.wsgi import SentryWsgiMiddleware

sentry_sdk.init(
```

```
    dsn='https://<key>:<secret>@app.getsentry.com/<project>'
)

# ...

# Примечание: application — объект приложения WSGI, определенный ранее
application = SentryWsgiMiddleware(application)
```

Другая примечательная интеграция — возможность отслеживать сообщения, регистрируемые через встроенный модуль Python `logging`. Для включения такой поддержки требуется всего несколько дополнительных 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],
)
```

Захват сообщений журнала может иметь свои сложности, вследствие чего вам нужно изучить официальную документацию по данной теме. Это должно избавить вас от неприятных сюрпризов.

Наконец, скажем о запуске собственного Sentry в качестве способа сэкономить деньги. Помните: *бесплатный сыр только в мышеловке*. Вы все равно будете нести дополнительные расходы на инфраструктуру, а Sentry станет просто еще одним сервисом, требующим сопровождения. *Сопровождение = дополнительная работа = = затраты!* По мере развития вашего приложения количество исключений растет, следовательно, вы будете вынуждены масштабировать Sentry вместе со своим продуктом. К счастью, это очень надежный проект, но вам от него никакой пользы в случае слишком высокой нагрузки. Кроме того, сложно держать Sentry в готовности к сценарию отказа, в котором могут возникать тысячи отчетов о сбоях в секунду. Таким образом, вы должны решить, какой вариант для вас действительно дешевле и имеете ли вы достаточно ресурсов, чтобы сделать все это самостоятельно. Подобной дилеммы, конечно, не возникает, если политика безопасности вашей организации не подразумевает отправку каких-либо данных третьим лицам. В этом случае вы можете просто разместить Sentry в вашей собственной инфраструктуре. Расходы, конечно, есть, но они того стоят.

Далее мы рассмотрим метрики систем мониторинга и приложений.

Метрики систем мониторинга и приложений

Когда дело доходит до мониторинга производительности, вашему вниманию предлагается огромный выбор инструментов. Если у вас большие ожидания, то вполне возможно, что вам придется использовать сразу несколько из них.

Munin (munin-monitoring.org) — один из самых популярных вариантов, применяемых многими организациями, независимо от стека технологий. Это отличный инструмент для анализа тенденций в ресурсах, который предоставляет много полезной информации даже при установке по умолчанию, без дополнительных настроек. Его установка состоит из следующих двух основных компонентов:

- мастера Munin, собирающего метрику от других узлов и выводящего графики;
- узла Munin, который устанавливается на управляемом хосте и собирает локальные метрики и посылает их мастеру Munin.

Большинство плагинов написаны на Perl. Есть также реализация узла на других языках: `munin-node-c` на C (github.com/munin-monitoring/munin-c) и `munin-node-python` на Python (github.com/agroszer/munin-node-python). Munin поставляется с огромным количеством плагинов, доступных в репозитории `contrib`. Это значит, что в нем «из коробки» поддерживается большинство популярных баз данных и системных сервисов. Есть даже плагин для мониторинга популярных веб-серверов Python, таких как uWSGI или Gunicorn.

Основной недостаток Munin заключается в том, что он выводит графики как статические изображения, а конфигурация графика возложена на настройку плагина. Вы не сможете создавать гибкие панели мониторинга и сравнивать значения метрики из разных источников на одном графике. Но это цена простой установки и универсальности. Писать собственные плагины достаточно легко. Существует пакет `munin-python` (python-munin.readthedocs.org/en/latest/), который позволяет писать плагины Munin на Python.

К сожалению, архитектура Munin, предполагающая, что всегда существует отдельный демон-процесс мониторинга на каждом хосте, который отвечает за сбор метрик, не слишком хорошо подходит для мониторинга показателей производительности пользовательских приложений. Вам будет очень легко написать свой собственный плагин Munin, но при условии, что процесс контроля уже позволяет вам получать статистические данные о производительности.

Если вы хотите собрать метрики на уровне приложений, то вам может понадобиться их сбор и хранение в каком-то временном хранилище до представления в пользовательском плагине Munin. Это делает создание пользовательских метрик более сложным, вследствие чего стоит рассмотреть другие решения.

Другое популярное решение, с помощью которого легко собирать пользовательские метрики, — StatsD (github.com/etsy/statsd). Это сетевой демон, написанный на

Node.js, который собирает различные статистические данные со счетчиков, таймеров и датчиков. Его очень легко интегрировать благодаря простому протоколу, основанному на UDP. Кроме того, существует простой в применении пакет Python `statsd` для отправки метрик в демон StatsD:

```
>>> import statsd
>>> c = statsd.StatsClient('localhost', 8125)
>>> c.incr('foo') # Увеличиваем счетчик 'foo'.
>>> c.timing('stats.timed', 320) # Record a 320ms 'stats.timed'.
```

Поскольку UDP — протокол без установления соединения, он слабо влияет на производительность кода приложения и поэтому очень подходит для отслеживания и измерения пользовательских событий внутри кода приложения.

К сожалению, StatsD занимается только сбором метрик и не позволяет создавать отчетность. Поэтому вам понадобятся другие процессы, способные обрабатывать данные из StatsD и строить графики. Самый популярный выбор — Graphite (graphite.readthedocs.org). Он делает следующее:

- хранит данные времени в числовом формате;
- отрисовывает графики под эти данные.

Graphite позволяет сохранять пресеты с высокими возможностями к настройке. Вы также можете сгруппировать множество графиков в тематические панели. Графики, как и Munin, визуализируются как статические изображения, но есть также JSON API, которая позволит другим оболочкам считывать данные графика и выводить их другими средствами.

Один из самых крутых плагинов приборной панели, интегрированных с Graphite, — это Grafana (grafana.org). Его действительно стоит попробовать, поскольку он гораздо удобнее в использовании, чем простые панели Graphite. Графики, представляемые в Grafana, полностью интерактивны и проще в управлении.

Проект Graphite, к сожалению, немного сложноват. Это модульный сервис, который состоит из следующих трех отдельных компонентов:

- *Carbon* — демон, написанный с помощью Twisted, ожидающий данные временных рядов;
- *whisper* — простая библиотека баз данных для хранения данных временных рядов;
- *graphite WebApp* — веб-приложение Django, которое выводит графики как статические изображения (с помощью библиотеки Cairo) или в виде данных в формате JSON.

При использовании совместно с проектом StatsD демон `statsd` отправляет свои данные на демон `carbon`. В результате полноценное решение превращается в довольно большую стопку приложений, каждое из которых написано по отдельной

технологии. Кроме того, в нем нет настраиваемых графиков, плагинов и инструментальных панелей, следовательно, вам предстоит настроить все самостоятельно. Вам придется сделать много работы в начале и очень легко пропустить что-то важное. Поэтому хорошей идеей может быть использование Munin в качестве резервного мониторинга, даже если в качестве основного средства применяется Graphite.

Другое хорошее решение мониторинга для сбора метрики — Prometheus. Он имеет архитектуру, совершенно отличную от Munin и StatsD. Вместо того чтобы полагаться на контролируемые приложения или демоны в целях выдачи метрики в настраиваемых интервалах, Prometheus вытягивает метрику непосредственно из источника с помощью протокола HTTP. Для этого требуется контролируемый сервис для хранения (а иногда и обработки) метрик и их выдачи на конечные точки.

К счастью, Prometheus поставляется с кучей библиотек для разных языков и фреймворков, позволяющих сделать данный вид интеграции максимально простым. Существуют также различные экспортеры, которые действуют как мосты между другими системами мониторинга Prometheus. Поэтому если вы уже используете другие решения для мониторинга, то вам будет легко перейти на Prometheus. Он также прекрасно интегрируется с Graphana.

В следующем подразделе мы увидим, как работать с журнальными приложениями.

Работа с журнальными приложениями

Хотя такие решения, как Sentry, обычно намного более эффективны, чем простой вывод в текстовый файл, журналы — наше все. Вывод информации в стандартный вывод или файл — одна из самых простых вещей, на которые способно приложение, и это свойство никогда не следует недооценивать. Существует риск того, что сообщения, отправляемые к Sentry от Raven, не дойдут до получателя. Вероятны ошибки сети, у Sentry может закончиться память, или он не сможет обрабатывать поступающий трафик. Кроме того, ваше приложение может сломаться до отправки сообщения (с ошибкой сегментации, например) — и все это лишь некоторые из потенциальных сценариев.

Менее вероятно, что ваше приложение не сможет зарегистрировать сообщения, записываемые и сохраняемые в файловой системе. Конечно, все возможно, но скажем начистоту: если до такого дойдет, то это значит, что у вас начались более серьезные проблемы, чем утеря пары-тройки сообщений журнала.

Помните, что журналы нужны не только для ошибок. Многие разработчики привыкли считать журналы лишь источником полезных для отладки данных или данных для анализа проблемы.

Гораздо меньшее количество разработчиков пробует задействовать журналы в качестве источника данных для генерации метрик приложений или выполнения

статистического анализа. Но и на этом их польза не заканчивается. Журналы даже могут быть основой выпускаемого продукта. Отличный пример такого продукта приведен в статье Amazon, в которой показана архитектура для обслуживания торгов в режиме реального времени, где все сосредоточено вокруг сбора и обработки информации в журнале: aws.amazon.com/blogs/aws/real-time-ad-impression-bids-using-dynamodb.

Рассмотрим основные практики работы с журналами низкого уровня.

Низкоуровневые методы работы с журналами

Правила «Двенадцатифакторного приложения» гласят: журналы должны рассматриваться как потоки событий. Таким образом, файл журнала — это не журнал как таковой, а формат вывода. Это значит, что в журнале описаны упорядоченные по времени события. В сыром виде они имеют вид обычного текста, где одна строка соответствует одному событию, хотя в некоторых случаях событие может захватывать несколько строк (это характерно для ошибок времени выполнения).

В соответствии с методологией «Двенадцатифакторного приложения» приложение никогда не должно знать о формате, в котором хранятся журналы. Это значит, запись в файл или ротацию журналов никогда не следует выполнять в коде приложения.

Это является обязанностью среды, в которой выполняется приложение. Данный факт немного сбивает с толку, поскольку во многих фреймворках есть функции и классы для управления файлами журналов, их ротации, сжатия и сохранения. Возникает искушение использовать их, поскольку тогда все можно разместить в коде приложения базы, однако на самом деле такого подхода следует избегать.

Лучшие методы для работы с журналами заключаются в следующем:

- приложение должно записывать небуферизованные журналы в стандартный вывод (`stdout`);
- среда выполнения отвечает за сбор и маршрутизацию журналов до конечной точки.

Основная часть этой среды выполнения — как правило, своего рода инструмент для контролирования состояния процессов. Популярные решения Python, такие как Supervisor или Circus, позволяют работать с журналами и маршрутизацией. Если журналы нужно хранить в локальной файловой системе, то лишь они должны создавать сами файлы журналов.

И Supervisor, и Circus также способны выполнять ротацию журналов и сохранение для необходимых вам процессов, но следует определить, нужен ли вообще вам этот путь. Успех работы заключается в простоте и последовательности. Вполне возможно, что вам потребуется обрабатывать не только журналы вашего собственного приложения. Если вы используете Apache или NGINX в качестве обратного прокси-сервера, то их журналы вам тоже понадобятся.

Вы также можете хранить и обрабатывать журналы кэша и баз данных. Если вы работаете в каком-то популярном дистрибутиве Linux, то велика вероятность того,

что у каждого из этих сервисов будут свои собственные файлы журналов, обрабатываемые с помощью популярной утилиты под названием `logrotate`. Мы рекомендуем забыть о возможностях Supervisor и Circus ради достижения согласованности с другими сервисами. Утилита `logrotate` более гибкая в настройке и к тому же поддерживает сжатие.

Logrotate и Supervisor/Circus

Есть важная вещь, которую следует знать при использовании logrotate с Supervisor или Circus. Ротация журналов будет происходить всегда, а у Supervisor есть открытый дескриптор ротации журналов. Если вы не станете принимать надлежащие контрмеры, то новые события будут записываться в дескриптор, который уже был удален logrotate. В результате в файловой системе не останется вообще ничего. Решить эту проблему достаточно просто. Нужно настроить logrotate на файлы журналов процессов, управляемых Supervisor или Circus, с помощью опции copytruncate. Вместо перемещения файла журнала после ротации он будет копироваться, а исходный файл станет очищаться. Такой подход не отменяет существующие дескрипторы файлов и процессов, и запись в журналы будет идти своим ходом. Supervisor может также принимать сигнал SIGUSR2, который заново открывает все дескрипторы файлов. Он может быть включен в скрипт postrotate в настройке logrotate. Этот подход более экономичен с точки зрения операций ввода/вывода, но в то же время менее надежен и более труден в сопровождении.

Инструменты для обработки журналов описаны ниже.

Инструменты для обработки журналов

Если у вас нет опыта взаимодействия с большими объемами журналов, то вы рано или поздно получите его, работая с продуктом, обрабатывающим существенную нагрузку. Вы вскоре заметите, что простого подхода, основанного на хранении журналов в файлах и в каком-либо постоянном хранилище для последующего извлечения, будет недостаточно. Отсутствие соответствующих инструментов превратит эту работу в сложную и топорную. Простые утилиты, такие как `logrotate`, помогут вам освободить жесткий диск от постоянно увеличивающегося количества новых событий, хотя разделение и сжатие файлов журналов помогает только в работе с архивами данных, однако не упрощает их извлечение или анализ.

При работе с системами, распределенными по нескольким узлам, хорошо иметь одну центральную точку, из которой все журналы можно извлечь и затем проанализировать. Для этого требуется процедура обработки журнала, выходящая далеко за рамки простого сжатия и резервного копирования. К счастью, данная задача хорошо известна и есть много инструментов, позволяющих ее решить.

Один из популярных вариантов среди разработчиков — *Logstash*. Это демон сбора журнала, который может наблюдать за активными файлами журналов, анализировать записи и отправлять их в бэкенд-сервис в структурированной форме. В качестве сервиса почти всегда используется *Elasticsearch*, поисковая система, построенная вокруг Lucene. Помимо возможностей поиска текста, она содержит уникальный фреймворк агрегации данных, который очень хорошо подходит для целей анализа журнала. Другое дополнение к этой паре инструментов — *Kibana*. Это универсальная платформа для мониторинга, анализа и визуализации для Elasticsearch. Таким образом, эти три инструмента дополняют друг друга и потому почти всегда используются совместно, как единый стек для обработки журналов.

Интеграция существующих сервисов с Logstash очень проста, поскольку он может отслеживать изменения файла журнала на предмет событий при минимальных изменениях в конфигурации журнала. Он анализирует журналы в текстовом виде и имеет встроенную поддержку ряда популярных форматов журналов, таких как журналы доступа Apache/NGINX. Logstash может быть дополнена Beats. Это поставщик журнала, совместимый с входными протоколами Logstash, который может собирать не только необработанные данные из журналов файлов (filebeat), но и метрики различных систем (metricbeat) и даже действия пользователей аудита на хостах (auditbeat).

Другое решение, позволяющее устранить недостатки Logstash, — Fluentd. Это еще один демон для сбора журналов, который можно взаимозаменяемо применять с Logstash в упомянутом стеке мониторинга журнала. Он также позволяет слушать и анализировать события журнала непосредственно в файлах журналов, поэтому интеграцию настроить несложно. В отличие от Logstash он очень хорошо обрабатывает случаи перезагрузки и не нуждается в сигналах о ротации файлов журналов. Наибольшее преимущество достигается путем использования одного из его альтернативных вариантов сбора журнала, для которых потребуются кое-какие существенные изменения в настройке ведения журналов.

Fluentd действительно обрабатывает журналы в виде потоков событий (в соответствии с рекомендациями «Двенадцатифакторного приложения»). Интеграция на основе файлов по-прежнему возможна, но это единственный вид обратной совместимости для старых приложений, в котором журналы рассматриваются в виде файлов. Каждая запись в журнале — событие, и оно должно быть структурировано. Fluentd позволяет разбирать текстовые журналы и включает несколько вариантов плагинов, в том числе следующие:

- общие форматы (Apache, NGINX, и syslog);
- произвольные форматы, которые задаются с помощью регулярных выражений или обрабатываются пользовательским плагином;
- общие форматы для структурированных сообщений, таких как JSON.

Лучший формат событий для Fluentd — это JSON, поскольку позволяет достичь наименьшего количества затрат. Сообщения в формате JSON также могут быть переданы практически без каких-либо изменений в бэкенд-сервисе, таком как Elasticsearch, или в базе данных.

Другая очень полезная особенность Fluentd — способность пропускать потоки событий способами, отличными от тех, которыми записываются журналы на диск. Ниже приведены наиболее примечательные встроенные входные плагины и то, какие действия возможно выполнить с их помощью:

- `in_udp` — каждый журнал событий передается в виде пакетов UDP;
- `in_tcp` — события посылаются через соединение TCP;
- `in_unix` — события посылаются через сокет домена Unix (именованный сокет);
- `in_http` — события передаются в виде запросов HTTP POST;
- `in_exec` — процесс Fluentd периодически выполняет внешнюю команду, вытаскивая события в формате JSON или MessagePack;
- `in_tail` — процесс Fluentd прослушивает события в текстовом файле.

Альтернативные способы передачи журнала событий могут быть особенно полезны в ситуациях, когда приходится работать с медленным вводом/выводом памяти компьютера. Это часто применяется в сервисах облачных вычислений, у которых на диске-хранилище по умолчанию очень мало операций ввода-вывода в секунду (input output operations per second, IOPS), а получить большую производительность выходит дорого.

Если приложение выдает большое количество сообщений журнала, то вы можете легко достичь предела возможностей ввода-вывода, даже если размер данных не очень велик. Наличие альтернативного канала передачи позволит вам более эффективно использовать оборудование, поскольку работа по буферизации выполняется одним процессом сбора журнала. Настроив буферизацию сообщений в память вместо диска, вы можете полностью избавиться от диска как такового, хотя это может значительно снизить качество собранных журналов.

Использование различных каналов передачи, на первый взгляд, слегка противоречит 11-му правилу методологии «Двенадцатифакторного приложения». Обработка журналов как потоков событий говорит о том, что приложение всегда должно делать записи через один стандартный поток вывода (`stdout`). Вы по-прежнему можете использовать альтернативные способы передачи, не нарушая данное правило. Запись в стандартный вывод не обязательно означает, что этот поток должен быть записан именно в файл.

Вы можете оставить ведение журнала в таком виде и обернуть его внешним процессом, который будет захватывать данный поток и передавать его непосредственно в Logstash или Fluentd, не задействуя файловую систему. Это сложная

модель, которая может подойти не для каждого проекта. Она имеет очевидный недостаток в виде повышенной сложности, так что вы должны решить для себя, стоит ли оно того.

Резюме

Развертывание кода — сложная тема, и вы наверняка уже поняли это после прочтения текущей главы. Несмотря на то что мы ограничили сферу нашей деятельности исключительно веб-приложениями, затронута лишь верхушка айсберга. Мы использовали методику «Двенадцатифакторного приложения» как основу для демонстрации возможных решений различных проблем, связанных с развертыванием кода. Мы подробно обсудили только часть из них: работу с журналами, управление зависимостями и разделение этапов сборки и выполнения.

После прочтения данной главы вы наверняка начали понимать, как выполнить автоматизацию процесса развертывания с учетом передового опыта в этой области, а также иметь возможность добавить надлежащий инструментарий и средства мониторинга для кода, который выполняется на удаленных хостах.

В следующей главе мы изучим, почему писать расширения на C и C++ для Python иногда весьма полезно, и покажем, что это не так сложно, как кажется, если выбрать для работы подходящие инструменты.

9 Расширения Python на других языках

Занимаясь написанием приложений на Python, вы не обязаны ограничиваться одним лишь языком Python. Существуют такие инструменты, как *Hy* (о них мы кратко говорили в главе 5), позволяющие писать модули, пакеты и даже целые приложения на другом языке (диалекте Lisp), который станет работать на виртуальной машине Python. Это дает вам возможность выразить логику программы с совершенно другим синтаксисом, однако во время компиляции в байт-код это будет один и тот же язык, имеющий те же ограничения, что и у обычного кода на Python. Перечислим некоторые из таких ограничений:

- многопоточность значительно снижается из-за *глобальной фиксации интерпретатора* (global interpreter lock, GIL) в CPython и зависит от выбранной реализации Python;
- Python — некомпилируемый язык, поэтому во время компиляции нет оптимизации;
- Python не имеет статической типизации и возможных связанных с ней оптимизаций.

Преодолеть такие ограничения можно с помощью расширений Python, которые полностью написаны на другом языке, но пропускают свой интерфейс через API для расширения Python.

В текущей главе мы рассмотрим основные причины для написания собственных расширений на других языках и познакомимся с популярными инструментами, позволяющими их создавать.

В этой главе:

- различия между языками C и C++;
- создание простого расширения на C с использованием Python/C API;
- создание простого расширения на C с помощью Cython;
- понимание основных проблем и задач, связанных с использованием расширений;
- взаимодействие с компилируемыми динамическими библиотеками без создания выделенных расширений на чистом Python.

Технические требования

Для компиляции расширений Python, о которых мы поговорим в этой главе, нам понадобятся компиляторы C и C++. Ниже приведены подходящие компиляторы, которые можно бесплатно скачать для нужной операционной системы:

- Visual Studio 2019 (Windows): visualstudio.microsoft.com;
- GCC (Linux и большинство систем POSIX): gcc.gnu.org;
- Clang (Linux и большинство систем POSIX): clang.llvm.org.

В Linux компиляторы GCC и Clang, как правило, можно скачать через систему управления пакетами для данного конкретного дистрибутива. В macOS компилятор является частью Xcode IDE (доступна через App Store).

Пакеты Python, упомянутые в этой главе, можно скачать с PyPI:

- `Cython`;
- `cffi`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы доступны по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter9.

Различия между языками C и C++

Когда речь идет об интеграции других языков с Python, это почти всегда C и C++. Даже такие инструменты, как Cython или Pyrex, которые определяют суперсеты языка Python только в целях создания расширений Python, на самом деле являются компиляторами «код — код», генерирующими код C из расширенного синтаксиса Python.

Фактически вы можете использовать динамические/разделяемые библиотеки Python, написанные на любом языке, если он поддерживает компиляцию в виде динамических/разделяемых библиотек. Таким образом, возможности межъязыковой интеграции выходят далеко за пределы C и C++. Это связано с тем, что библиотеки задействуются повсеместно и могут быть применены в любом языке, который поддерживает их загрузку. Итак, даже если вы пишете библиотеку на совершенно другом языке (скажем, Delphi или Prolog), вы можете использовать ее в Python. Тем не менее называть такую библиотеку расширением Python, если в ней не применяется Python/C API, не поворачивается язык.

К сожалению, писать собственные расширения лишь на C или C++, используя голый Python/C API, довольно сложно. Дело не только в том, что это требует хорошего понимания одного из двух языков, изначально трудных в освоении, но и в том,

что здесь потребуется большой объем шаблонной работы. Вам придется писать много повторяющегося кода, нужного исключительно для создания интерфейса, который склеит ваш код C или C++ с интерпретатором Python и его типами данных. Однако вам будет полезно знать, как построены чистые расширения C, ввиду следующих причин:

- вы лучше поймете, как в целом работает Python;
- однажды вам может понадобиться выполнить отладку или сопровождение нативного расширения C/C++;
- это помогает понять, как работают инструменты высокого уровня для создания расширений.

В следующем подразделе поговорим о загрузке расширений на C или C++.

Загрузка расширений на C или C++. Интерпретатор Python способен загружать расширения из динамических/общих библиотек, таких как модули Python, если у них предусмотрен подходящий интерфейс с использованием Python/C API. Данный API должен быть включен в исходный код расширения с помощью файла заголовка `Python.h`, который распространяется вместе с исходниками Python. Во многих дистрибутивах Linux этот файл заголовка содержится в отдельном пакете (например, `python-dev` в Debian/Ubuntu), а под Windows распространяется по умолчанию с интерпретатором. В системах POSIX (например, Linux и macOS) его можно найти в каталоге `include/` в месте установки Python, в операционной системе Windows — в каталоге `Include/` в месте установки Python.

Python/C API традиционно изменяется с каждой новой версией Python. Как правило, эти изменения представляют собой добавление в API новых функций, совместимых с исходниками. Однако в большинстве случаев бинарная совместимость не сохраняется из-за изменений в *бинарном интерфейсе приложения* (application binary interface, ABI). Это значит, что расширения нужно создавать отдельно для каждой версии Python. Кроме того, различные операционные системы имеют несовместимые ABI, вследствие чего практически невозможно создать бинарный код для каждой возможной среды. Поэтому большинство расширений Python распространяется в виде исходного кода.

Начиная с версии Python 3.2, было определенно подмножество Python/C API в целях стабильности ABI. И теперь вы можете создавать расширения с использованием этого ограниченного API (со стабильным ABI), следовательно, можно собирать расширения для данной операционной системы только один раз, после чего работать с любой версией Python от 3.2 и выше, не прибегая к перекомпиляции. Стоит заметить, что это ограничивает количество функций API и не решает проблем старых версий Python. Кроме того, не позволяет создать единый бинарный код, который будет работать на различных операционных системах. То есть мы получаем некий компромисс, но цена стабильного ABI кажется немного высокой для такой незначительной выгоды.

Важно знать: Python/C API работает только с реализацией CPython. Были предприняты некоторые усилия с целью организовать поддержку расширений в альтернативных реализациях, таких как PyPI, Jython или IronPython, но нам кажется, что в данный момент для них не существует стабильного и полноценного решения. Единственная альтернативная реализация Python, позволяющая легко работать с расширениями, — Stackless Python, поскольку это всего лишь модифицированная версия CPython.

Расширения C для Python должны компилироваться в общие/динамические библиотеки до импорта, поскольку отсутствует встроенный способ импортировать код C/C++ в Python непосредственно из исходного кода. К счастью, в `distutils` и `setuptools` есть помощники для определения скомпилированных расширений в виде модулей, поэтому компиляция и распространение могут быть выполнены с помощью скрипта `setup.py`, как если бы они были обычными пакетами Python. Ниже приведен пример скрипта `setup.py` из официальной документации, в котором выполняется создание простого пакета с расширением на C:

```
from distutils.core import setup, Extension

module1 = Extension(
    'demo',
    sources=['demo.c']
)

setup(
    name='PackageName',
    version='1.0',
    description='This is a demo package',
    ext_modules=[module1]
)
```

После написания такого кода нужно сделать еще один шаг:

```
python setup.py build
```

Это действие позволяет скомпилировать все ваши расширения, определенные в качестве аргумента `ext_modules`, в соответствии со всеми дополнительными настройками компилятора, предоставленными конструктором `Extension()`. При этом будет использоваться компилятор, определенный по умолчанию для вашей среды. Данный компиляционный шаг не требуется, если пакет распространяется в виде исходного кода. В таком случае вы должны быть уверены в том, что целевая среда имеет все необходимое для компиляции, то есть компилятор, файлы заголовков и дополнительные библиотеки, которые будут связаны с бинарным файлом (если тот нужен расширению). Подробнее об упаковке расширений Python мы поговорим позже, в разделе «Проблемы с использованием расширений» данной главы.

А в следующем разделе мы обсудим, почему мы должны использовать расширения.

Необходимость в использовании расширений

Сложно сказать, когда имеет смысл писать расширения на C/C++. Можно предположить такое правило: *пишите, если у вас нет другого выбора*. Но это очень субъективное мнение, которое оставляет много места для интерпретации выполнимого в Python. Довольно трудно найти нечто, чего нельзя было бы сделать с помощью чистого кода Python. Тем не менее есть определенные задачи, в которых расширения могут быть особенно полезны и дают следующие преимущества:

- игнорирование GIL в потоковой модели CPython;
- повышение производительности в критических секциях кода;
- интеграцию сторонних динамических библиотек;
- интеграцию исходного кода, написанного на других языках;
- создание пользовательских типов данных.

Конечно, для каждой подобной проблемы, как правило, есть жизнеспособное решение на Python. Например, основные ограничения интерпретатора CPython, такие как GIL, довольно легко преодолеваются с помощью другого подхода, скажем, многопроцессорной обработки вместо потоковой модели. Сторонние библиотеки можно интегрировать с модулем `ctypes`. Любой тип данных можно реализовать на Python. Однако нативный подход Python не всегда оказывается оптимальным. Интеграция внешней библиотеки с чистым Python может быть громоздкой и сложной в сопровождении. Реализация пользовательских типов данных может быть неоптимальной и не иметь доступа к управлению памятью низкого уровня. Таким образом, окончательное решение нужно принимать очень осторожно, учитывая множество факторов. Лучше всего начать с чистой реализации Python и прибегать к использованию расширений, только когда нативный подход оказывается недостаточно эффективным.

В следующем подразделе мы попробуем улучшить производительность критических фрагментов кода.

Повышение производительности критических фрагментов кода

Будем честны. Python стал любимцем разработчиков не из-за производительности. Он выполняется не слишком быстро, зато позволяет ускорить разработку программы. Тем не менее, независимо от нашей эффективности как программистов, иногда находятся проблемы, которые не получается эффективно решить с помощью чистого Python.

В большинстве случаев решение проблем с производительностью сводится к выбору правильных алгоритмов и структур данных, а не к ограничению постоянных затрат языка. И как правило, не стоит использовать расширения с целью сэкономить пару тактов процессора, если код уже написан плохо или задействует неэффективные алгоритмы. Часто бывает так, что удается повысить производительность до приемлемого уровня, не прибегая к увеличению сложности вашего проекта, которая вытекает из добавления еще одного языка в стек технологий. Если есть возможность использовать в проекте только один язык программирования — это приоритетная цель. Но также весьма вероятно, что даже *самые передовые* алгоритмы и наиболее подходящие структуры данных не помогут преодолеть технологические ограничения чистого Python.

Пример ситуации, в которой возникают четко определенные ограничения на производительность приложения, — бизнес *ставок в реальном времени* (real-time bidding, RTB). Вкратце, RTB — это покупка и продажа инструментов для рекламы (мест для нее) примерно таким же образом, как на аукционе или фондовой бирже. Вся торговля обычно проходит через некоторые сервисы биржи рекламы, отправляющей информацию о доступных ресурсах на *платформу спроса* (demand-side platform, DSP), заинтересованной в покупке места для рекламы. И вот здесь начинается самое интересное. На большинстве бирж для связи с потенциальными покупателями используется протокол OpenRTB (основанный на HTTP). DSP — сайт, отвечающий за обслуживание ответов на HTTP-запросы OpenRTB. Рекламные биржи всегда ставят очень жесткие ограничения на то, сколько времени может выполняться весь процесс. Оно может иметь довольно малые значения — 50 мс от получения первого пакета TCP до записи последнего байта на сервер DSP. Чтобы ускорить работу, DSP-платформа может обрабатывать десятки тысяч запросов в секунду. Возможность сократить время отклика на несколько миллисекунд часто определяет прибыльность сервиса. Это значит, что перенос даже тривиального кода на C может оказаться в такой ситуации разумным, но только если действительно есть узкое место, в котором не получается добиться улучшений алгоритмически. Как однажды сказал Гвидо, «если вы жаждете скорости... — цикл, написанный на C, будет непобедим».

Об интеграции существующего кода, написанного на разных языках, поговорим в следующем подразделе.

Интеграция существующего кода, написанного на разных языках

Компьютерные науки довольно молоды по сравнению с другими областями техники, однако многие программисты уже написали большое количество полезных библиотек для решения часто возникающих задач на разных языках

программирования. Было бы расточительно забывать обо всем этом наследии всякий раз в момент появления нового языка программирования, но в то же время невозможно надежно портировать все программы, когда-либо написанные на всех языках.

C и C++ — наиболее важные языки, предоставляющие много библиотек и реализаций, которые можно было бы интегрировать в код приложения, не прибегая к полному портированию на Python. К счастью, CPython уже написан на C, поэтому наиболее естественным способом интегрировать такой код будет применение пользовательских расширений.

В следующем подразделе мы объясним, как интегрировать сторонние динамические библиотеки.

Интеграция сторонних динамических библиотек

Интеграция кода, написанного с помощью отличных от Python технологий, не заканчивается на C/C++. Множество библиотек, особенно программное обеспечение сторонних производителей с закрытым кодом, распространяется в виде скомпилированных двоичных файлов. На C загружать такие общие/динамические библиотеки и вызывать их функции весьма легко. Это значит, что вы можете использовать любую библиотеку C, если она обернута расширением с помощью Python/C API.

Это, конечно, не единственное решение, и существуют инструменты, такие как `ctypes` или CFFI, которые позволяют взаимодействовать с динамическими библиотеками с помощью чистого Python, не прибегая к необходимости писать расширения на C. Очень часто выбор Python/C API — все еще лучший вариант, поскольку в этом случае вы получаете лучшее разделение между интеграционным слоем (написанным на C) и остальной частью вашего приложения.

В следующем подразделе расскажем, как создавать пользовательские типы данных.

Создание пользовательских типов данных

В Python есть весьма универсальный выбор встроенных типов данных. Некоторые из них реализованы *по последнему писку моды* (по крайней мере в CPython) и специально созданы для использования на языке Python. Количество базовых типов и коллекций, доступных «из коробки», впечатляюще выглядит для новичков, но все же не охватывает все возможные потребности.

Вы можете создавать на Python пользовательские структуры данных, основывая их полностью на некоторых встроенных типах или создавая их с нуля как совершенно новые классы. К сожалению, в приложениях, чья работа существенно зависит от таких пользовательских структур данных, производительность подобной

структуры может быть неоптимальной. Вся мощь сложных коллекций наподобие `dict` или `set` исходит от лежащей в их основе реализации C. Почему бы не сделать то же самое и не реализовать некоторые из ваших пользовательских структур данных на C?

В следующем разделе мы обсудим, как писать расширения.

Написание расширений

Как уже было сказано, написание расширений — непростая задача, но в результате напряженной работы появится множество преимуществ. Самый простой подход к созданию расширений — использование таких инструментов, как Cython или Pyrex. Эти проекты увеличат вашу продуктивность, а также сделают код более легким для разработки, чтения и сопровождения.

В любом случае если вы новичок в данной области, то лучше всего начать ваше приключение в мире расширений с написания такого расширения на чистом C и Python/C API. Это позволит вам лучше понять, как работают расширения, а также поможет оценить преимущества альтернативных решений. Для простоты возьмем в качестве примера простую алгоритмическую задачу и попытаемся реализовать ее с помощью двух различных подходов, таких как:

- написание расширения на чистом C;
- использование Cython.

Задача: найти *n*-е число последовательности Фибоначчи. Очень маловероятно, что вы захотите писать для такой задачи компилируемые решения, но мы возьмем ее в качестве примера написания функции на C для Python/C API. Наша цель — ясность и простота, поэтому мы не будем пытаться создать наиболее эффективное решение.

Прежде чем создать наше первое расширение, определим эталонную реализацию функции Фибоначчи, которая позволит сравнить различные решения. Она написана на чистом Python и выглядит следующим образом:

```
"""Модуль Python, который обеспечивает функцию последовательности Фибоначчи"""

def fibonacci(n):
    """Возвращает номер n-го элемента Фибоначчи, вычисляемого рекурсивно"""
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Следует отметить, что это одна из самых простых реализаций функции `fibonacci()`, и даже на Python можно было бы ее улучшить. Мы не будем улучшать нашу

реализацию (используя шаблоны запоминания, например), поскольку цель нашего примера не в том. Аналогично мы не будем оптимизировать наш код позже, при обсуждении реализации на C или Cython, даже несмотря на то, что скомпилированный код дает гораздо больше возможностей сделать это.

В следующем подразделе посмотрим на расширения на чистом C.

Расширения на чистом языке C

Прежде чем полностью погрузиться в примеры кода расширений Python, написанных на C, нужно отметить важнейший момент. Если вы хотите расширить Python языком C, то должны хорошо знать оба языка. Особенно это касается C. Отсутствие опыта работы с ним может привести к катастрофе, поскольку ошибиться в нем легко.

Если вы решили, что вам точно нужно писать расширения C для Python, то мы предполагаем, что вы уже знаете язык C на уровне, который позволит вам полностью понять показанные здесь примеры. Разъясняться будут только детали Python/C API. Наша книга о Python, а не каком-либо другом языке. Если вы вообще не знаете C, то вам не стоит пытаться писать свои собственные расширения, пока не получите достаточно опыта и навыков. Оставьте эту задачу другим и используйте Cython или Pyrex, поскольку они намного безопаснее с точки зрения новичка. Все дело в том, что Python/C API, несмотря на тщательную проработку, не слишком хорошо подходит для начала работы с C.

Как предлагалось ранее, мы попытаемся портировать функцию `fibonacci()` на C и вставить ее в код Python в качестве расширения. Начнем с базовой реализации, которая будет аналогична предыдущему примеру на Python. Голые функции без использования Python/C API могут выглядеть грубо:

```
long long fibonacci(unsigned int n) {
    if (n < 2) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

А вот пример полного и абсолютно функционального расширения, которое передает эту единственную функцию в скомпилированный модуль:

```
#include <Python.h>

long long fibonacci(unsigned int n) {
    if (n < 2) {
        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): Return nth Fibonacci sequence number "
    "computed recursively\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",
    "Extension module that provides fibonacci sequence function",
    -1,
    fibonacci_module_methods
};

PyMODINIT_FUNC PyInit_fibonacci(void) {
    Py_Initialize();

    return PyModule_Create(&fibonacci_module_definition);
}
```

Представленный пример может вас огорчить, поскольку нам пришлось добавить в четыре раза больше кода просто для того, чтобы функция `fibonacci()`, написанная на C, стала доступна из Python. Позже мы обсудим каждую строку данного кода, так что не волнуйтесь. Но прежде чем сделаем это, мы посмотрим, как этот код можно упаковать и выполнить на Python.

Ниже приведена минимальная конфигурация `setuptools` для нашего модуля, которая должна использовать класс `setuptools.Extension`, чтобы сообщить интерпретатору, как компилируется наше расширение:

```
from setuptools import setup, Extension

setup(
    name='fibonacci',
    ext_modules=[
        Extension('fibonacci', ['fibonacci.c']),
    ]
)
```

Процесс сборки расширения инициализируется с помощью команды `setup.py build`, но также будет автоматически выполняться после установки пакета. Эти же файлы кода можно найти в каталоге `chapter9/fibonacci_c` в приложении к этой книге. Ниже показаны результат установки в режиме разработки и простая интерактивная сессия, где проверяется и выполняется наша скомпилированная функция `fibonacci()`:

```
$ ls -1ap
fibonacci.c
setup.py

$ python3 -m pip install -e .
Obtaining file:///Users/swistakm/dev/Expert-Python-Programming-
Third_edition/chapter9
Installing collected packages: fibonacci
  Running setup.py develop for fibonacci
Successfully installed Fibonacci

$ ls -1ap
build/
fibonacci.c
fibonacci.cpython-35m-darwin.so
fibonacci.egg-info/
setup.py
$ python3
Python 3.7.2 (default, Feb 12 2019, 00:16:38)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import fibonacci
>>> help(fibonacci.fibonacci)

Help on built-in function fibonacci in fibonacci:
fibonacci.fibonacci = fibonacci(...)
    fibonacci(n): Return nth Fibonacci sequence number computed recursively

>>> [fibonacci.fibonacci(n) for n in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

В следующем пункте поговорим о Python/C API.

Подробнее о Python/C API

Поскольку мы знаем, как правильно упаковать, скомпилировать и установить пользовательские расширения C, и уверены, что все будет работать ожидаемым образом, настало время обсудить наш код более подробно.

Модуль `extensions` начинается со следующей директивы препроцессора C, которая включает файл заголовка `Python.h`:

```
#include <Python.h>
```

Эта команда подтягивает Python/C API и все остальное, необходимое для написания расширения. В более реальных ситуациях вашему коду потребуется гораздо больше директив препроцессора, чтобы извлечь пользу из функций стандартной библиотеки C или интегрировать другие файлы кода. Наш пример был прост, поэтому больше не потребовалось никаких директив.

Далее ядро нашего модуля выглядит следующим образом:

```
long long fibonacci(unsigned int n) {
    if (n < 2) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

Функция `fibonacci()` — единственная часть нашего кода, делающая что-то полезное. Это реализация на чистом C, которую Python по умолчанию не понимает. Остальная часть нашего примера позволяет создать интерфейс, пропускающий функцию через API Python/C.

Первый этап воздействия данного кода на Python — создание функции C, которая совместима с интерпретатором CPython. В Python все является объектом. Это значит, что функции C, вызываемые в Python, также должны вернуть реальные объекты Python. В Python/C API есть тип `PyObject`, и каждый вызываемый объект должен вернуть указатель на него. Сигнатура нашей функции выглядит следующим образом:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args)
```

Обратите внимание: в этой сигнатуре не указан точный список аргументов, но только `PyObject* args` будет включать указатель на структуру, которая содержит кортеж предоставленных значений. Фактическая проверка списка аргументов должна выполняться внутри тела функции, и это именно то, что делает функция `fibonacci_py()`. Она анализирует список аргументов `args` при условии, что все они имеют тип `unsigned int`, и использует данное значение в качестве аргумента в функции `fibonacci()`, чтобы получить элемент последовательности Фибоначчи, как показано в следующем коде:

```
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;
}
```

В предыдущем примере есть серьезная ошибка, которую опытный разработчик легко заметит. Попробуйте найти его в качестве упражнения по работе с расширениями C. Пока оставим все как есть (для краткости). Мы постараемся все исправить позже, когда будем обсуждать борьбу с ошибками в пункте «Обработка исключений».

Строка `"l"` в вызове `PyArg_ParseTuple(args, "l", &n)` означает, что мы ожидаем в `args` одного значения типа `long`. В случае неудачи функция возвращает `NULL` и хранит информацию об исключении в потоке в состоянии интерпретатора. Более подробно об обработке поговорим в соответствующем пункте книги.

На самом деле сигнатура функции парсинга выглядит как `int PyArg_ParseTuple(PyObject *args, const char *format, ...)`, и после строки `format` идет список аргументов неизвестной длины, которые представляют собой разобранное выходное значение (в виде указателей). Это аналогично тому, как работает функция `scanf()` из стандартной библиотеки C. Если наше предположение неверно и пользователь введет список несовместимых аргументов, то `PyArg_ParseTuple()` выбросит соответствующее исключение. Этот способ кодирования сигнатур функций оказывается очень удобным, если привыкнуть к нему, но все равно является громоздким по сравнению с простым кодом Python. Такие сигнатуры Python, неявно определенные вызовом `PyArg_ParseTuple()`, сложно проверить внутри интерпретатора Python. Следует помнить об этом факте при использовании кода, предоставленного в виде расширений.

Как уже говорилось, Python ожидает, что из вызываемых объектов должны возвращаться некие объекты. То есть мы не можем вернуть сырое значение `long`, полученное из функции `fibonacci()`, в качестве результата `fibonacci_py()`. Такой код вообще не скомпилируется, поскольку не предусмотрена автоматическая переделка типов C в объекты Python. Вместо этого надо использовать функцию `Py_BuildValue(*format, ...)` — аналог `PyArg_ParseTuple()`, принимающий на вход подобный набор строк. Это не выход функции, а вход, вследствие чего это должны быть фактические значения, а не указатели.

Когда функция `fibonacci_py()` определена, большая часть тяжелой работы позади. Последним шагом будет выполнение инициализации модуля и добавление в нашу функцию метаданных, которые слегка упростят ее применение для пользователей. Это шаблонная часть кода нашего расширения и в ряде простых примеров наподобие нашего может занять больше места, чем сама функция, реализуемая нами. В большинстве случаев эта часть состоит из нескольких статических структур и одной функции инициализации, которые будут выполняться интерпретатором во время импорта модуля.

Сначала мы создаем строку `static`, которая будет являться строкой документации Python для функции `fibonacci_py()`, следующим образом:

```
static char fibonacci_docs[] =
    "fibonacci(n): Return nth Fibonacci sequence number "
    "computed recursively\n";
```

Обратите внимание: ее можно *встроить* где-то в конце `fibonacci_module_methods`, однако мы рекомендуем хранить строки документации отдельно, но в непосредственной близости от фактического определения функции, на которую они ссылаются.

Следующая часть нашего определения — это массив структур `PyMethodDef`, определяющие методы (функции), которые будут доступны в нашем модуле. Эта структура содержит четыре следующих поля:

- `char* ml_name` — имя метода;
- `PyCFunction ml_meth` — указатель на реализацию функции на C;
- `int ml_flags` — включает в себя флаги, указывающие на соглашение о вызовах или на связывающее соглашение. Последнее применимо только к определениям методов класса;
- `char* ml_doc` — указатель на содержание строки документации метода/функции.

Такой массив всегда должен заканчиваться контрольным значением `{NULL, NULL, 0, NULL}`. Оно указывает на конец структуры. В нашем простом случае мы создали массив `static fibonacci_module_methods PyMethodDef[]`, который содержит только два элемента (включая контрольное значение), следующим образом:

```
static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
    {NULL, NULL, 0, NULL}
};
```

Покажем, как первый элемент отображается на структуру `PyMethodDef`:

- `ml_name = "fibonacci"` — C-функция `fibonacci_py()` станет Python-функцией с именем `fibonacci`;
- `ml_meth = (PyCFunction)fibonacci_py` — переделка `PyCFunction` требуется Python/C API и диктуется соглашением, определенным далее в `ml_flags`;
- `ml_flags = METH_VARARGS` — флаг `METH_VARARGS` указывает на то, что соглашение о вызове нашей функции принимает список переменных аргументов и не имеет именованных аргументов;
- `ml_doc = fibonacci_docs` — функция Python будет документирована контентом строки `fibonacci_docs`.

Когда массив определений функций будет завершен, мы сможем создать еще одну структуру, которая содержит определение всего модуля. Она описывается

с помощью типа `PyModuleDef` и содержит несколько полей. Некоторые из них полезны только для более сложных сценариев, требующих точного контроля над процессом инициализации модуля. Здесь нас интересуют лишь первые пять из них:

- `PyModuleDef_Base m_base` — всегда должно быть инициализировано через `PyModuleDef_HEAD_INIT`;
- `char* m_name` — имя вновь созданного модуля; в нашем случае `fibonacci`;
- `char* m_doc` — указатель на содержимое строки документации модуля. Обычно у нас в одном исходном файле C определен всего один модуль, так что вполне нормально встраивать нашу строку документации по всей структуре;
- `Py_ssize_t m_size` — размер памяти, выделенной для поддержания состояния модуля. Используется, только когда требуется поддержка нескольких субинтерпретаторов или многофазная инициализация. В большинстве случаев не требуется, и размер памяти имеет значение `-1`;
- `PyMethodDef * m_methods` — указатель на массив, содержащий функции модульного уровня, описанные значениями `PyMethodDef`. Может иметь значение `NULL`, если в модуле нет каких-либо функций. В нашем случае это `fibonacci_module_methods`.

Остальные поля подробно описаны в официальной документации Python (docs.python.org/3/c-api/module.html), но для нашего примера они не требуются. Они должны иметь значения `NULL`, если не требуются, и будут неявно инициализированы этим значением при отсутствии заданных иных значений. Как следствие, наше описание модуля в переменной `fibonacci_module_definition` может иметь простой вид:

```
static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
    "fibonacci",
    "Extension module that provides fibonacci sequence function",
    -1,
    fibonacci_module_methods
};
```

Вишенка на торте нашей работы — функция инициализации модуля. Она должна именоваться с учетом весьма конкретных соответствующих правил, чтобы интерпретатор Python мог легко выбрать ее при загрузке динамической/общей библиотеки. Она должна называться `PyInit_ <name>`, где `<name>` — имя вашего модуля. Это точно та же строка, которая была использована как поле `m_base` в определении `PyModuleDef` и первый аргумент вызова `setuptools.Extension()`. Если вам не требуется сложный процесс инициализации модуля, то он принимает очень простую форму, как и в нашем примере:

```
PyMODINIT_FUNC PyInit_fibonacci(void) {
    return PyModule_Create(&fibonacci_module_definition);
}
```

Макрос `PyMODINIT_FUNC` — это макрос препроцессора, который будет объявлять тип возвращаемого значения данной функции инициализации как `PyObject*` и добавлять специальные декларации для связи, если того требует платформа.

В следующем пункте мы увидим, как можно вызывать и привязывать соглашения.

Вызов и привязка соглашений

Как мы говорили в предыдущем фрагменте текста, поле `ml_flags` структуры `PyMethodDef` содержит флаги для вызова и привязки соглашений. Описание *флагов соглашения о вызове* приведено ниже.

- `METH_VARARGS` — типичное соглашение для функции или метода Python, который принимает в качестве параметров только аргументы. Тип в поле `ml_meth` для такой функции должен быть `PyCFunction`. Функция будет снабжена двумя аргументами типа `PyObject*`. Первый — это объект `self` (для методов) или `module` (для функций модуля). Типичная сигнатура для функции C с таким соглашением — `PyObject* function(PyObject* self, PyObject* args)`.
- `METH_KEYWORDS` — соглашение для функции Python, которая при вызове принимает именованные аргументы. Ее соответствующий тип C — `PyCFunctionWithKeywords`. Функция C должна принимать три аргумента типа `PyObject*`: `self`, `args` и словарь именованных аргументов. В сочетании с `METH_VARARGS` первые два аргумента имеют такое же значение, как и для предыдущего вызова, а в противном случае `args` будет иметь значение `NULL`. Типичная сигнатура функции C — `PyObject* function(PyObject* self, PyObject* args, PyObject* keywds)`.
- `METH_NOARGS` — соглашение для функций Python, которые не принимают другие аргументы. Функция C должна быть типа `PyCFunction`, поэтому сигнатура такая же, как и для соглашения `METH_VARARGS` (аргументы `self` и `args`). Единственное отличие состоит в том, что `args` всегда имеет значение `NULL`, поэтому нет необходимости вызывать `PyArg_ParseTuple()`. Данный флаг не объединяется с любым другим флагом.
- `METH_O` — сокращение для функций и методов, принимающих в качестве аргументов одиночные объекты. Тип функции C снова `PyCFunction`, поэтому она принимает два аргумента `PyObject*`: `self` и `args`. Ее отличие от `METH_VARARGS` заключается в отсутствии необходимости вызывать `PyArg_ParseTuple()`, поскольку `PyObject*` в `args` уже является единственным аргументом, представленным в вызове `Python` к этой функции. Данный флаг не объединяется с любым другим флагом.

Функция, которая принимает ключевые слова, описывается либо с помощью METH_KEYWORDS, либо побитовой комбинацией флагов вызовов в виде METH_VARARGS | METH_KEYWORDS. Если да, то можно разбирать аргументы с помощью PyArg_ParseTupleAndKeywords() вместо PyArg_ParseTuple() или PyArg_UnpackTuple().

Ниже приведен пример модуля с единственной функцией, возвращающей None и принимающей два именованных аргумента, которые печатаются в стандартном выводе:

```
#include <Python.h>

static PyObject* print_args(PyObject *self, PyObject *args, PyObject *keywds)
{
    char *first;
    char *second;

    static char *kwlist[] = {"first", "second", 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,
    "print provided arguments"},
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef module_definition = {
    PyModuleDef_HEAD_INIT,
    "kwargs",
    "Keyword argument processing example",
    -1,
    module_methods
};

PyMODINIT_FUNC PyInit_kwargs(void) {
    return PyModule_Create(&module_definition);
}
```

Парсинг аргументов в Python/C API очень эластичен и был подробно описан в официальной документации (docs.python.org/3.7/c-api/arg.html). Аргумент формата в PyArg_ParseTuple() и PyArg_ParseTupleAndKeywords() позволяет осуществлять точный контроль над количеством аргументов и типами. Любое продвинутое со-

глашение о вызовах, известное из Python, может быть закодировано в C с помощью API, включая следующие:

- функции со значениями по умолчанию для аргументов;
- функции с аргументами — только ключевыми словами;
- функции с переменным количеством аргументов.

Флаги привязки соглашений `METH_CLASS`, `METH_STATIC` и `METH_COEXIST` зарезервированы для методов и не могут использоваться для описания функций модуля. Первые два пункта вполне очевидны. Они являются двойниками декораторов `classmethod` и `staticmethod` и изменяют значение аргумента `self`, переданного в функцию C.

Флаг `METH_COEXIST` позволяет загружать метод в месте существующего определения. Это редко бывает полезно. В основном данный флаг используется в случаях, когда нужно реализовать метод C, который будет генерироваться автоматически из других особенностей уже определенного типа. В документации Python приведен пример обертки `__contains__()`, который будет генерироваться, если в типе определен `sq_contains`. К сожалению, определение собственных классов и типов с помощью Python/C API выходит за рамки этой вводной главы.

Далее рассмотрим обработку исключений.

Обработка исключений

C, в отличие от Python или даже C++, не имеет синтаксиса для выбрасывания и перехвата исключений. Вся обработка ошибок обычно выполняется с помощью возвращаемых функций и необязательного глобального состояния для хранения информации, которая могла бы объяснить причину последнего сбоя.

Обработка исключений в Python/C API построена в соответствии с этим простым принципом. Существует глобальный индикатор последней произошедшей ошибки. Он задается для того, чтобы описать причину проблемы. Существует также стандартный способ проинформировать вызывающий объект данной функции о том, что это состояние было изменено во время вызова, например:

- если функция должна возвращать указатель, то возвращает `NULL`;
- если функция должна возвращать тип `int`, то возвращает `-1`.

Единственные исключения из предыдущих правил в Python/C API — это флаги `PyArg_*()`, которые возвращают `1` в случае успеха и `0` в случае сбоя.

Чтобы посмотреть, как это работает на практике, вспомним нашу функцию `fibonacci_py()`, упоминавшуюся ранее:

```
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;
}
```

Строки, которые тем или иным образом участвуют в обработке ошибок, выделены жирным. Обработка ошибок начинается в самом начале нашей функции с инициализации переменной `result`. Данная переменная должна хранить возвращаемое значение нашей функции. Она инициализируется как `NULL`, а мы уже знаем, что это — индикатор ошибки. Обычно расширения так и используются — с предположением о том, что ошибки для нашего кода — это нормально.

Затем у нас будет вызов `PyArg_ParseTuple()`, который установит информацию об ошибке на случай исключения и возврата `0`. Это часть оператора `if`, и в таком случае мы больше ничего не делаем и возвращаем `NULL`. Тот, кто вызывает нашу функцию, получит уведомление об ошибке.

`Py_BuildValue()` также может выбрасывать исключения. Предполагается вернуть `PyObject*` (указатель), поэтому в случае отказа он дает `NULL`. Мы можем просто хранить его в качестве нашей переменной результата и передавать дальше как возвращаемое значение.

Однако наша работа не заканчивается на обработке исключений, поднятых вызовом Python/C API. Вполне вероятно, что вам нужно будет сообщить пользователю о том, какая именно ошибка произошла. В Python/C API есть несколько функций, которые помогут вам выбросить исключение, но наиболее распространена `PyErr_SetString()`. Она устанавливает индикатор ошибки с заданным типом исключения и с дополнительной строкой, представленной в качестве объяснения причины ошибки.

Полная сигнатура этой функции выглядит следующим образом:

```
void PyErr_SetString(PyObject* type, const char* message)
```

Мы уже говорили о том, что в реализации нашей функции `fibonacci_py()` есть серьезная ошибка. Сейчас самое время поговорить о ней и попытаться исправить ее. К счастью, у нас уже есть необходимые для этого инструменты. Проблема заключается в небезопасном преобразовании `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 будет интерпретироваться как тип `long` (спецификатор `"l"`) и хранится в локальной переменной `n`. Затем он превращается в `unsigned int`, поэтому может возникнуть проблема, если пользователь вызывает функцию `fibonacci()` из Python с отрицательным значением. Например, `-1` в виде 32-разрядного целого числа со знаком бу-

дет интерпретироваться как 4294967295 при превращении в беззнаковое 32-битное целое число. Такое значение приведет к очень глубокой рекурсии, переполнению стека и ошибкам сегментации. Обратите внимание: то же самое может произойти, если пользователь задаст сколь угодно большой положительный аргумент. Мы не можем исправить это без полной реконструкции функции `fibonacci()`, но можем по крайней мере попытаться убедиться в том, что входной аргумент функции удовлетворяет некоторым предварительным условиям. Здесь мы убеждаемся, что значение аргумента `n` больше или равно `0`, и выбрасываем исключение `ValueError`, если это не так:

```
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;
}
```

Последнее, что нам нужно сказать об обработке исключений, — глобальная ошибка не устранится сама собой. Некоторые ошибки можно корректно обработать в функциях C (как и в операторе `try … except` в Python), и вы должны иметь возможность очистить индикатор ошибки, если он больше не нужен. Для этого понадобится функция `PyErr_Clear()`.

В следующем пункте текста мы обсудим выпуск GIL.

Выпуск GIL

Мы уже говорили, что расширения можно использовать для обхода GIL Python. В CPython есть известное ограничение реализации, которое заключается в том, что только один поток единовременно может выполнять код Python. Устранить данную проблему можно с помощью многопроцессорного подхода (см. главу 15), но это плохое решение для некоторых выраженно параллельных алгоритмов, так как запуск дополнительных процессов влечет за собой большие затраты.

Поскольку расширения в основном применяются в тех случаях, когда большая часть работы выполняется на чистом C без вызовов Python/C API, можно (и даже желательно) в некоторых разделах приложения задействовать GIL, выполняя в то же время обработку данных. Благодаря этому вы можете извлечь выгоду из

многоядерности процессора и многопоточности приложения. Единственное, что вам нужно сделать, — обернуть блоки кода, в которых не используются вызовы Python/C API или Python, специальными макросами, имеющимися в Python/C API. Два приведенных ниже макроса препроцессора служат для упрощения всей процедуры выпуска и возврата GIL:

- `Py_BEGIN_ALLOW_THREADS` — объявляет скрытую локальную переменную, в которой сохраняется текущее состояние потока, и выпускает GIL;
- `Py_END_ALLOW_THREADS` — возвращает GIL и восстанавливает состояние потока из локальной переменной, объявленной в предыдущем макросе.

Если мы внимательно посмотрим на наш пример расширения `fibonacci`, то увидим: в функции `fibonacci()` нет кода на Python и вызова каких-либо структур Python. Это значит, что функция `fibonacci_py()`, которая просто оборачивает исполнение `fibonacci()`, позволяет при должных изменениях выпустить GIL вокруг данного вызова следующим образом:

```
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;
}
```

Описанный выше метод справедлив, но его использование требует осторожности.

Далее поговорим о подсчете ссылок.

Подсчет ссылок

Наконец, мы подошли к важному вопросу об управлении памятью в Python. В Python есть собственный сборщик мусора, однако тот предназначен только для удаления циклических ссылок в алгоритме *подсчета ссылок*. Подсчет ссылок — это основной способ уничтожения объектов, которые больше не нужны программе.

В документации API Python/C вводится *владение ссылками*, объясняющее, как выполняется уничтожение объектов. Объекты в Python никогда не привязываются к ссылкам и являются общими. Фактическое создание объектов управляется менеджером памяти Python. Это компонент интерпретатора CPython, который отвечает за выделение и освобождение памяти для объектов, хранящихся в общей куче. То есть работать можно со ссылками на объект.

Каждый объект в Python, представленный ссылкой (указатель `PyObject*`), имеет соответствующий счетчик ссылок. Когда он равняется нулю, это означает, что ссылок на данный объект больше нет и можно вызывать уничтожитель, связанный с этим типом. В Python/C API есть два макроса для увеличения и уменьшения счетчика ссылок: `Py_INCREF()` и `Py_DECREF()`. Но прежде чем мы поговорим о них подробнее, нужно разъяснить следующие термины, связанные с владением ссылками.

- *Передача владения* — всякий раз, когда мы говорим *«функция передает владение ссылкой»*, это означает, что она уже увеличила счетчик ссылок и именно вызывающий должен уменьшить счетчик, когда ссылка на объект больше не нужна. Большинство функций, которые возвращают вновь созданные объекты, такие как `Py_BuildValue`, выполняет эту задачу. Если объект возвращается из нашей функции другому абоненту, то право собственности передается снова. В таком случае мы не уменьшаем счетчик ссылок, поскольку это уже не наша ответственность. Как следствие, функция `fibonacci_py()` не вызывает `Py_DECREF()` для переменной `result`.
- *Заимствованные ссылки* — возникают, когда функция получает ссылку на какой-либо объект Python в качестве аргумента. Счетчик ссылок для такой ссылки в данной функции не уменьшается, если он явно не увеличивался. В нашей функции `fibonacci_py()` аргументы `self` и `args` — такие заимствованные ссылки, и поэтому мы не вызываем для них `PyDECREF()`. Некоторые из функций API Python/C позволяют возвращать заимствованные ссылки. Заметные примеры — `PyTuple_GetItem()` и `PyList_GetItem()`. Часто говорят, что такие ссылки являются *незащищенными*. Нет необходимости уничтожать владение, если они не будут возвращены, как возвращаемое значение функции. Чаще всего дополнительное внимание требуется в случае применения таких заимствованных ссылок в качестве аргументов других вызовов API Python/C. Это может быть необходимо, когда требуется дополнительная защита таких ссылок отдельной функцией `Py_INCREF()` перед ее использованием в качестве аргумента для других функций, а затем можно вызвать функцию `Py_DECREF()`, когда ссылка больше не нужна.
- *Украденные ссылки* — функция API Python/C может *украсть* ссылку вместо того, чтобы *заимствовать*, если она является аргументом. Это случай для двух функций — `PyTuple_SetItem()` и `PyList_SetItem()`. Они в полной мере берут на себя ответственность за передаваемые им ссылки. Сами по себе они не увеличивают счетчик ссылок, но вызывают `Py_DECREF()`, когда ссылка больше не нужна.

Следить за счетчиками ссылок — одна из самых непростых вещей при написании сложных расширений. Некоторые из неочевидных проблем могут остаться незамеченными, пока код не будет работать в многопоточном виде.

Другая распространенная проблема вызвана самой природой объектной модели Python и тем фактом, что некоторые функции возвращают заимствованные ссылки. Когда счетчик ссылок обнуляется, выполняется функция уничтожения. Для классов, определенных пользователем, можно задать метод `__del__()`, который будет вызываться в данный момент. Это может быть любой код Python, и, вероятно, он будет влиять на другие объекты и их счетчики ссылок. В официальной документации Python приведен следующий пример кода, в котором из-за этого могут возникнуть проблемы:

```
void bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

Пример выглядит совершенно безвредно, но проблема в том, что мы не знаем, какие элементы содержит объект `list`. Когда `PyList_SetItem()` устанавливает новое значение элемента `list[1]`, владение объектом, который ранее хранился по данному индексу, уничтожается. Если это единственная существующая ссылка, то счетчик ссылок обнуляется и объект уничтожается. Вполне возможно, что это был какой-то пользователь класса с собственной реализацией метода `__del__()`. Возникнет серьезная проблема при удалении из списка элемента `list[0]` в результате такого метода. Обратите внимание: `PyList_GetItem()` возвращает *заимствованную* ссылку! Она не вызывает `Py_INCREF()` перед возвращением ссылки. Поэтому в данном коде возможна ситуация, что `PyObject_Print()` будет вызываться со ссылкой на объект, которого больше не существует. Это приведет к ошибке сегментации и сбою интерпретатора Python.

Правильный подход заключается в защите заимствованных ссылок, пока они еще нужны, поскольку есть вероятность того, что любой вызов может вызвать уничтожение объекта. Это может произойти, даже если они на первый взгляд не связаны, как показано в следующем коде:

```
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);
}
```

В следующем подразделе мы узнаем, как писать расширения, используя Cython вместо чистого Python/C API.

Написание расширений на Cython

Cython — одновременно оптимизирующий статический компилятор и язык программирования из надмножества Python. Как компилятор, он может выполнять компиляцию *«код — код»* нативного Python и расширений на Cython на Python C с использованием Python/C API. Это позволяет объединить мощь Python и C, не прибегая к необходимости разбираться с Python/C API вручную.

Поговорим о Cython как о компиляторе «код — код».

Cython как компилятор «код — код»

Основное преимущество расширений, созданных с помощью Cython, заключается в языке надмножества. Однако можно создавать расширения из простого кода Python с помощью компиляции «код — код». Это самый простой подход к Cython, поскольку он не требует почти никаких изменений в коде и позволяет значительно улучшить производительность при очень малых затратах на развитие.

В Cython есть простая функция `cythonize`, которая позволяет легко интегрировать процесс компиляции с `distutils` или `setuptools`. Допустим, мы хотели бы получить чистую реализацию Python нашей функции `fibonacci()` для расширения C. Если она находится в модуле `fibonacci`, то минимальный `setup.py` может выглядеть следующим образом:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name='fibonacci',
    ext_modules=cythonize(['fibonacci.py'])
)
```

Cython, будучи используемым в качестве компилятора для языка Python, имеет еще одно преимущество. Компиляция расширения типа «код — код» может быть совершенно необязательной частью процесса установки дистрибутива. Если в среде, в которой нужно установить пакет, нет Cython или каких-либо других предпосылок для сборки, то он может быть установлен как обычный пакет на *чистом Python*. При этом пользователь не должен заметить никакой функциональной разницы в поведении установленного таким образом кода. Общий подход к распространению расширений, собранных на Cython, — включить код на Python/Cython и код C, который будет генерироваться из этих исходных файлов. Итак, пакет можно установить тремя различными способами, в зависимости от наличия предпосылок:

- если в среде установки есть Cython, то код расширения C генерируется из имеющегося кода Python/Cython;

- если Cython не установлен, но все предпосылки имеются (компилятор C, заголовки Python/C API), то расширение собирается из предварительно сгенерированных файлов C;
- если предыдущие условия не выполнены, но расширение создается из чистого кода Python, то модули устанавливаются как обычный код Python, а этап компиляции пропускается.

Обратите внимание: в документации по Cython говорится, что включение созданных файлов C и кода на Cython — рекомендуемый способ распространения расширений на Cython. Вдобавок утверждается, что компиляция Cython по умолчанию должна быть отключена, поскольку пользователь может не иметь в среде нужной версии Cython и это чревато неожиданными проблемами компиляции. Хотя с появлением изоляции среды данная проблема отошла на второй план. Кроме того, Cython — действующий пакет Python, доступный на PyPI, поэтому его можно легко определить как требование проекта, причем в конкретной версии. Включение такой предпосылки — решение с серьезными последствиями, и рассматривать его следует весьма тщательно. Более безопасное решение — применить силу функции `extras_require` в пакете `setuptools` и позволить пользователю самому решить, хочет ли он задействовать Cython с определенной переменной среды, например:

```
import os

from distutils.core import setup
from distutils.extension import Extension

try:
    # Компиляция исходного кода на Cython доступна
    # только при наличии Cython
    import Cython
    # И определенная переменная среды явно сообщает
    # о том, что Cython должен использоваться для генерации
    # C-источников
    USE_CYTHON = bool(os.environ.get("USE_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={
        # Cython будет настроен в данной версии как обязательный,
        # если пакет установлен с дополнительной функцией '[with-cython]'
        'with-cython': ['cython==0.23.4']
    }
)
```

Установочный инструмент `pip` поддерживает установку пакетов с опцией *extras* путем добавления к имени пакета суффикса `[extra-name]`. В предыдущем примере дополнительное требование Cython и компиляция во время установки из локально расположенного кода включается с помощью следующей команды:

```
$ USE_CYTHON=1 pip install .[with-cython]
```

Переменные среды `USE_CYTHON` гарантируют, что `pip` будет использовать Cython для компиляции файлов `.pyx` в C, а `[with-cython]` — что компилятор будет скачан Cython перед установкой.

Далее поговорим о Cython как о языке.

Cython как язык

Cython не только компилятор, но и надстройка языка Python. Это значит, что в нем допускается любой код Python и возможно использование дополнительных функций, таких как поддержка вызова функций C или объявление типов C на переменных и атрибутах класса. Следовательно, любой код, написанный на Python, можно писать и на Cython. Это объясняет, почему обычные модули Python настолько легко компилируются на C с помощью компилятора Cython.

Однако на этом мы не остановимся. Вместо того чтобы говорить, будто наша функция `fibonacci()` — еще и код допустимых расширений в надстройке Python, мы попробуем внести некие улучшения. Это не внесет никакой реальной оптимизации в нашу функцию, но тем не менее позволит коду получить больше выгоды от записи на Cython.

В коде Cython используется другое расширение файла: `.pyx` вместо `.py`. Мы все еще хотим реализовать нашу последовательность Фибоначчи в виде рекурсивного алгоритма.

Содержание `fibonacci.pyx` может выглядеть следующим образом:

```
"""Модуль Cython, который реализует функцию последовательности Фибоначчи"""

def fibonacci(unsigned int n):
    """Возвращение n-го числа Фибоначчи, вычисляется рекурсивно"""
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Как видим, единственное, что действительно изменилось, — сигнатура функции `fibonacci()`. Благодаря опциональной статической типизации в Cython мы можем присвоить аргументу `n` тип `unsigned int`, и это должно немного улучшить работу нашей функции. Кроме того, она делает гораздо больше, чем мы делали ранее при написании расширений вручную. Если аргумент функции Cython объявлен со статическим типом, то расширение автоматически обработает ошибки преобразования и переполнения за счет выбрасывания собственных исключений следующим образом:

```
>>> from fibonacci import fibonacci
>>> fibonacci(5)
5
>>> fibonacci(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 21, in fibonacci.fibonacci (fibonacci.c:704)
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 21, in fibonacci.fibonacci (fibonacci.c:704)
OverflowError: value too large to convert to unsigned int
```

Мы уже знаем, что Cython компилирует только из *кода в код*, и сгенерированный код задействует тот же API Python/C, который мы будем использовать при написании расширений C вручную. Следует отметить: `fibonacci()` — рекурсивная функция, поэтому очень часто вызывает сама себя. Это будет означать, что, хоть мы и объявили тип `static` входного аргумента, при рекурсивном вызове она будет относиться к себе так же, как и к любой другой функции Python. Таким образом, числа `n-1` и `n-2` будут упакованы обратно в объект Python и затем переданы на скрытый слой внутренней реализации `fibonacci()`, который станет возвращать тип `unsigned int`. Так будет происходить снова и снова, пока мы не достигнем предела глубины рекурсии. Это не обязательно станет проблемой, но требует гораздо большего объема обработки аргументов, чем нужно на самом деле.

Мы можем снизить затраты на вызовы функций Python и обработку аргументов, передавая больше работы чистой функции C, которая ничего не знает о структурах Python. Мы делали это ранее при создании расширений C на чистом C и можем повторить в Cython. Можно использовать ключевое слово `cdef`, чтобы объявить функции в стиле C, принимающие и возвращающие только типы C:

```
cdef long long fibonacci_cc(unsigned int n):
    if n < 2:
        return n
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)
```

```
def fibonacci(unsigned int n):
    """Возвращаем n-й элемент из последовательности чисел Фибоначчи,
       вычисленный рекурсивно"""
    return fibonacci_cc(n)
```

Можно пойти еще дальше. Пример на чистом C показывает, как выпустить GIL во время вызова нашей чистой функции C, вследствие чего это расширение слегка улучшилось для многопоточных приложений. В предыдущих примерах мы использовали макросы препроцессора `Py_BEGIN_ALLOW_THREADS` и `Py_BEGIN_ALLOW_THREADS` из заголовков API Python/C, чтобы отметить раздел кода без вызовов Python. Синтаксис Cython намного короче, и его легче запомнить. GIL можно выпустить вокруг секции кода, задействуя простой оператор `with nogil` следующим образом:

```
def fibonacci(unsigned int n):
    """Возвращаем n-й элемент из последовательности чисел Фибоначчи,
       вычисленный рекурсивно"""
    with nogil:
        result = fibonacci_cc(n)

    return fibonacci_cc(n)
```

Можно также пометить всю функцию C как безопасную для вызова без GIL следующим образом:

```
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 такие функции не могут принимать в качестве аргументов или возвращать объекты Python. Всякий раз, когда функция помечается как `nogil`, необходимо выполнить любой вызов Python/C API, и он должен вернуть GIL с помощью оператора `with gil`.

В следующем разделе поговорим о проблемах с использованием расширений.

Проблемы с использованием расширений

Честно говоря, мы начали работу с Python только потому, что устали от сложности написания программ на C и C++. На самом деле довольно часто программисты начинают изучать Python, придя к пониманию того, что другие языки не позволяют получить то, что нужно их пользователям. Программирование на Python, по сравнению с C, C++ или Java, — легкая прогулка по вечернему пляжу. Все кажется простым и хорошо продуманным. Кажется, словно здесь все идеально и другие языки программирования вообще больше не нужны.

Но, конечно, думать так ошибочно. Да, Python — удивительный язык с большим количеством интересных функций и используется во многих областях. Но это вовсе не означает, что он идеален и не имеет никаких недостатков. Его легко понять и на нем легко писать, но за легкость приходится платить. Он не настолько медленный, как многие думают, но никогда не будет быстрым, как C. Он весьма портативен, но его интерпретатор не так распространен в разных архитектурах и компиляторах, как у других языков. И этот список можно продолжать очень долго.

Чтобы исправить данную проблему, были написаны расширения, тем самым произведен перенос некоторых преимуществ *старого доброго* C обратно в Python. И в большинстве случаев это работает хорошо. Но вопрос вот в чем — мы действительно хотим использовать Python, чтобы в конечном итоге писать расширения на C? Ответ: *нет*. Это лишь неудобная необходимость в ситуациях, когда у нас не хватает возможностей.

О других трудностях поговорим в следующем подразделе.

Дополнительная сложность

Не секрет, что разработка приложений на разных языках — непростая задача. Python и C — совершенно разные технологии, и найти в них общие черты очень трудно. Кроме того, не существует приложений, в которых не было бы ошибок. Если расширения стали обычным явлением в вашей кодовой базе, то отладка превращается в сущий ад. Дело не только в том, что отладка кода C требует совершенно иного рабочего процесса и инструментария, но и в предстоящей необходимости часто переключать контекст между двумя различными языками.

Мы все люди, и наши когнитивные способности ограниченны. Разумеется, есть люди, которые могут обрабатывать несколько уровней абстракции одинаково эффективно, но это редкость. Независимо от вашего опыта и навыков, за сопровождение таких гибридных решений всегда придется чем-то платить. Платой могут быть дополнительные трудозатраты и время, необходимое для переключения между C и Python, или дополнительный стресс, который в конечном счете приведет к снижению вашей эффективности.

Согласно индексу TIOBE, C — все еще один из самых популярных языков программирования. Несмотря на это, весьма часто программисты на Python знают его очень мало или вообще не знают. Лично я считаю, что C должен стать *общепринятым языком* программирования, но мое мнение вряд ли что-то изменит в данном вопросе. Python столь соблазнителен и легок в изучении, что многие программисты забывают весь свой прошлый опыт и с радостью полностью переходят на новую технологию. Но программирование — не езда на велосипеде. Это умение очень быстро забывается, если его забросить и не оттачивать. Даже программисты на C с большим

опытом рискуют утратить свои знания, слишком сильно погрузившись в Python. Все это ведет к простому выводу о том, что труднее найти людей, которые будут в состоянии понять и дополнить ваш код. Для пакетов с открытым исходным кодом речь идет о меньшем количестве помощников. Для закрытого кода это означает, что не все ваши товарищи по команде смогут корректно работать с вашим кодом.

В следующем подразделе поговорим об отладке.

Отладка

Когда дело доходит до ошибок, расширение ломается красиво и с треском. Статическая типизация дает много преимуществ по сравнению с Python и позволяет обнаруживать на этапе компиляции такие проблемы, которые будет трудно заметить в Python. Это может произойти даже без тщательного тестирования и полноценных испытаний. С другой стороны, управление памятью в таком случае организуется вручную, а плохое управление памятью — основная причина большинства ошибок в C. В лучшем случае такие ошибки приведут к паре утечек памяти, которые приведут к поглощению всех ресурсов среды. Но решить данную проблему сложно. Действительно трудно искать утечки памяти без использования надлежащих внешних инструментов, таких как Valgrind. В большинстве случаев проблемы управления памятью в коде расширения приведут к ошибкам сегментации, которые в Python не исправляются и заставляют интерпретатор останавливать работу, не выбрасывая исключение. То есть вам все равно придется вооружиться дополнительными инструментами, которые большинству программистов Python обычно не требуются. Это усложняет вашу среду разработки и рабочий процесс.

В следующем разделе рассматривается взаимодействие с динамическими библиотеками без использования расширений.

Взаимодействие с динамическими библиотеками без расширений

Благодаря `ctypes` (модулю в стандартной библиотеке) или `cffi` (внешнему пакету) вы можете интегрировать все скомпилированные динамические/разделяемые библиотеки в Python, независимо от того, на каком языке они были написаны. Вы можете сделать это на чистом Python без какой-либо компиляции, что является интересной альтернативой написанию собственных расширений в C.

Это не значит, что вам не нужно ничего знать о C. Оба решения требуют от вас понимания C и принципов работы динамических библиотек. С другой стороны, они снимают бремя борьбы с подсчетом ссылок на Python и значительно снижают

риск болезненных ошибок. Кроме того, взаимодействие с кодом C через `ctypes` или `cffi` более компактно, чем написание и компиляция модулей расширения C.

В следующем подразделе посмотрим на модуль `ctypes`.

Модуль ctypes

Модуль `ctypes` — самый популярный модуль для вызова функций из динамических или разделяемых библиотек без необходимости написания пользовательских расширений на C. Причина тому очевидна. Он является частью стандартной библиотеки, поэтому всегда доступен и внешних зависимостей не требуется. Это библиотека *Foreign Function Interface (FFI)*, которая предоставляет API-интерфейсы для создания C-совместимых типов данных.

Далее рассмотрим загрузку библиотек.

Загрузка библиотек

Существует четыре типа загрузчиков динамических библиотек в `ctypes` и две конвенции, регулирующие их использование. Классы, которые представляют собой динамические и разделяемые библиотеки, — `ctypes.CDLL`, `ctypes.PyDLL`, `ctypes.OleDLL` и `ctypes.WinDLL`. Последние две доступны только в Windows, поэтому здесь мы не будем обсуждать их подробно. Различия между `CDLL` и `PyDLL` заключаются в следующем:

- класс `ctypes.CDLL` представляет собой подгружаемые общие библиотеки. Функции в этих библиотеках используют стандартное соглашение о вызовах и возвращают `int`. Во время разговора выпускается GIL;
- класс `ctypes.PyDLL` работает как `CDLL`, но GIL во время вызова не выпускается. После выполнения проверяется флаг ошибки Python и выбрасывается исключение, если он установлен. Это полезно только в случае вызова загруженной библиотекой функции непосредственно из Python/C API или использования функций обратного вызова, которые могут быть кодом Python.

Чтобы загрузить библиотеку, вы можете создать экземпляр одного из предыдущих классов с соответствующими аргументами или 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 заключается в их портативном поиске. В различных системах используются разные суффиксы для раз-

деляемых библиотек (`.dll` на Windows, `.dylib` на macOS, `.so` на Linux) и поиск выполняется в разных местах. Хуже всего здесь Windows, в которой нет предопределенной схемы именования для библиотек. Как следствие, мы не будем обсуждать подробности загрузки библиотек с `ctypes` в данной системе и сконцентрируемся в основном на Linux и macOS, в которых эта проблема решается последовательно и похожим образом. Если вы заинтересованы в платформе Windows, то обратитесь к официальной документации `ctypes`, в которой много информации о поддержке данной системы (docs.python.org/3.5/library/ctypes.html).

Оба соглашения загрузки библиотек (функция `LoadLibrary()` и конкретные классы типа библиотеки) требуют использования полного имени библиотеки. Это означает, что все префиксы и суффиксы предопределенных библиотек должны быть включены. Например, для загрузки стандартной библиотеки C на Linux вам нужно написать следующее:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.so.6')
<CDLL 'libc.so.6', handle 7f0603e5f000 at 7f0603d4cbd0>
```

Для macOS это выглядело бы так:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.dylib')
```

К счастью, в подмодуле `ctypes.util` есть функция `find_library()`, которая позволяет загрузить библиотеку, используя ее имя без префиксов или суффиксов, и будет работать в любой системе, имеющей определенную схему именования разделяемых библиотек:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> ctypes.cdll.LoadLibrary(find_library('c'))
<CDLL '/usr/lib/libc.dylib', handle 7fff69b97c98 at 0x101b73ac8>
>>> ctypes.cdll.LoadLibrary(find_library('bz2'))
<CDLL '/usr/lib/libbz2.dylib', handle 10042d170 at 0x101b6ee80>
>>> ctypes.cdll.LoadLibrary(find_library('AGL'))
<CDLL '/System/Library/Frameworks/AGL.framework/AGL', handle 101811610 at
0x101b73a58>
```

Так что, если вы пишете пакет `ctypes`, который должен работать как в macOS, так и в Linux, то всегда используйте `ctypes.util.find_library()`.

Вызов функции C с помощью `ctypes` описывается ниже.

Вызов функции C помощью ctypes

Когда динамическая/общая библиотека успешно загружена в объект Python, ее обычно сохраняют в качестве переменной уровня модуля с тем же именем, что и у загруженной библиотеки. Эти функции доступны в качестве атрибутов объекта,

и вызываются они так же, как функции Python из любого другого импортируемого модуля:

```
>>> 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 вследствие этого должны быть обернуты в соответствующие классы, имеющиеся в модуле `ctypes`. В табл. 9.1 приведен полный список совместимых типов данных из документации `ctypes`.

Таблица 9.1. Полный список совместимых типов данных из документации ctypes

Тип ctypes	Тип C	Тип Python
c_bool	_Bool	BOOL
c_char	char	Односимвольный объект bytes
c_wchar	wchar_t	Односимвольная строка
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned int	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned 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* (NUL terminated)	Объект Bytes или None
c_wchar_p	wchar_t* (NUL terminated)	string или None
c_void_p	void*	int или None

Как вы можете видеть, в таблице нет специальных типов, которые превращали бы коллекции Python в массивы C. Рекомендуемый способ создания типов для массивов C — простое использование оператора умножения с требуемым типом `ctypes` следующим образом:

```
>>> 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
```

Такой же синтаксис работает для каждого основного типа `ctypes`.

В следующем пункте посмотрим на то, как функции Python передаются в виде обратных вызовов C.

Передача функций Python в виде обратных вызовов C

Очень популярный паттерн проектирования заключается в том, чтобы делегировать часть работы по реализации функции в пользовательские обратные вызовы. Наиболее известная функция из стандартной библиотеки C, принимающая такие обратные вызовы функции, — это `qsort()`, которая обеспечивает общую реализацию алгоритма *быстрой сортировки*. Весьма маловероятно, что вы захотите использовать данный алгоритм вместо стандартного алгоритма *TimSort*, реализованного в интерпретаторе CPython, который больше подходит для сортировки коллекций Python. Однако `qsort()` кажется каноническим примером эффективного алгоритма сортировки и API C, который использует механизм обратного вызова, описанный во многих книгах по программированию. Поэтому мы будем стараться использовать его в качестве примера прохождения функции Python в виде обратного вызова C.

Обычный тип функции Python несовместим с типом функции обратного вызова, требуемым спецификацией `qsort()`. Вот сигнатура `qsort()` со страницы BSD `man`, которая также содержит тип принимаемого обратного вызова (аргумент `compar`):

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

Поэтому для выполнения `qsort()` из `libc` вам потребуется:

- `base` — массив, который сортируется как указатель `void*`;
- `nel` — число элементов как `size_t`;
- `width` — размер одного элемента в массиве `size_t`;

- `compar` — указатель на функцию, которая должна возвращать `int` и принимает два указателя `void*`. Он указывает на функцию, которая сравнивает размер двух упорядоченных элементов.

Из пункта «Вызов функции C с помощью ctypes» мы уже знаем, как построить массив C из другого типа `ctypes`, используя оператор умножения. `nel` должен быть `size_t` и отображает `int` на Python, поэтому не требует никакой дополнительной упаковки и может быть передан как `len(itarable)`. Значение `width` может быть получено с помощью функции `ctypes.sizeof()`, как только мы узнаем тип нашего массива `base`. Последнее, что нам нужно знать, — это как создать указатель на функцию Python, совместимую с аргументом `compar`.

Модуль `ctypes` содержит функцию `CFUNCTYPE()`, которая позволяет обернуть функцию Python и представить ее в виде вызываемого указателя функции C. Первый аргумент — это возвращаемый тип C, который должна возвращать обернутая функция.

За ним следует список переменных типов C, который функция принимает в качестве аргументов. Тип функции, совместимый с аргументом `compar` `qsort()`, будет выглядеть следующим образом:

```
CMPFUNC = ctypes.CFUNCTYPE(
    # Возвращаемый тип
    ctypes.c_int,
    # Тип первого аргумента
    ctypes.POINTER(ctypes.c_int),
    # Тип второго аргумента
    ctypes.POINTER(ctypes.c_int),
)
```

CFUNCTYPE() задействует соглашение о вызове cdecl, поэтому она совместима только с CDLL и PyDLL. Динамические библиотеки в операционной системе Windows, которые загружаются с WinDLL или OleDLL, применяют соглашение sdtcall. Это означает, что для обертки функции Python как указателя C используется другой механизм. В ctypes это WINFUNCTYPE().

Чтобы обернуть все это, предположим, что хотим отсортировать случайный список целых чисел с помощью функции `QSort()` из стандартной библиотеки C. Вот пример скрипта, который показывает, как сделать это, используя наши новые знания 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 — указатели, поэтому нужен индекс [0]
    print(" %s cmp %s" % (a[0], b[0]))

    # По спецификации qsort должно возвращаться:
    # * меньше нуля, если a < b,
    # * ноль, если a == b,
    # * больше нуля, если a > b.
    return a[0] - b[0]

def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("shuffled: ", numbers)

    # Создается новый тип — массив длиной,
    # равной длине списка чисел
    NumbersArray = ctypes.c_int * len(numbers)
    # Создается новый массив C с помощью нового типа
    c_array = NumbersArray(*numbers)

    libc.qsort(
        # Указатель на отсортированный массив
        c_array,
        # Длина массива
        len(c_array),
        # Размер элемента массива
        ctypes.sizeof(ctypes.c_int),
        # Обратный вызов (указатель на функцию сравнения C)
        CMPFUNC(ctypes_int_compare)
    )
    print("sorted: ", list(c_array))

if __name__ == "__main__":
    main()
```

Функция сравнения, которая предоставляется в виде обратного вызова, имеет дополнительный оператор `print`, поэтому видно, как она выполняется в процессе сортировки:

```
$ python ctypes_qsort.py
shuffled:  [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
sorted:    [0, 1, 2, 3, 4]
```

Конечно, использование `qsort` в Python имеет мало смысла, поскольку в Python есть свой собственный специализированный алгоритм сортировки. Но зато передача функции Python в виде обратного вызова C — очень полезный метод для интеграции сторонних библиотек.

В следующем подразделе поговорим о CFFI.

CFFI

CFFI — это FFI для Python и интересная альтернатива `ctypes`. Она не является частью стандартной библиотеки, но зато доступна на PyPI под именем `cffi`. Она отличается от `ctypes`, поскольку тут делается больший акцент на повторном использовании простых деклараций C вместо предоставления обширного Python API в одном модуле. Этот путь сложнее и имеет особенность, позволяющую автоматически компилировать некоторые части интеграционного слоя в расширения с помощью компилятора C. Таким образом, CFFI можно использовать в качестве гибридного решения, заполняющего пробел между простыми расширениями C и `ctypes`.

Поскольку это очень большой проект, описать его в двух словах невозможно. С другой стороны, было бы стыдно не уделить ему внимания. Мы уже обсудили один пример интегрирования функции `qsort()` из стандартной библиотеки с использованием `ctypes`. Таким образом, лучший способ показать основные различия между этими двумя решениями — определить тот же пример с помощью `cffi`. Мы надеемся, следующий блок поможет вам. «Просто “Питон” вместо тысячи слов»:

```
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.
    # Магии здесь меньше, чем в ctypes,
    # но зато больше точности и требуется
    # явное преобразование.
    int_a = ffi.cast('int*', a)[0]
    int_b = ffi.cast('int*', b)[0]
    print(" %s cmp %s" % (int_a, int_b))

    # По спецификации qsort должно возвращаться:
    # * меньше нуля, если a < b,
    # * ноль, если a == b,
    # * больше нуля, если a > b
    return int_a - int_b

def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("shuffled: ", numbers)

    c_array = ffi.new("int[]", numbers)

    C.qsort(
        # Указатель на отсортированный массив
        c_array,
        # Длина массива
        len(c_array),
        # Размер элемента массива
        ffi.sizeof('int'),
        # Обратный вызов (указатель на функцию сравнения C)
        cffi_int_compare,
    )
    print("sorted: ", list(c_array))

if __name__ == "__main__":
    main()
```

Результат будет аналогичен тому, который был приведен выше, при обсуждении примера обратных вызовов C в `ctypes`. Использование CFFI для интеграции `qsort` в Python не более осмысленно, чем применение `ctypes` для той же цели. Во всяком случае, предыдущий пример должен показать основные различия между `ctypes` и CFFI относительно обработки типов данных и функций обратного вызова.

Резюме

В данной главе описан один из самых сложных вопросов, поднимаемых в нашей книге. Мы обсудили причины создания расширений Python и привели примеры соответствующих инструментов. Мы начали писать расширения на чистом C, которые зависят только от API Python/C, а затем переписали их на Cython, чтобы показать, как правильный выбор инструмента упрощает работу.

Все же существуют причины делать нечто, набивая шишки, используя только компилятор чистого C и заголовки `Python.h`. Во всяком случае, лучше всего применять такие инструменты, как Cython или Pyrex (о нем мы здесь не говорили), поскольку это сделает ваш код более читабельным и удобным в сопровождении. Это также избавит вас от большинства проблем, связанных с неосторожным подсчетом ссылок и распределением памяти.

Наше обсуждение расширений завершилось разговором о `ctypes` и CFFI, которые представляют собой альтернативные способы решения проблем интеграции общих библиотек. Поскольку они не требуют написания пользовательских расширений для вызова функций из скомпилированных бинарных файлов, именно эти инструменты лучше всего подходят для интеграции с закрытым кодом динамических/разделяемых библиотек, особенно если вам не нужно применять пользовательский код C.

В следующей главе мы слегка отдохнем от передовых методов программирования и углубимся в не менее важные темы — управление кодом и системы управления версиями.

Часть III
Качество, а не количество

В этой части рассматриваются различные процессы разработки, которые позволяют улучшить качество программного обеспечения и оптимизацию разработки в целом. Вы узнаете, как именно работать с кодом в системах управления версиями, документировать код и убедиться, что тестирование будет выполнено должным образом.

10 Управление кодом

Довольно трудно работать над программным проектом, если им занимается несколько человек. Когда состав команды увеличивается, работа словно замедляется и усложняется. Это происходит по многим причинам. В данной главе мы рассмотрим некоторые из них, а также поговорим о методах работы, направленных на улучшение совместной разработки кода.

Любая кодовая база со временем эволюционирует, и очень важно отслеживать все изменения, особенно если над ней трудятся много разработчиков. Для этого нужна *система управления версиями (система контроля версий)*.

Часто бывает, что несколько людей одновременно и параллельно дополняют кодовую базу в ходе работы. Было бы легче, если бы они имели разные роли и части в проекте. Но так бывает редко. Подобное отсутствие глобальной картины порождает много путаницы в отношении того, что происходит и кто чем занимается. Это неизбежно, ввиду чего нужно использовать инструменты для непрерывного улучшения видимости и смягчения проблем. Это делается путем создания ряда инструментов для непрерывной разработки, таких как *непрерывная интеграция* или *непрерывная доставка*.

В этой главе:

- работа с системой управления версиями;
- настройка процесса непрерывной разработки.

Технические требования

Скачать последнюю версию для этой главы можно по ссылке git-scm.com.

Работа с системой управления версиями

Системы управления версиями (version control systems, VCS) предоставляют возможность общего доступа, синхронизации и резервного копирования файлов лю-

бого типа, но в основном работают с текстовыми файлами, содержащими исходный код. Эти системы подразделяются на следующие два семейства:

- централизованные системы;
- распределенные системы.

В следующих подразделах рассмотрим эти семейства.

Централизованные системы

Централизованная система контроля версий — это один сервер с файлами, который позволяет пользователям вносить свои и видеть чужие изменения, внесенные в эти файлы. Принцип довольно прост — каждый может скопировать нужные файлы на свой компьютер и работать над ними. После этого каждый пользователь может *внести* на сервер изменения. Они будут применены, после чего генерируется номер *версии*. Другие пользователи смогут получить измененные файлы путем *синхронизации* их копий проектов через механизм *обновления*.

Как показано на схеме ниже (рис. 10.1), репозиторий обновляется с каждой посылкой, а система архивирует все изменения в базу данных, позволяя откатить любые изменения, а также предоставить информацию о том, что и кем было сделано.

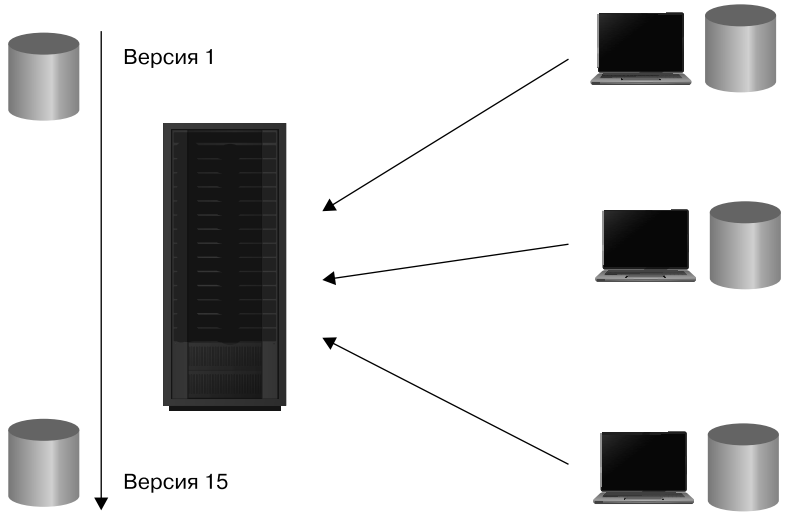


Рис. 10.1. Схема репозитория 1

Каждый пользователь в такой централизованной конфигурации должен синхронизировать свой локальный репозиторий с основным, чтобы своевременно

получить изменения других пользователей. Это означает потенциальное возникновение конфликтов, когда измененный вами файл был изменен кем-то еще. Тогда подключается механизм разрешения конфликтов, в данном случае в системе пользователя, как показано на следующей схеме (рис. 10.2).

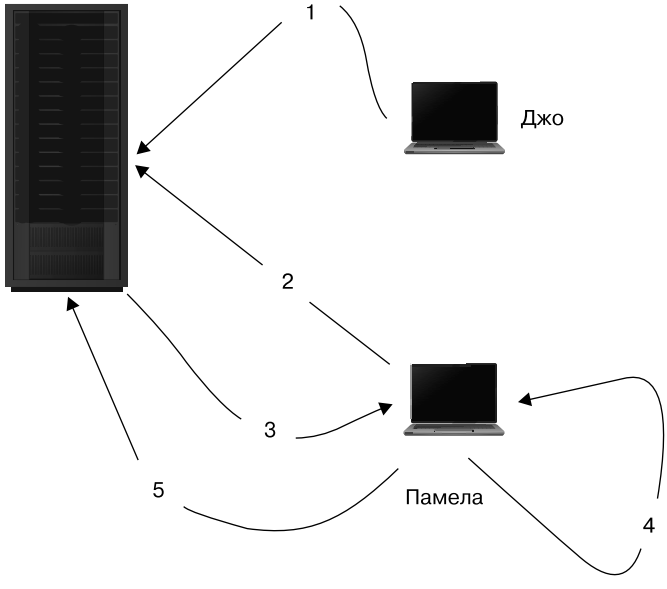


Рис. 10.2. Схема репозитория 2

Следующие шаги помогут вам лучше понять этот процесс:

- Джо вносит изменения;
- Памела тоже пытается изменить тот же файл;
- сервер говорит, что ее копия файла уже устарела;
- Памела обновляет свою локальную копию. Программное обеспечение контроля версий бесшовно объединяет (если получится) эти две версии;
- Памела отсылает новую версию, в которой есть последние изменения, сделанные и Джо, и ею самой.

Такой процесс прекрасно функционирует в проектах, над которыми трудятся несколько разработчиков и где используется небольшое количество файлов, но для более крупных проектов дело становится труднее. Например, сложные изменения часто затрагивают много файлов, а это отнимает много времени, и хранить у себя локальную копию всего проекта становится просто невозможно. Ниже приведены некоторые проблемы описанного выше подхода:

- пользователь может долго работать только в своей локальной копии без надлежащего резервного копирования;
- тяжело делиться с другими своей работой, пока она не отправлена в систему, а делать это без проверки означает поставить под угрозу нормальную деятельность хранилища.

В централизованной VCS можно решить эту проблему с помощью *ответвлений* и *слияний*. От основного потока изменений могут отделяться ветви, которые затем снова сливаются в основной поток.

На следующей схеме (рис. 10.3) Джо начинает новую ветвь от версии 2, поскольку планирует поработать над новой функцией. Изменения подсчитываются в основном потоке и в отделенной ветви. Дойдя до версии 7, Джо закончил работу и внес свои изменения в ствол (основную ветвь). В этом случае часто требуется разрешение конфликтов.

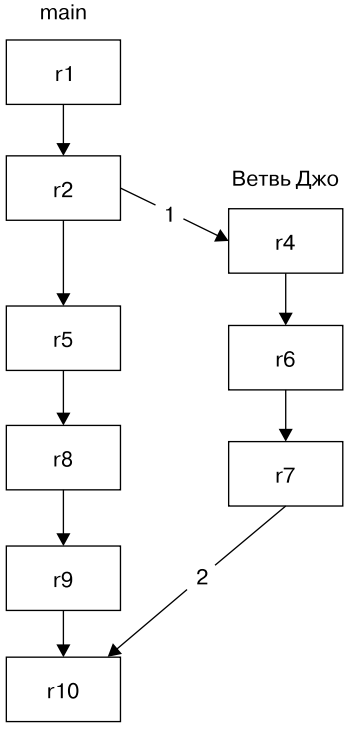


Рис. 10.3. Схема репозитория 3

Однако, несмотря на все преимущества, централизованная система VCS имеет следующие недостатки.

- Ветвление и слияние довольно трудно организовать. Это сложная система, которая может превратиться в кошмар.
- Поскольку система централизована, невозможно зафиксировать изменения в автономном режиме. В какой-то момент накопится большой и сложный пакет изменений.
- И наконец, подобная система не очень хорошо работает для таких проектов, как Linux, где у компаний есть собственный филиал программного обеспечения и нет центрального хранилища, в котором у каждого была бы учетная запись.

Что касается последнего, некоторые инструменты, такие как SVK, позволяют функционировать в автономном режиме, но более фундаментальная проблема заключается в самом принципе работы централизованной VCS.

Несмотря на эти ловушки, централизованные VCS по-прежнему весьма популярны у многих компаний, в основном за счет высокой инертности корпоративных сред. В качестве примеров централизованных VCS, используемых многими организациями, можно привести *Subversion (SVN)* и *System Concurrent Version (CVS)*. Очевидные проблемы централизованной архитектуры систем управления версиями привели к тому, что большинство сообществ, работающих с открытым кодом, перешли на более надежную *распределенную архитектуру VCS (DVCS)*.

Распределенные системы

Распределенные VCS призваны скомпенсировать недостатки централизованных VCS. Они располагаются не на главном сервере, с которым работают люди, а на равноправных (peer-to-peer) узлах. Каждый пользователь применяет собственный независимый репозиторий для проекта и синхронизирует его с другими репозиториями, как показано на следующей схеме (рис. 10.4).

На этой схеме показан пример использования такой системы.

- Билл *выгружает* файлы из хранилища HAL.
- Билл вносит в эти файлы некоторые изменения.
- Амина *выгружает* файлы из хранилища Билла.
- Амина тоже вносит изменения.
- Амина *загружает* изменения в HAL.
- Кенни *выгружает* файлы из HAL.
- Кенни вносит изменения.
- Кенни регулярно *отправляет изменения* в HAL.

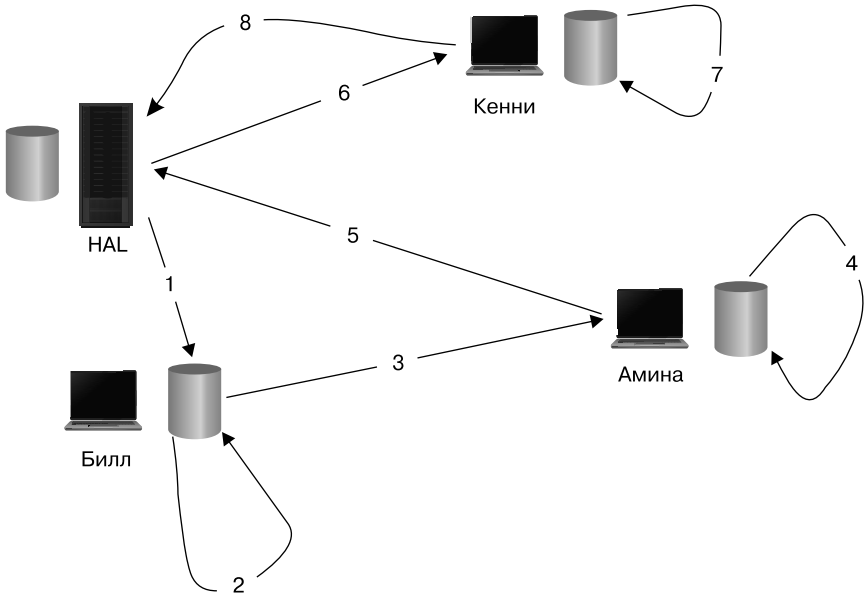


Рис. 10.4. Распределенная система

Главным здесь является принцип, по которому люди *выгружают* файлы в другие репозитории и *загружают* из них, и он меняется в зависимости от того, как организованы работа людей и управление проектом. Поскольку никакого «главного» репозитория больше нет, менеджеру проекта необходимо определить стратегию этой *выгрузки* и *загрузки* и *внесения изменений*.

Кроме того, работающие с несколькими репозиториями люди вынуждены больше думать. Во многих распределенных системах управления версиями номера версий определяются локально для каждого хранилища, и не существует глобальных номеров, с которыми можно сверяться. Таким образом, нужно задействовать специальные *теги*, чтобы работа стала яснее. Теги — текстовые метки, которые могут быть присоединены к версии. Наконец, пользователи сами отвечают за резервное копирование собственных репозиториев, в то время как в централизованной инфраструктуре разработкой стратегий резервного копирования занимается администратор.

В следующем подразделе поговорим о распределенных стратегиях.

Распределенные стратегии

Даже при использовании распределенной системы желательно иметь центральный сервер, если вы работаете с другими людьми. Но его назначение будет совершенно не тем, что у централизованных VCS. Этот сервер представляет собой хаб, который

позволяет всем разработчикам одновременно использовать свои изменения в одном месте, а не заниматься загрузкой и выгрузкой друг у друга. Такой единый центральный репозиторий (часто называемый также *вышестоящим*) служит в качестве резервного для всех изменений, выполняемых в отдельных репозиториях всех членов команды.

Для объединения доступа к коду с центральным репозиторием в DVCS используются различные подходы. Самый простой из них заключается в создании сервера, который выполняет функцию обычного централизованного сервера, и каждый участник проекта может вносить свои изменения в общий поток. Но такой подход несколько простоват и не позволяет получить максимум преимуществ распределенной системы, поскольку работа в конечном итоге будет такой же, как и с централизованной системой.

Другой подход заключается в создании на сервере нескольких репозиториев с различными уровнями доступа:

- *нестабильный репозиторий* — сюда каждый может вносить изменения;
- *стабильный репозиторий* — открыт только для чтения для всех участников, за исключением менеджеров. Они могут выгружать изменения из нестабильного репозитория и решать, что нужно объединять;
- *релизный репозиторий* — для релизов, только для чтения.

Такой подход позволяет участникам вносить свой вклад, а менеджерам — просматривать изменения, прежде чем вносить их в стабильный репозиторий. Однако, в зависимости от используемых инструментов, подобный подход может оказаться затратным. Во многих распределенных системах контроля версий эта проблема также решается надлежащей стратегией ветвления.

Сравним централизованные и распределенные системы управления версиями.

Централизованность или распределенность

Забудьте о централизованных системах управления версиями. Поговорим начистоту: централизованные системы управления версиями — пережиток прошлого. Сегодня, когда большинство из нас может работать полный рабочий день удаленно, неразумно ограничиваться всеми недостатками централизованных VCS. Например, при использовании CVS или SVN вы не можете отслеживать изменения в автономном режиме. Это глупо.

А как тогда поступить, если у вас временно пропало подключение к Интернету или «упал» сам репозиторий? И что теперь, просто забыть о рабочем процессе и не вносить изменения до тех пор, пока ситуация не изменится, а затем просто отправить большой объем неструктурированных обновлений? Нет!

Кроме того, большинство централизованных систем контроля версий не поддерживают эффективных схем ветвления версий — а это очень полезный метод, который позволяет снизить количество конфликтов при слиянии версий в проектах, где разработчики трудятся над одними и теми же частями проекта. Ветвление в SVN организовано настолько глупо, что большинство разработчиков старается любой ценой избегать его. Вместо этого в большинстве централизованных VCS есть механизм блокировки файлов, который несет больше вреда, чем пользы.

Печальная правда о каждом инструменте управления версиями (да и о программном обеспечении в целом) заключается в том, что если в ней есть какая-нибудь рискованная функция, то кто-то в вашей команде обязательно начнет использовать ее каждый день. Блокировка — как раз такая функция, которая в обмен на снижение количества конфликтов слияния резко снизит производительность всей вашей команды. Выбрав систему управления версиями, не позволяющую вести подобные рабочие процессы, вы создадите среду, которую ваши разработчики, вероятно, будут использовать эффективно.

В следующем подразделе поговорим о распределенной системе управления версиями Git.

По возможности используйте Git

Git — самая популярная распределенная система управления версиями. Она была создана Линусом Торвальдсом с целью сохранения версий ядра Linux, когда разработчикам ядра потребовалось отказаться от патентованного программного обеспечения BitKeeper, которое они использовали ранее.

Если вы еще не пробовали никаких систем контроля версий, то стоит начать с Git. В случае использования каких-либо других инструментов для управления версиями с Git все равно стоит познакомиться, даже при условии, что ваша организация не желает переключаться на Git в ближайшем будущем. Иначе вы рискуете стать живым ископаемым.

Однако мы не говорим, что Git — идеальная и лучшая в мире DVCS. Безусловно, она имеет свои недостатки. Прежде всего, это довольно сложный в использовании инструмент, особенно для новичков. Крутая кривая обучения Git уже стала источником множества шуток в Интернете. Наверняка есть системы управления версиями, которые для определенных проектов работали бы лучше, а полный список конкурентов Git с открытым исходным кодом будет довольно длинным. Во всяком случае, Git в настоящее время — самая популярная DVCS, поэтому *сетевой эффект* явно работает в ее пользу.

Если говорить кратко, то сетевой эффект означает, что совокупная выгода от использования популярных инструментов больше, чем от других, даже при наличии определенно лучшей альтернативы. Это связано именно с высокой популярностью

инструмента (так в свое время VHS уничтожил Betamax). Весьма вероятно, что кто-то в вашей организации, а также новые сотрудники будут иметь некий опыт работы с Git, вследствие чего стоимость интеграции именно этой системы DVCS будет ниже, чем в случае с менее популярной системой.

Ну и последний аргумент — всегда хорошо узнавать что-то еще и знакомство с другими DVCS вам точно не повредит. Самые популярные конкуренты Git с открытым исходным кодом — это Mercurial, Bazaar и Fossil. Первая система особенно хороша, поскольку написана на Python и была официальной системой управления версиями в исходниках CPython. Существуют некоторые признаки того, что в ближайшем будущем они могут перейти на другую систему, поэтому разработчики CPython, возможно, переходят на Git в ту самую секунду, пока вы читаете данную книгу. Но на самом деле это не имеет значения. Обе системы достаточно круты. Если бы не было Git или она была менее популярна, то мы бы определенно рекомендовали Mercurial. В ее устройстве есть своя особая красота. Она, безусловно, не так эффективна, как Git, но ее намного проще освоить начинающим.

Теперь поговорим о работе с GitFlow и GitHub.

Рабочий процесс GitFlow и GitHub Flow

Очень популярная и стандартизированная методика работы с Git называется просто GitFlow. Ниже приведено краткое описание основных правил рабочего процесса.

- Существует основная рабочая ветвь, как правило называемая `develop`, где происходит вся разработка последней версии приложения.
- Новые функции проекта реализуются в отдельных ветвях, называемых *функциональными*, которые всегда начинаются от ветви `develop`. Когда работа по функции будет завершена и код надлежащим образом проверен, новая ветвь объединяется с `develop`.
- Когда код в `develop` стабилизируется (без известных ошибок) и возникает потребность в новой версии приложения, создается новая *релиз-ветвь*. Для этой ветви обычно делаются дополнительные тесты (обширные испытания по оценке качества, интеграционные тесты и т. д.), так что, безусловно, вы найдете новые ошибки. Если потребуются дополнительные изменения (например, исправление ошибок), то они в конечном счете должны быть перенесены и в `develop`.
- Когда код в *релиз-ветви* готов к выпуску, она объединяется с ветвью `master` и последняя посылка в `master` помечается соответствующим тегом версии. Никакие другие ветви, кроме функций, не могут быть объединены с `master`. Исключение составляют лишь хотфиксы, которые необходимо развернуть или выпустить немедленно.
- Хотфиксы, которые требуют срочного выпуска, всегда реализуются на отдельных ветвях, отходящих от `master`. Когда исправление сделано, оно объединяется

с ветвями `develop` и `master`. Слияние ветви хотфиксов выполняется так, как если бы это была обычная релиз-ветвь, поэтому она должна быть корректно промаркирована и получить тег версии.

Визуальная схема рабочего процесса GitFlow представлена ниже (рис. 10.5). Тем, кто никогда не работал таким образом или никогда не использовал распределенную систему управления версиями, она может показаться сложноватой. Во всяком случае, вам стоит попробовать ее в вашей организации, если у вас нет какого-либо формализованного процесса. Такая система имеет множество преимуществ и позволяет решать серьезные проблемы. Это особенно полезно для команд, в которых несколько программистов работают над несколькими отдельными функциями и когда требуется постоянная поддержка нескольких релизов.

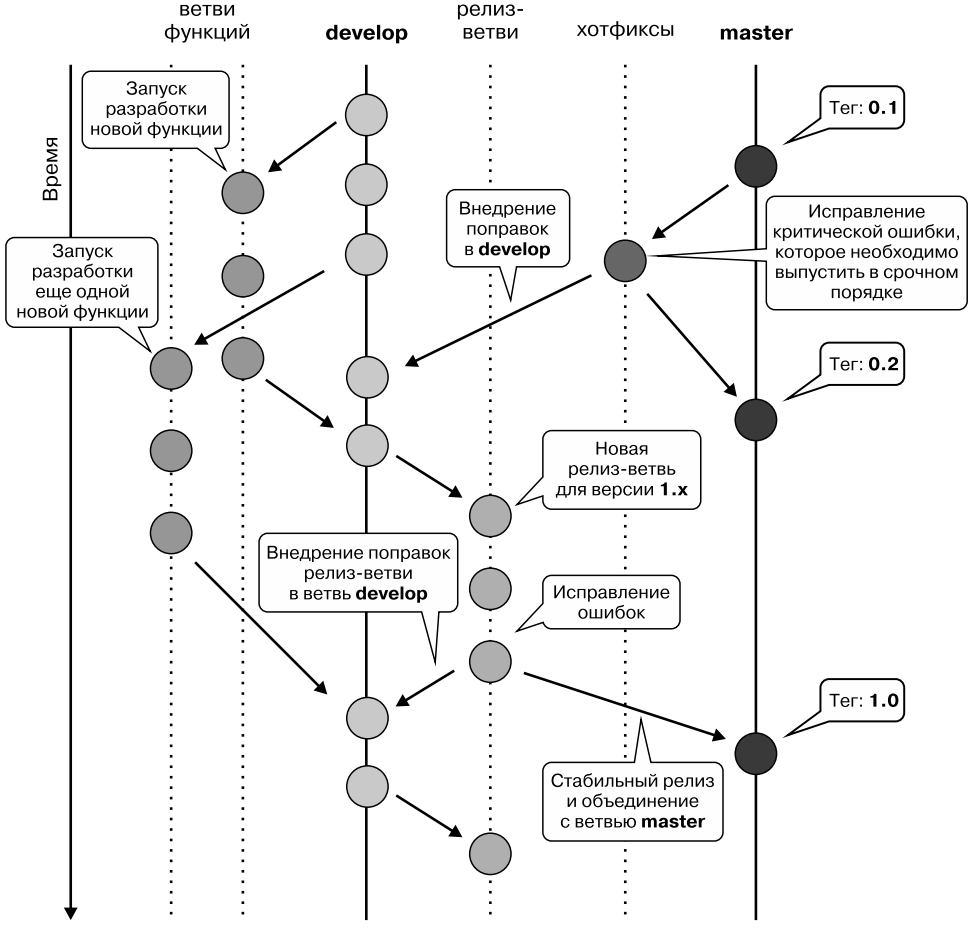


Рис. 10.5. Рабочий процесс GitFlow

Данная методика также удобна, если вы хотите осуществить непрерывную доставку с помощью непрерывных процессов развертывания, поскольку становится ясно, какая версия кода представляет собой доставляемый релиз вашего приложения или сервиса. Кроме того, это отличный инструмент для проектов с открытым исходным кодом, поскольку обеспечивает большую прозрачность для пользователей и активных участников.

Итак, если это краткое описание GitFlow вас не напугало и хоть что-то понятно, то следует копнуть глубже в онлайн-ресурсах по данной теме. Трудно сказать, кто автор описанного выше рабочего процесса, но большинство интернет-источников указывает на Винсента Дрессена. Таким образом, почитать о GitFlow лучше всего в его статье под названием *A successful Git branching model* (nvie.com/posts/a-successful-git-branching-model/).

Как и любая другая популярная методика, GitFlow получила много критики в Интернете от программистов, которым не понравилась. Больше всего комментариев получило правило (строго техническое) о том, что при каждом слиянии должна создаваться новая искусственная посылка, содержащая результат слияния. В Git есть возможность делать *быстрые слияния*, но Винсент не одобряет ее. Получается неразрешимая проблема, поскольку лучший способ выполнить слияние — дело вкуса. Во всяком случае, реальная проблема схемы GitFlow заключается в том, что она весьма сложная. Полный набор правил весьма велик, и поэтому в них легко сделать ошибку. Вполне вероятно, вам захочется выбрать некий более простой процесс.

Один из таких процессов используется в GitHub и описан Скоттом Чаконом в его блоге (scottchacon.com/2011/08/31/github-flow.html). Он называется GitHub Flow и очень похож на GitFlow в следующих двух основных аспектах:

- все, что находится в ветви `master`, развертывается;
- новые функции реализованы в отдельных ветвях.

Основное отличие этой схемы от GitFlow — простота. Существует только одна главная ветвь `master`, и она всегда стабильна (в отличие от `develop` в GitFlow). В схеме нет релиз-ветвей и фанатичного увлечения тегами в коде. Все дело в том, что, когда какая-то ветвь сливается с `master`, она обычно сразу развертывается в продакшене. Схема с примером GitHub Flow приведена ниже (рис. 10.6).

GitHub Flow выглядит как хороший и облегченный рабочий процесс для команд, которые хотят реализовать в проекте непрерывный процесс развертывания. Такой рабочий процесс, конечно же, будет нежизнеспособен для любого проекта, в котором с релизами все строго (со строгими номерами версий), по крайней мере без каких-либо модификаций. Важно знать, что основное предположение *всегда развертываемой* ветви `master` не может быть обеспечено без надлежащего автоматизированного тестирования и сборки процедуры. Эти проблемы решают системы непрерывной интеграции, и мы поговорим о них чуть позже.

Ниже представлена схема рабочего процесса GitHub в действии.

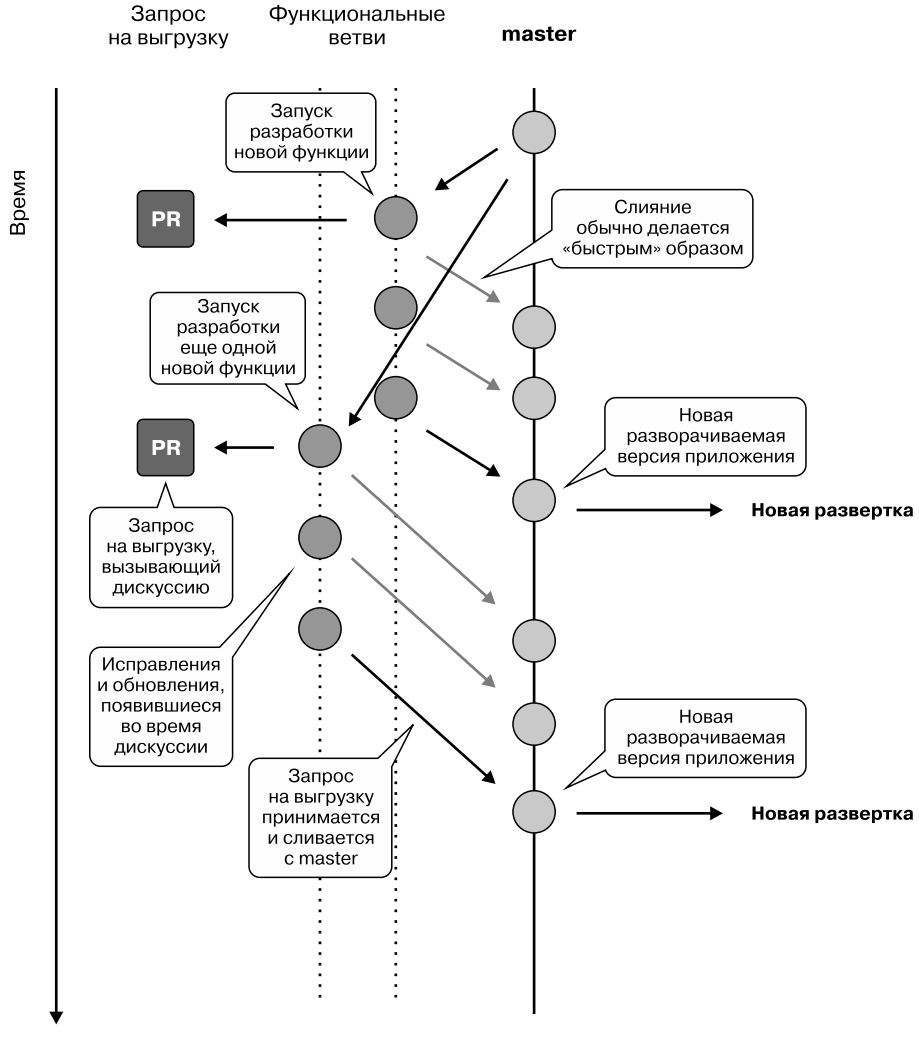


Рис. 10.6. Визуальное представление GitHub Flow

Обратите внимание: рабочие процессы GitFlow и GitHub Flow — просто стратегии разветвления, и потому, несмотря на наличие слова *Git* в их названиях, эти стратегии не ограничиваются только DVCS. Не секрет, что в официальной статье, описывающей GitFlow, даже упоминается конкретный параметр `git`, который должен использоваться при выполнении слияния, но в целом данную идею можно легко применить почти к любой другой распределенной системе управления версиями.

Именно из-за своего принципа работы со слияниями Mercurial кажется лучшим инструментом для данной конкретной стратегии ветвления! То же самое относится к GitHub Flow. Это единственная стратегия, в которой есть толика специфической культуры разработки, поэтому ее можно использовать в любой системе контроля версий, позволяющей легко создавать и объединять ветви кода.

И напоследок — помните: ни одна методика не идеальна и никто не заставляет вас использовать именно ту или другую. Разные методики созданы, чтобы решить некоторые из имеющихся проблем и уберечь вас от распространенных ошибок и ловушек. Вы можете придумать свои правила или изменить уже имеющиеся под собственные нужды. Это отличные инструменты для начинающих, которые иначе неизбежно будут совершать стандартные ошибки.

Если вы не знакомы с какими-либо системами управлениям версиями, то начать стоит с облегченной методологии наподобие GitHub Flow, не имеющей каких-либо пользовательских модификаций. Думать о более сложных рабочих процессах имеет смысл только тогда, когда вы получите достаточный опыт взаимодействия с Git или любым другим инструментом по вашему выбору. По мере роста вашего мастерства вы в конечном итоге поймете: не существует идеального рабочего процесса, который подходит каждому проекту. То, что хорошо работает в одной организации, не обязательно будет так же хорошо работать в других.

Настройка процесса непрерывной разработки

Существуют процессы, которые могут значительно ускорить разработку и сократить время на подготовку приложения к выпуску или развертке в production-среде. Они часто имеют слово *«непрерывный»* в имени. В данном разделе мы обсудим наиболее важные и популярные из них. Важно подчеркнуть, что это строго технические процессы, практически не связанные с методологией управления проектами, хотя могут сильно влиять на них.

Ниже приведены наиболее важные процессы, о которых мы будем говорить:

- непрерывная интеграция;
- непрерывная доставка;
- непрерывное развертывание.

Порядок перечисления здесь имеет значение, поскольку каждый элемент — продолжение предыдущего. Непрерывное развертывание можно было бы рассматривать просто как изменение одного и того же процесса. Мы в любом случае обсудим каждый пункт в отдельности, ведь то, что для одной организации может показаться мелочью, для другой может иметь решающее значение.

Мы сказали, что это технические процессы. То есть их реализация сильно зависит от использования соответствующих инструментов. Основная идея каждого

из них достаточно проста, и вы можете создавать собственные инструменты непрерывной интеграции/доставки/развертывания, но лучше всего выбрать что-то давно созданное и испытанное. Таким образом, вы можете сосредоточиться на создании собственного продукта вместо создания цепи инструментов для непрерывной разработки.

В следующем подразделе поговорим о непрерывной интеграции.

Непрерывная интеграция

Непрерывная интеграция (continuous integration, часто используется сокращение CI) — процесс, который зависит от систем автоматизированного тестирования и управления версиями, позволяющий получить полностью автоматизированную интеграцию. Он может быть задействован совместно с централизованными системами контроля версий, однако на практике в полную силу используется только в момент применения распределенной системы управления версиями.

Настройка хранилища — первый шаг на пути к непрерывной интеграции, представляющий собой набор практик программирования, которые возникли из *экстремального программирования* (eXtremeProgramming, XP).

Первое и самое важное требование для осуществления непрерывной интеграции заключается в создании полностью автоматизированного рабочего процесса, который позволяет целиком протестировать приложение в данной версии и проверить ее на предмет технической корректности. Техническая корректность означает, что в программе нет известных ошибок и все функции работают ожидаемым образом.

Основная идея CI заключается в том, что перед слиянием версии с основной ветвью всегда должны выполняться тесты. Это достигается только с помощью официальных соглашений, принятых в команде разработчиков, однако практика показывает ненадежность такого подхода. Проблема заключается в том, что программисты, как правило, самоуверенны и не в состоянии критически оценивать свой код. Если непрерывная интеграция зиждется лишь на договоренности внутри команды, то это неизбежно приведет к краху, поскольку некоторые из разработчиков рано или поздно пропустят свой этап тестирования и выпустят в производство код с ошибками, и эти ошибки останутся там навечно. А опыт показывает, что даже простые изменения могут привести к критическим проблемам.

Очевидное решение — применение выделенного сервера сборки, который автоматически запускает в приложении все необходимые тесты при всяком изменении кодовой базы. Есть множество инструментов, позволяющих оптимизировать этот процесс, и они легко интегрируются с сервисами управления версиями, такими как GitHub или Bitbucket, и локальными сервисами наподобие GitLab. Преимущество использования подобных инструментов заключается в том, что разработчик может запустить локально только определенное подмножество тестов (которое, по его

мнению, непосредственно связано с его текущей работой), а потенциально трудозатратные наборы интеграционных тестов ложатся на плечи сервера сборки. Такой способ действительно ускоряет разработку, но при этом все еще существует риск того, что новые функции испортят стабильный код в основной ветви.

Еще одно преимущество использования выделенного сервера для сборки заключается в том, что можно запускать тесты в среде, максимально близкой к продакшену. Вообще, разработчики тоже должны применять такие среды, и для этого есть специальные средства (например, Vagrantand Docker), но в организации такой вариант осуществить трудно. Вы можете легко выполнить это на одном выделенном сервере сборки или даже на кластере серверов сборки. Многие инструменты CI еще сильнее упрощают процесс путем использования инструментов виртуализации и/или контейнеризации, которые позволяют гарантировать, что тесты всегда выполняются в одной новенькой, «с иголочки» среде тестирования.

Организовать себе сервер сборки обязательно, если вы создаете десктопные или мобильные приложения, которые доставляются пользователям в виде бинарных файлов. Очевидно, выполнять такую сборку всегда нужно в одной и той же среде. Почти все системы CI учитывают тот факт, что приложения часто следует скачивать в бинарном виде после тестирования и сборки. Такие результаты обычно называют *артефактами сборки*.

Поскольку инструменты CI возникли во времена, когда большинство приложений писалось на компилируемых языках, их работа описывается с помощью термина *«сборка»*. Для таких языков, как C или C++, это очевидно, поскольку приложения вообще не запускаются и не проверяются, если их не собрать (не скомпилировать). В Python все немного иначе, ввиду того что большинство программ распространяются в виде исходного кода и могут работать без дополнительной сборки. Таким образом, в рамках нашего языка термины *«тестирование»* и *«сборка»* часто используются как взаимозаменяемые в контексте темы непрерывной интеграции.

Тестирование каждой посылки

Лучший подход к непрерывной интеграции заключается в том, чтобы выполнять полный набор тестов после каждого изменения в центральном репозитории. Даже если один программист отправляет несколько посылок в одну ветвь, часто имеет смысл протестировать каждое изменение в отдельности. Решение тестировать только самое последнее изменение повлечет сложности в поиске источников возможных проблем, если они появились где-то в середине и тестов не проходили.

Конечно, многие DVCS, такие как Git или Mercurial, позволят вам ограничить время, затрачиваемое на поиск исходников для регрессии, с помощью команд, позволяющих взять *срез* истории изменений, однако на практике гораздо удобнее организовать все автоматически, как часть процесса непрерывной интеграции.

Бывают проблемы с проектами, в которых слишком большие тестовые наборы, и, как следствие, на тесты уходят десятки минут или даже часов. В этом случае один сервер может и не справиться со сборкой всех посылок, и тогда ждать результатов придется еще дольше. На самом деле длительность тестирования — сама по себе проблема, и об этом мы поговорим ниже, в пункте «Проблема 2 — слишком долгая сборка» подраздела «Выбор правильного инструмента и распространенные ошибки» данного раздела.

Ну а пока вам стоит знать, что по возможности всегда нужно проверять каждую посылку, отправляемую в репозиторий. Если вы не можете делать это на одном сервере, то настройте кластер для сборки. В случае использования платных сервисов стоит выбрать более дорогой тарифный план с большим количеством потенциальных параллельных сборок. Оборудование стоит недорого, а вот время ваших разработчиков — другое дело. В конце концов, вы сэкономите больше денег, организовав параллельную сборку и используя более дорогую систему CI, чем потратите в случае пропуска ошибок при недостаточных тестах.

Тестирование слияния через CI

В реальности все сложно. Если код в функциональной ветви прошел все тесты, то это еще не означает корректного функционирования сборки в стабильной основной ветви. Обе популярные стратегии ветвления, упомянутые в подразделе «Рабочий процесс GitFlow и GitHub Flow» предыдущего раздела предполагают, что код, объединяемый в ветви `master`, всегда можно протестировать и развернуть. Но как вы можете быть уверены, что данное предположение имеет место, если слияние еще не выполнялось? Для GitFlow это не такая большая проблема (при надлежащей реализации и использовании) в связи с акцентом на релизных ветвях. Но вот у простого GitHub Flow есть реальная проблема, поскольку слияние с `master` часто связано с конфликтами и может привести к регрессии в тестах. Даже для GitFlow это серьезная проблема. Это сложная модель ветвления, ввиду чего разработчики наверняка будут делать ошибки при ее использовании. Как следствие, вы никогда не можете быть уверены в том, что код в ветви `master` пройдет тесты после слияния, если не принять специальные меры предосторожности.

Одно из решений этой проблемы — делегирование полномочий по слиянию отдельных ветвей в стабильную ветвь системе CI. Во многих инструментах CI можно легко настроить сборку по требованию. Система будет локально выполнять слияние ветвей и отправлять результат в центральный репозиторий, если он прошел все тесты. При неудавшейся сборке такое слияние будет отозвано, оставив стабильную ветвь нетронутой.

Конечно, этот подход сильно усложняется в быстрорастущих проектах, где одновременно разрабатывается несколько ветвей, поскольку существует высокий

риск конфликтов, которые не могут решаться автоматически ни одной системой CI. Конечно, решение данной проблемы есть — например, *релиз* в Git.

Такой подход к слиянию в стабильную ветвь в системе управления версиями практически обязателен, если вы хотите двигаться дальше и реализовать процесс непрерывной доставки. Кроме того, он требуется при наличии в рабочем процессе строгого правила о том, что все в стабильной ветви должно быть готово к релизу.

Матричное тестирование

Матричное тестирование — очень полезный инструмент, если необходимо тестировать код в различных средах. В зависимости от потребностей проекта, в вашем решении CI может быть в той или иной мере нужна прямая поддержка такой функции.

Самый простой способ объяснить идею матричного тестирования — взять в качестве примера пару пакетов Python с открытым исходным кодом. Так, Django — проект, имеющий строго определенный набор поддерживаемых версий Python. В версии 1.9.3 в требованиях для запуска Django перечислены версии Python 2.7, 3.4 и 3.5. То есть каждый раз, когда разработчики Django вносят изменения в проект, в этих трех версиях Python нужно выполнить полный набор тестов, чтобы все работало корректно. Если хотя бы один тест в одной среде не срабатывает, то вся сборка должна быть помечена как неудачная, поскольку может быть нарушена обратная совместимость. Такой простой случай не требует поддержки CI. Существует отличный инструмент Tox (tox.readthedocs.org), который, помимо всего прочего, позволяет легко запускать тестовые наборы в разных версиях Python в изолированных виртуальных окружениях. Кроме того, эта утилита хорошо подходит для локальной разработки.

Но это был лишь простейший пример. Часто возникает необходимость протестировать приложение в нескольких средах, в которых при этом проверяются совершенно различные параметры, например:

- операционные системы;
- базы данных;
- версии сервисов;
- типы файловых систем.

Полный набор всех возможных комбинаций этих параметров образует многомерную матрицу параметров окружающей среды, и отсюда название — матричное тестирование. Если вам нужен такой глубокий процесс тестирования, то вполне возможно, что в вашем решении CI вам потребуется поддержка матричного тестирования. При большом количестве вероятных комбинаций вам понадобится сильно распараллеленный процесс сборки, поскольку каждый проход по матрице

потребует от сервера сборки значительных усилий. Возможно, вы будете вынуждены пойти на некий компромисс, если тестовая матрица окажется слишком многомерной.

В следующем подразделе мы поговорим о непрерывной доставке.

Непрерывная доставка

Непрерывная доставка — прямое продолжение идеи непрерывной интеграции. Такой подход к разработке программного обеспечения стремится гарантировать, что приложение можно выпустить в любой момент. Цель непрерывной доставки — выпуск программного обеспечения короткими по времени циклами. В целом это снижает и затраты, и риски во время выпуска программного обеспечения, поскольку изменения внедряются более плавно.

Ниже приведены обязательные требования для организации процесса непрерывной доставки:

- надежный процесс непрерывной интеграции;
- автоматизированный процесс развертывания в production-среде (если в проекте есть понятие такой среды);
- четкая система управления версиями или стратегия разветвления, которая позволяет легко определить, какая именно версия программного обеспечения отправляется в релиз.

Во многих проектах автоматизированных тестов может оказаться недостаточно, чтобы с уверенностью говорить о готовности той или иной версии ПО к релизу. В таких случаях нужны дополнительные приемочные тесты, которые, как правило, выполняются квалифицированными специалистами по оценке качества. В зависимости от вашей методологии управления проектами здесь может потребоваться некоторое одобрение со стороны клиента. Это вовсе не означает, что вы не можете использовать GitFlow, GitHub Flow или аналогичную стратегию ветвления, если некоторые из ваших приемочных тестов выполняются вручную. Это влияет только на формулировку — теперь ветвь будет не *готовой к развертке*, а *готовой к приемочным тестам и утверждению*.

Кроме того, все описанное не отменяет того факта, что развертывание кода всегда должно быть автоматизировано. Мы уже обсуждали инструменты и преимущества автоматизации в главе 8. Как заявлялось, подобный подход позволяет снизить затраты и риски, связанные с релизом. Вдобавок большинство из доступных инструментов CI позволяют создавать специальные целевые сборки, которые вместо тестирования будут выполнять для вас автоматическое развертывание приложения. В большинстве процессов непрерывной доставки оно обычно

инициируется вручную (по требованию) уполномоченными сотрудниками, если они уверены, что требуемое утверждение получено и все приемочные испытания увенчались успехом.

В следующем подразделе поговорим о непрерывном развертывании.

Непрерывное развертывание

Непрерывное развертывание — процесс, выводящий непрерывную доставку на качественно новый уровень. Это идеальный подход для проектов, где все приемочные испытания автоматизированы и нет необходимости в ручном утверждении чего бы то ни было. Когда код объединяется в стабильной ветви (обычно в ветви `master`), он автоматически развертывается в продакшене.

Подобный подход кажется удобным и надежным, но используется не слишком часто, поскольку довольно трудно найти проект, в котором перед выпуском новой версии совсем не требуется ручное тестирование и чье-то одобрение. Однако это вполне выполнимо, и некоторые компании утверждают, что работают именно так.

Реализация непрерывного развертывания потребует от вас тех же основных предпосылок, что и непрерывная доставка. Кроме того, часто нужен более осторожный подход к слиянию в стабильной ветви. То, что попадает в ветвь `master` при непрерывной интеграции, обычно сразу попадает в продакшен. Поэтому разумно переложить задачу слияния на плечи системы CI, как описано выше, в пункте «Тестирование слияния через CI».

В следующем подразделе описаны популярные инструменты для непрерывной интеграции.

Популярные инструменты для непрерывной интеграции

Сегодня существует огромное многообразие инструментов CI. Они сильно различаются по простоте использования, а также по ассортименту доступных функций, и почти каждый имеет ряд уникальных особенностей, отсутствующих у других. Из-за этого довольно трудно дать универсальную рекомендацию, поскольку у каждого проекта свои потребности, а также свой рабочий процесс разработки. Конечно, есть большие бесплатные проекты с открытым исходным кодом, но и платные сервисы тоже достойны внимания. Дело в том, что программное обеспечение с открытым исходным кодом, такое как Jenkins или BuildBot, свободно доступно для бесплатной установки, однако не стоит думать, что и работать оно будет бесплатно. Аппаратное обеспечение и техническое сопровождение увеличивает ваши затраты на содержание системы CI. В некоторых случаях может быть дешевле заплатить за такой сервис, вместо того чтобы содержать дополнительную инфраструктуру и тратить время на решение проблем в программном обеспечении CI с открытым исходным кодом. Однако всегда стоит убедиться, что отправка кода в сторонний сервис не нарушает политику безопасности вашей компании.

Здесь мы рассмотрим некоторые из наиболее популярных бесплатных инструментов с открытым исходным кодом, а также платные сервисы. Мы не хотим рекламировать конкретного поставщика, поэтому обсудим только бесплатные, чтобы сохранить хоть какую-то объективность. Указаний наподобие «что лучше» мы давать не будем, но укажем хорошие и плохие стороны каждого решения. Если вы все еще сомневаетесь, то следующий подраздел, в котором описываются популярные ловушки интеграции, должны помочь вам принять правильное решение.

Теперь поговорим о Jenkins.

Jenkins

Jenkins (jenkins-ci.org), похоже, самый популярный инструмент для непрерывной интеграции (рис. 10.7). Он также является одним из самых старых проектов с открытым исходным кодом в этой области, деля первенство с Hudson (проект в какой-то момент разделился, и Jenkins — ответвление Hudson).

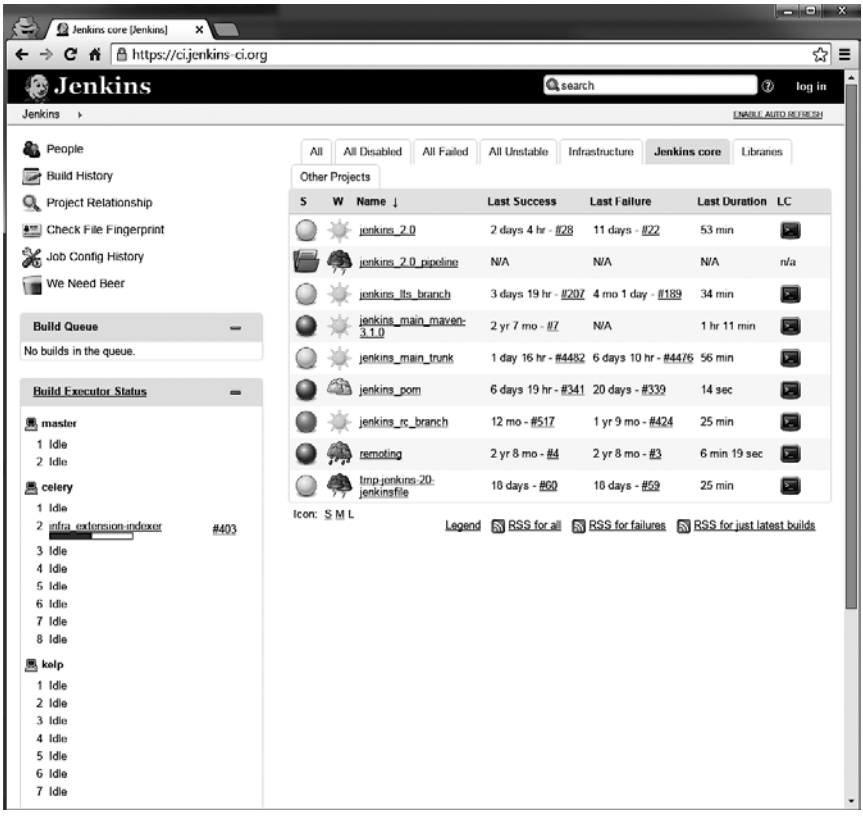
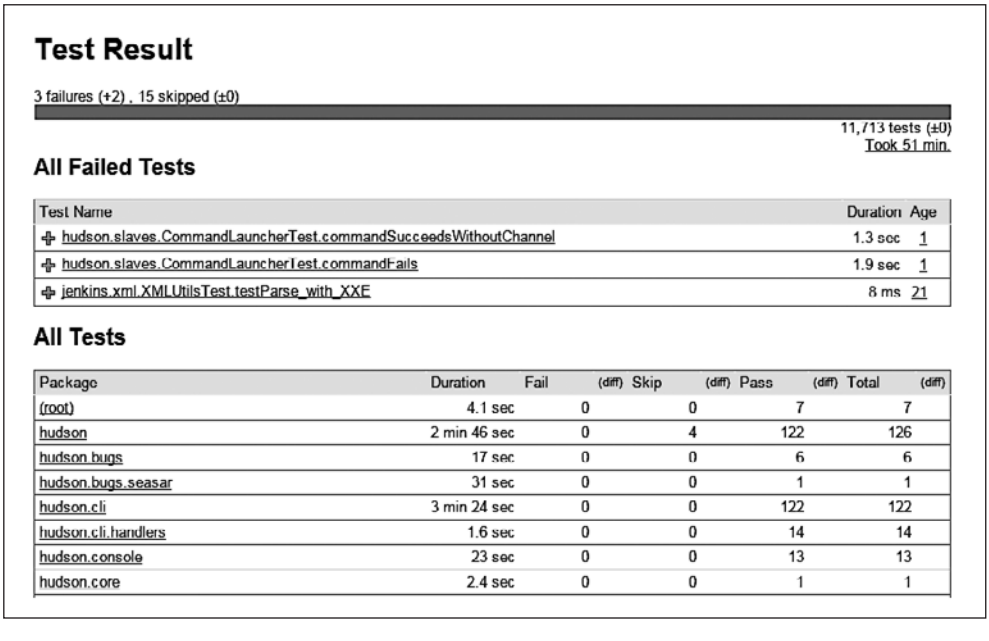


Рис. 10.7. Окно предпросмотра главного интерфейса Jenkins

Система Jenkins написана на Java и была первоначально разработана в основном для создания проектов, написанных также на Java. Это значит, что для разработчиков Java данная система CI идеальна, а если вы хотите использовать ее с другим стеком технологий, то придется слегка попотеть.

Серьезное преимущество Jenkins — огромный список функций, которые реализованы в ней прямо «из коробки». Наиболее важная из них, с точки зрения программиста Python, — способность понимать результаты теста. Вместо того чтобы давать только простую бинарную информацию об успехе сборки, Jenkins позволяет показать результаты всех тестов, которые были выполнены во время запуска, в виде таблиц и графиков. Такая система, конечно, не будет работать автоматически, поскольку вам нужно выводить результаты в определенном формате (по умолчанию Jenkins понимает файлы JUnit) во время сборки. К счастью, многие фреймворки тестирования Python позволяют экспортировать результаты в машинно-читаемом формате.

На рис. 10.8 приведен пример представления результатов тестов в Jenkins в веб-интерфейсе.



Test Result

3 failures (+2) , 15 skipped (±0)

11,713 tests (±0)
Took 51 min.

All Failed Tests

Test Name	Duration	Age
hudson.slaves.CommandLauncherTest.commandSucceedsWithoutChannel	1.3 sec	1
hudson.slaves.CommandLauncherTest.commandFails	1.9 sec	1
jenkins.xml.XMLUtilsTest.testParse_with_XXE	8 ms	21

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
(root)	4.1 sec	0		0		7		7	
hudson	2 min 46 sec	0		4		122		126	
hudson.bugs	17 sec	0		0		6		6	
hudson.bugs.seasar	31 sec	0		0		1		1	
hudson.cli	3 min 24 sec	0		0		122		122	
hudson.cli.handlers	1.6 sec	0		0		14		14	
hudson.console	23 sec	0		0		13		13	
hudson.core	2.4 sec	0		0		1		1	

Рис. 10.8. Представление результатов тестирования модуля в Jenkins

На рис. 10.9 показано, как Jenkins представляет дополнительную информацию о сборке, например тренды и скачиваемые артефакты.

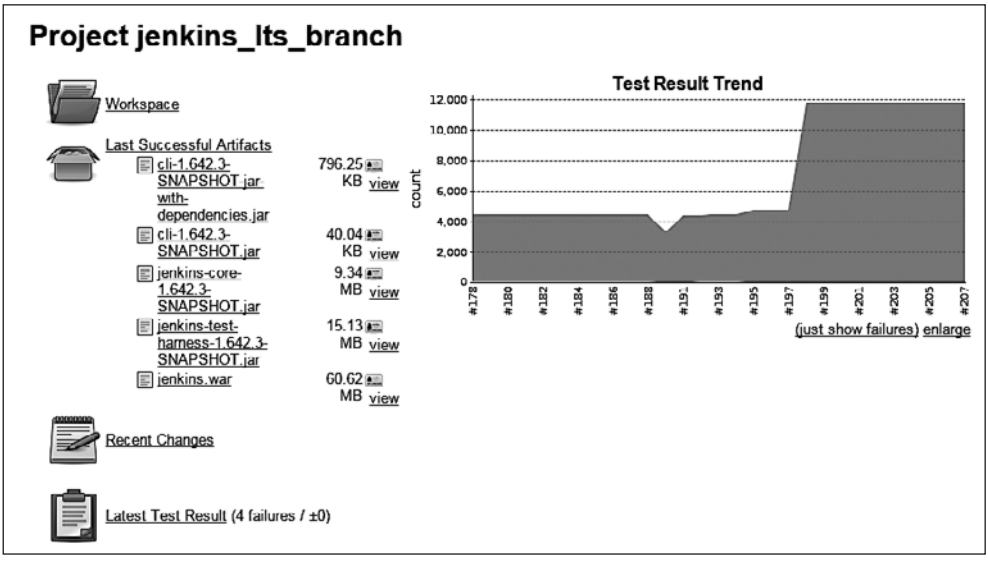


Рис. 10.9. График трендов тестирования примера проекта Jenkins

Удивительно, но большая часть эффективности Jenkins заключается не во встроенных функциях, а в огромном количестве бесплатных плагинов. Даже «из коробки» набор довольно широк для разработчиков Java, но программистам, использующим другие технологии, потребуется много времени, чтобы подогнать все под свой проект. Даже поддержка Git обеспечивается через плагин.

Эта расширяемость Jenkins весьма хороша, но имеет и серьезные недостатки. Вы станете зависеть от установленных плагинов, на которых держится весь процесс непрерывной интеграции, а они разрабатываются обособленно от ядра Jenkins. Большинство авторов популярных плагинов стараются регулярно обновлять их и поддерживать совместимость с последними версиями Jenkins. Однако если у расширения слабое сообщество, то оно будет обновляться реже, и в один прекрасный день придется либо отказаться от расширения, либо отложить обновление вашей системы. Это может стать реальной проблемой при необходимости срочно выпустить обновление (исправление безопасности, например), а ключевые плагины вдруг перестанут работать с новой версией.

Базовая установка Jenkins, в которой есть главный сервер CI, также способна выполнять сборки. Этим она отличается от других систем CI, делающих больший акцент на распространении и четко разделяющих главный и подчиненный серверы. Это и хорошо и плохо. С одной стороны, возникает возможность создать полностью работающий сервер CI за считаные минуты. Jenkins, конечно же, поддерживает перенос работы на подчиненные серверы, и вы можете масштабировать

ее в будущем, когда это необходимо. С другой стороны, Jenkins часто бывает неэффективной, поскольку развернута в настройках отдельных серверов, а пользователи жалуются на производительность, так как не получают достаточно ресурсов. Однако добавить новые узлы сборки в кластер Jenkins нетрудно. Для привыкших к одиночному серверу это скорее задачка для ума, нежели техническая проблема.

Далее рассмотрим BuildBot.

BuildBot

BuildBot (buildbot.net) — программное обеспечение, написанное в Python, которое позволяет автоматизировать компиляцию и тестирование любых программных проектов. Оно настраивается таким образом, что все изменения, сделанные в репозитории исходного кода, генерируют сборку и запускают несколько тестов, а затем обеспечивает обратную связь (рис. 10.10).

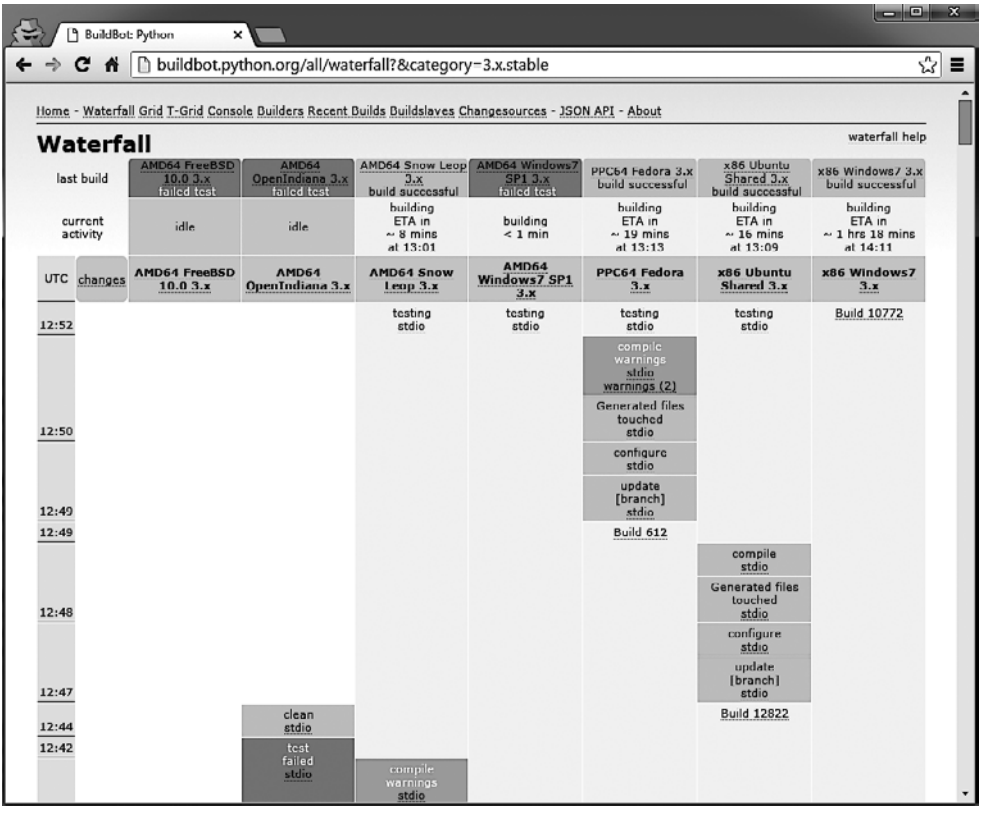


Рис. 10.10. Окно Waterfall в BuildBot для ветви CPython 3.X

Этот инструмент используется, например, в ядре CPython, и его можно найти по адресу buildbot.python.org/all/waterfall?&category=3.x.stable.

Представление сборок по умолчанию в BuildBot — окно, показанное на рис. 10.10. Каждый столбец соответствует *сборке*, состоящей из *шагов* и связанной с некоторыми *ведомыми узлами сборки*. Вся система управляется главным узлом таким образом:

- главный узел сборки централизует и запускает все процессы;
- сборка представляет собой последовательность шагов, используемых для сборки приложения и их тестирования;
- *шаг* — минимальная команда. Обычно это следующие действия:
  - проверка файлов проекта;
  - сборка приложения;
  - запуск тестов.

Ведомый сборщик — механизм, который отвечает за запуск сборки. Он может быть расположен в любом месте, лишь бы имелась связь с ведущим (мастером). Благодаря этой архитектуре BuildBot обретает очень хорошую масштабируемость. Все тяжелая работа выполняется ведомыми сборщиками, и вы можете завести их сколько угодно.

Весьма простой и понятный дизайн делает BuildBot очень гибким инструментом. Каждый шаг сборки выполняется одной командой. BuildBot написан на Python, но хорошо работает с любым языком. Итак, шаг сборки может быть абсолютно любым. Для принятия решения об успешном завершении шага используется код завершения процесса, и весь стандартный вывод шага захватывается по умолчанию. Большинство инструментов для тестирования и компиляторов работают по передовым принципам дизайна, указывают на ошибки с соответствующими кодами завершения и возвращают читаемые ошибки и предупреждающие сообщения в потоки `stdout` или `stderr`. В противном случае вы обычно можете легко обернуть их простым скриптом Bash. В большинстве случаев это довольно простая задача. Таким образом, многие проекты можно интегрировать с BuildBot при минимальных усилиях.

Еще одно преимущество BuildBot — он поддерживает следующие системы управления версиями прямо «из коробки», не прибегая к необходимости устанавливать какие-либо дополнительные плагины:

- CVS;
- Subversion;
- Perforce;
- Bzr;

- Darcs;
- Git;
- Mercurial;
- Monotone.

Основной недостаток BuildBot заключается в отсутствии инструментов для представления результатов сборки более высокого уровня. К примеру, у других проектов, таких как Jenkins, есть возможность отображать тесты, запускаемые в процессе сборки. Если вы дадите системе данные результатов тестов в соответствующем формате (обычно XML), то они могут представить все тесты в читабельном виде, то есть в виде таблиц и графиков. В BuildBot нет такой встроенной функции, и это цена гибкости и простоты. Если вам нужны дополнительные излишества, то их придется делать самостоятельно или искать какие-то пользовательские расширения. С другой стороны, такая простота делает поведение BuildBot предсказуемым и поддержка упрощается. Во всем нужно искать компромисс.

Далее поговорим о Travis CI.

Travis CI

Travis CI (travis-ci.org) — система непрерывной интеграции, продаваемая в виде «ПО как сервис». Это платный сервис для предприятий, но его можно использовать бесплатно в проектах с открытым исходным кодом, размещенных на GitHub (рис. 10.11).

Естественно, именно бесплатность сделала данный сервис очень популярным. В настоящее время это одно из самых популярных решений CI для проектов, размещенных на GitHub. Но самое большое преимущество, по сравнению со старыми проектами, такими как BuildBot или Jenkins, заключается в методе хранения конфигурации сборки. Все определение сборки содержится в одном файле `.travis.yml` в корне репозитория проекта. Travis работает лишь с GitHub, поэтому если вы включили такую интеграцию, то ваш проект будет тестироваться при каждой посылке только при наличии файла `.travis.yml`.

Иметь всю конфигурацию CI для проекта в репозитории кода — действительно удобно. Это делает весь процесс для разработчиков намного понятнее, а также дает неплохую гибкость. В системах, где для создания должны быть предусмотрены отдельные конфигурации сборки (с помощью веб-интерфейса или через конфигурацию сервера), всегда возникают сложности, когда в тестовую схему нужно добавить нечто новое. В некоторых организациях, где только избранные сотрудники имеют право работать с системой CI, это действительно замедляет процесс добавления новых шагов сборки. Кроме того, иногда возникает необходимость проверять различные ветви кода совершенно разными процедурами. Когда конфигурация сборки доступна в исходных файлах, сделать это становится намного проще.

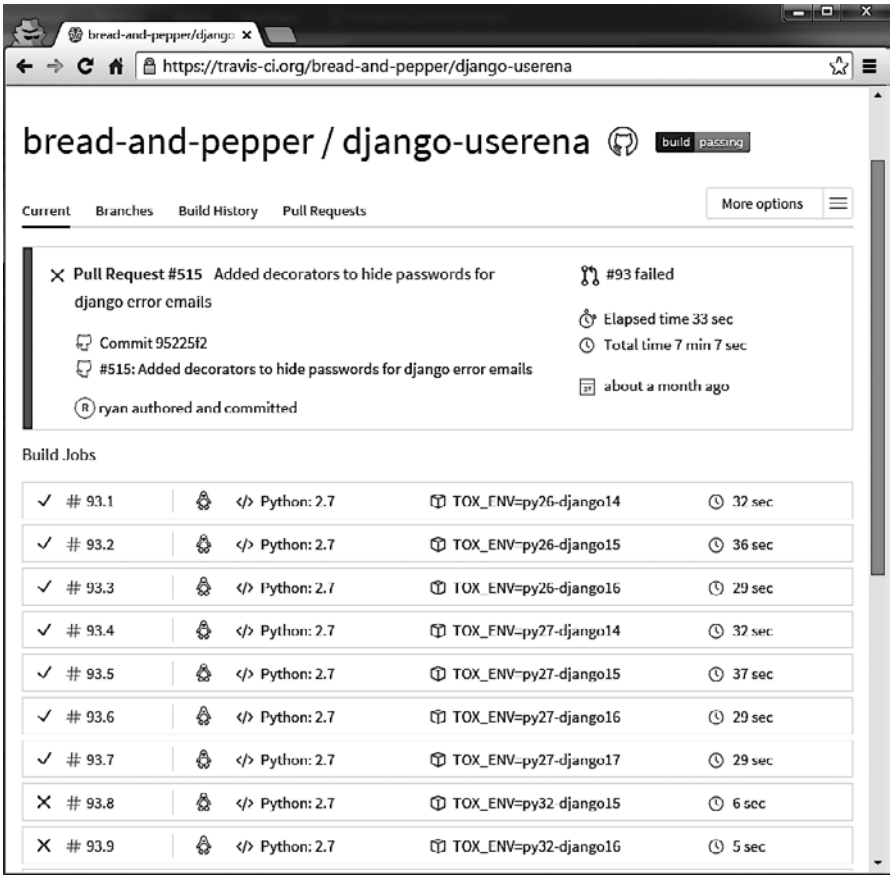


Рис. 10.11. Страница проекта в Travis CI, где показываются неудавшиеся сборки в матрице сборок

Еще одна особенность Travis — акцент на выполнении сборок в чистой среде. Каждая сборка выполняется на полностью новой виртуальной машине, поэтому не возникает никакого риска влияния чего-либо на результаты сборки. В Travis используется довольно большой образ виртуальной машины, ввиду чего вам доступно много программного обеспечения с открытым исходным кодом и сред программирования и нет надобности в дополнительной установке. В этой изолированной среде у вас есть полные права администратора, поэтому вы можете скачивать и устанавливать все необходимое для сборки, и синтаксис файла `.travis.yml` все упрощает. К сожалению, ваш выбор операционных систем, используемых в качестве базы среды тестирования, ограничен. Travis не позволяет вам создавать собственные образы виртуальных машин, так что придется полагаться

на очень ограниченное количество опций. Обычно этот выбор и вовсе отсутствует и сборки приходится делать в какой-то версии Ubuntu, macOS или Windows (все еще экспериментальной на момент написания книги). Иногда вы можете выбрать несколько устаревшую версию в качестве «пробника» новой среды тестирования, но это всегда временная мера. Всегда существует способ обойти ее. Вы можете запустить другую виртуальную машину или контейнер внутри Travis. Тогда у вас должно быть нечто позволяющее легко кодировать конфигурацию VM в исходных файлах, например Vagrant или Docker. Но это увеличит время создания сборки, и потому подобный подход не самый лучший. Использование виртуальных машин, таким образом, не может быть лучшим и эффективным подходом в случае необходимости выполнять тесты в разных операционных системах. Если вам нужно работать именно так, то Travis вам не подходит.

Самый большой недостаток Travis — он полностью заблокирован в GitHub. Если вы хотели бы использовать этот инструмент в проекте с открытым исходным кодом, то данный недостаток не имеет большого значения. А вот для предприятий и закрытых проектов это неразрешимая проблема.

Далее рассмотрим GitLab CI.

GitLab CI

GitLab CI — часть более крупного проекта GitLab. Инструмент доступен в виде платного сервиса (Enterprise Edition) и в виде проекта с открытым кодом, который вы можете разместить в вашей собственной инфраструктуре (Community Edition). В издании с открытым кодом не хватает некоторых функций платного сервиса, но для большинства случаев его функций достаточно, если компании нужно программное обеспечение, управляющее версиями через репозитории и системы непрерывной интеграции.

GitLab CI по своему набору функций очень похож на Travis. В нем даже заложен весьма похожий синтаксис YAML, который хранится в файле `.gitlab-ci.yml`. Самая большая разница заключается в том, что модель ценообразования GitLab Enterprise Edition не предусматривает бесплатных аккаунтов для проектов с открытым исходным кодом. Community Edition открыта сама по себе, но вам требуется собственная инфраструктура, чтобы запустить ее.

По сравнению с Trevis, GitLab имеет очевидное преимущество в виде большего контроля над средой выполнения. К сожалению, с точки зрения изоляции среды сборщик по умолчанию в GitLab немного слабоват. Процесс GitLab Runner выполняет все этапы сборки в той же среде, в которой запущен, и в этом смысле больше похож на ведомые сборщики Jenkins или BuildBot. К счастью, инструмент хорошо работает с Docker, так что вы можете легко добавить больше изоляции с помощью контейнерной виртуализации, но это потребует дополнительных усилий и настроек.

Выбор правильного инструмента и распространенные ошибки

Как было сказано ранее, не существует идеального инструмента CI, который подошел бы каждому проекту и, что самое главное, любой организации и рабочему процессу. Мы можем дать рекомендацию по проектам с открытым исходным кодом, размещенным на GitHub. Для небольших проектов с открытым исходным кодом, не зависящих от платформы, Travis CI кажется лучшим выбором. Работу с ним легко начать, и он принесет вам почти мгновенное удовлетворение с минимальными трудозатратами.

У проектов с закрытым кодом ситуация совершенно иная. Вполне возможно, что вам нужно будет попробовать несколько систем CI в различных конфигурациях, прежде чем вы определитесь, какая из них подходит лучше. Мы обсудили лишь четыре популярных инструмента, но и эта выборка довольно репрезентативна. Чтобы облегчить процесс принятия решения, мы обсудим часть наиболее распространенных проблем, связанных с непрерывными системами интеграции. В некоторых из имеющихся сегодня систем CI определенные виды ошибок делаются чаще, чем в других. С другой стороны, те или иные проблемы могут быть незначительными в каких-то конкретных приложениях. Мы надеемся, что, объединив знание ваших потребностей с этим обзором, вы сможете правильно принять первое решение.

В следующих пунктах обсудим проблемы, о которых мы только что сказали.

Проблема 1 — сложные стратегии сборки

Некоторые организации стремятся все и вся формализовать и структурировать, даже если это выходит за рамки разумного. В компаниях, создающих программное обеспечение, это особенно актуально в двух областях, таких как инструменты управления проектами и стратегии сборки на серверах CI.

Слишком сложная конфигурация инструментов управления проектами обычно заканчивается вопросом обработки рабочих процессов с помощью JIRA (или любого другого программного обеспечения для управления), и в итоге конфигурация никогда не подойдет ни под один шаблон в виде графиков, независимо от того, насколько он велик. Если ваш менеджер в восторге от таких конфигураций, то вы можете либо попробовать поговорить с ним, либо начинать искать новую команду. К сожалению, вы не обязательно добьетесь улучшений в этом вопросе.

Но когда дело доходит до CI, мы можем сделать больше. Инструменты непрерывной интеграции, как правило, сопровождаются и настраиваются именно нами — разработчиками. Это наши инструменты, призванные улучшить именно нашу работу. Если у кого-то есть непреодолимое желание обязательно пользоваться каждым рычажком и каждой кнопочкой, то таким людям лучше держаться

подальше от конфигурации системы CI, особенно когда их основная работа — весь день говорить и принимать решения.

На самом деле не нужно составлять сложные стратегии, чтобы решить, какую посылку или ветвь следует тестировать. Нет необходимости ограничивать тестирование какими-то тегами. Нет нужды выстраивать посылки в очереди, чтобы сделать сборку покрупнее. Нет необходимости отключать сборку через сообщения в посылках. Ваш непрерывный процесс интеграции должен быть максимально простым — тестируйте все и всегда, вот и все! Если вам не хватает аппаратных ресурсов для тестирования каждой посылки, то добавьте больше оборудования. Помните, что время программиста дороже, чем «железки».

Проблема 2 — слишком долгая сборка

Долгая сборка — именно то, что убивает производительность любого разработчика. Если вам приходится ждать несколько часов, чтобы просто узнать, правильно ли вы все сделали, то ни о какой продуктивности и речи не идет. Хорошо, если вам есть чем заняться, пока ваша функция тестируется. Мы же люди, а это значит, не очень хорошо работаем с многозадачностью. Переключение между различными задачами занимает много времени и в конечном итоге снижает нашу производительность в программировании до нуля. Очень трудно не терять концентрацию при работе над несколькими задачами одновременно.

Решение очевидно: сборка должна выполняться максимально быстро, любой ценой. Во-первых, попытайтесь найти узкие места и оптимизировать их. Если дело в производительности серверов сборки, то попробуйте масштабировать их. При отсутствии нужных результатов разделите каждую сборку на более мелкие части и распараллельте задачи.

Существует множество решений, которые позволяют ускорить медленную сборку, но иногда с ней ничего нельзя поделать. Например, если вы автоматизировали тестирование браузера или вам нужно выполнять длинные обращения к внешним сервисам, то будет очень трудно улучшить производительность по достижении некоторого жесткого предела. Так, если скорость автоматизированных приемочных тестов в вашем CI вас не устраивает, то вы можете слегка отойти от правила *«тестировать все и всегда»*. Для многих программистов особую важность имеют модульные тесты и статический анализ. Таким образом, в зависимости от вашего рабочего процесса медленные тесты браузера иногда можно отложить до релиза.

Другое решение — переосмысление всей архитектуры приложения. Если тестирование приложения занимает много времени, то часто это признак того, что его стоило бы разбить на несколько независимых компонентов, которые можно разрабатывать и тестировать отдельно. Написание программного обеспечения большими монолитными кусками — один из самых коротких путей к провалу. Обычно любой процесс разработки программного обеспечения ломается именно на ПО, в котором не организована модульная структура.

Проблема 3 — внешние определения задач

Некоторые системы непрерывной интеграции, особенно Jenkins, позволяют полностью настроить большинство параметров сборки и процессов тестирования через веб-интерфейс, не прибегая к необходимости работать с репозиторием кода. Однако стоит избегать добавления в команды сборки чего-то более сложного, чем простые точки входа. Такой подход сулит одни неприятности.

Ваш процесс сборки и тестирования, как правило, тесно связан с вашей кодовой базой. Если вы храните все определение во внешней системе, например, Jenkins или BuildBot, то внести изменения в этот процесс будет очень трудно.

В качестве примера проблемы, которая возникает из-за глобального внешнего определения сборки, предположим, что у нас есть проект с открытым исходным кодом. Сперва разработка была хаотичной и мы не удосуживались как-то формализовать правила стиля. Наш проект возымел успех, поэтому потребовался еще один крупный релиз. Через какое-то время мы перешли от версии `0.x` к `1.0` и решили переформатировать весь наш код под рекомендации PEP 8. Хорошо бы иметь статический контрольный анализ в составе процесса сборки, вследствие чего мы решили добавить исполнение инструмента `pep8` в определение сборки. Если бы у нас была только глобальная конфигурация внешней сборки, то возникла бы проблема при необходимости исправить что-либо в старом коде. Предположим, появилась критическая проблема безопасности, которая должна быть исправлена в версиях `0.x` и `1.y`. Мы знаем, что все версии старше 1.0 соответствовали указаниям по стилю и новая проверка PEP 8 отметит сборку как неудачную.

Решение этой проблемы — держать определение процесса сборки максимально близко к исходному коду. В некоторых системах CI (Travis CI и CI GitLab) такой рабочий процесс получается сам по умолчанию. В других решениях (Jenkins и BuildBot) вам придется принять дополнительные меры с целью гарантировать, что большая часть определений процесса сборки включена в код, а не передается от внешнего инструмента. К счастью, у вас есть следующие варианты такой автоматизации:

- скрипты Bash;
- Makefiles;
- код Python.

Проблема 4 — отсутствие изоляции

Мы уже неоднократно обсудили важность изоляции при программировании на Python. Мы знаем, что лучше всего изолировать среду выполнения Python на уровне приложений с помощью виртуальных окружений с `virtualenv` или `python -m venv`. К сожалению, при тестировании кода в непрерывном процессе интеграции этого, как правило, недостаточно. Среда тестирования должна быть максимально близка к production-среде, а этого трудно добиться без дополнительной виртуализации.

Ниже перечислены основные возможные проблемы, возникающие в отсутствие надлежащей изоляции на уровне системы при создании сборки:

- между сборками в файловой системе или в дополнительных сервисах сохранилось некоторое состояние (кэш, базы данных и т. д.);
- множественные сборки или тесты, взаимодействующие друг с другом через среду, файловую систему или сервисы;
- проблемы, возникающие из-за специфических особенностей операционной системы в продакшене и не перехваченные на сервере сборки.

Эти проблемы особенно опасны, если вам необходимо выполнять одновременные сборки одного приложения или распараллеливать одиночные сборки.

Некоторые структуры Python (в основном Django) позволяют получить дополнительный уровень изоляции для баз данных в попытке гарантировать, что хранилище будет очищено перед запуском тестов. Существует также весьма полезное расширение для `py.test` под названием `pytest-dbfixtures` (github.com/clearcodehq/pytest-dbfixtures), позволяющее добиться еще большей надежности. Однако такие решения не упрощают, а, наоборот, усложняют процесс сборки. Применение *новой* виртуальной машины для каждой сборки (в стиле Travis CI) кажется более элегантным и простым подходом.

Резюме

В данной главе мы рассмотрели разницу между централизованными и распределенными системами управления версиями и обсудили, почему стоит предпочесть распределенную систему другим. Мы увидели, почему для начала нужно выбрать Git. Обсудили общие рабочие процессы и стратегии ветвления Git. Наконец, узнали, что такое непрерывная интеграция/доставка/развертывание и какие популярные инструменты позволяют реализовать эти процессы.

В следующей главе мы расскажем, как четко документировать код.

11 Документирование проекта

Документирование — работа, которой разработчики и их руководители часто пренебрегают. Это часто происходит вследствие нехватки времени к концу цикла разработки, а также неуверенности людей в своих писательских способностях. Действительно, некоторые из них не слишком хороши в документировании, но большинство разработчиков должны быть в состоянии вести точную документацию.

Результатом пренебрежения документацией становится неупорядоченность в документах, которые к тому же пишутся в спешке. Разработчики часто ненавидят делать такого рода работу. Все становится еще хуже, когда приходится обновить существующие документы. У многих проектов скудная и устаревшая документация, поскольку никто в команде не знает, как правильно работать с ней.

Но ведь организовать процесс документирования в начале проекта и работать с документами так, будто они часть проекта, гораздо проще. Написание может быть даже интересным, если вы будете следовать нескольким простым правилам.

В этой главе:

- семь правил технического письма, обобщающих практические рекомендации;
- написание документации в виде кода;
- использование генераторов документации;
- написание самодокументирующихся API.

Технические требования

Ниже перечислены пакеты Python, упомянутые в этой главе, которые можно скачать с PyPI:

- `Sphinx`;
- `mkdocs`.

Вы можете установить эти пакеты с помощью команды:

```
python3 -m pip install <имя-пакета>
```

Семь правил технической документации

Вести хорошую документацию гораздо проще, чем писать код, но многие разработчики думают иначе. Документировать станет легче, как только вы усвоите следующий простой набор правил, касающихся технического письма.

Речь не идет о написании романа или поэмы, а лишь о фрагменте текста, который позволит другому человеку понять, что к чему в вашем ПО, API и т. д.

Каждый разработчик может производить такой материал, и в этом разделе мы приведем следующие семь правил, которые подходят на любой случай.

- *Пишите в два этапа*. Сначала сконцентрируйтесь на идеях, а затем на редактировании и формировании текста.
- *Ориентируйтесь на читателя*. Кто будет читать ваше творчество?
- *Упрощайте*. Излагайте прямо и просто. И пишите грамотно.
- *Ограничивайте объем информации*. Рассматривайте по одному понятию за раз.
- *Используйте реалистичные примеры кода*. Вместо иллюстраций и фантазий.
- *Пишите по минимуму, но достаточно*. Вы же не книгу пишете!
- *Используйте шаблоны*. Помогите читателям привыкнуть к общей структуре ваших документов.

Эти правила в основном почерпнуты из книги *Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects* Андреаса Рюпина, в которой описывается создание лучшей документации в программных проектах.

Пишите в два этапа

Питер Элбоу в своей книге *Writing With Power: Techniques for Mastering the Writing Process* объясняет: практически невозможно за один подход выдать идеальный текст. Проблема в том, что многие разработчики во время написания документации пытаются сразу же придумать какой-то идеальный текст. Единственный способ преуспеть в этом — останавливаться после каждых двух предложений, чтобы перечитать их и внести некоторые корректировки. Получается, разработчику приходится думать и об идеях, и о стиле одновременно.

Для мозга это слишком трудно, а результат зачастую получается не таким хорошим, каким мог бы быть. На шлифовку стиля и формы текста может уйти много времени и энергии, прежде чем его смысл будет полностью продуман.

Другой подход — забыть о стиле и организации текста и сначала заняться содержанием. Все идеи излагаются на бумаге, независимо от того, как написаны. Вы просто начинаете писать непрерывный поток мыслей и не останавливаетесь,

даже если допускаете очевидные грамматические ошибки или текст выглядит глупо. На данной стадии это не имеет значения, пока вы не изложите все идеи. Вы просто пишете то, что хотите сказать, занимаясь структурированием текста по минимуму.

Делая это, вы думаете только о том, что хотите сказать, и, вероятно, выжмете из себя больше, чем изначально ожидали.

Еще один побочный эффект такого *свободного письма* — вам могут прийти в голову сторонние идеи, не связанные с вашим предметом в данный момент. В этом случае их можно записать отдельно, чтобы они не потерялись, а затем вернуться к основной записи.

Второй шаг, очевидно, состоит в перечитывании документа и его шлифовке, чтобы он стал понятен всем. Вы улучшаете стиль, исправляете ошибки, реорганизуете текст и удаляете избыточную информацию.

Главное — сделать так, чтобы оба шага занимали примерно одинаковое количество времени. Если время на написание документации строго ограничено, то следует спланировать его соответствующим образом.

Пишите в два этапа

Сперва думаем о содержании, а затем о стиле и чистоте.

Ориентируйтесь на читателя

При написании контента вы должны принимать во внимание простой, но важный вопрос: *«Кто будет читать ваш документ?»*

Не всегда очевидно, что документация часто пишется для любого человека, который может использовать код. Читателем может быть кто угодно: исследователь, находящийся в поиске соответствующего технического решения проблемы, или разработчик, желающий реализовать новую функцию в задокументированном программном обеспечении.

Хорошая документация должна следовать простому правилу: текст должен предназначаться одному типу читателя. Эта философия упрощает написание, поскольку вы будете точно знать, кто ваш читатель.

Мы рекомендуем писать небольшой вводный документ, в котором в двух словах сказано, о чем эта документация и какую ее часть читать разным читателям, например: *«Atomisator — продукт, который получает данные по каналу RSS и сохраняет их в базе данных, выполняя фильтрацию.*

Если вы разработчик, то можете захотеть взглянуть на описание API (api.txt). Если менеджер, то вам будет интересен список функций и FAQ (features.txt).

Если вы дизайнер, то можно почитать об архитектуре и инфраструктуре (arch.txt)».

Знайте своего читателя, прежде чем начать писать

Упрощайте стиль

Простые вещи легче понять. Без сомнений.

Если используются короткие и простые предложения, то ваше творчество будет проще понять. Цель написания технической документации — предоставить читателям руководство по вашему ПО. Это не сказка и не роман, и ваш текст должен быть больше похож на инструкцию по эксплуатации микроволновки, а не на роман Толстого.

Ниже приведены несколько советов, которые стоит иметь в виду.

- Используйте короткие предложения. Они должны быть не длиннее 100–120 символов (включая пробелы). Для книги в мягкой обложке это около двух строк.
- Каждый абзац должен состоять из трех-четырех больших предложений, выражающих одну главную идею. Позвольте тексту «дышать».
- Не повторяйтесь слишком часто. Избегайте публицистического стиля, где одно и то же повторяется по сто раз.
- Не используйте разные формы времени. Достаточно просто настоящего времени.
- Избегайте шуток, если вы не крутой писатель. Шутки в технической книге — целое искусство, и лишь немногие писатели овладели им. При желании действительно добавить немного юмора делайте это в примерах кода, и все будет хорошо.

Вы пишете не фантастику. Стиль должен быть максимально простым.

Ограничивайте объем информации

У плохой документации программного обеспечения есть яркий признак — вы не можете найти в ней нужную информацию, даже если уверены, что она там есть. Немного почитав оглавление, вы начинаете искать по текстовым файлам, используя разные методы поиска, но так и не можете найти искомое. Но вы уверены, что нужная информация там есть, поскольку как-то раз видели ее.

Так часто происходит, когда писатели не организуют свои тексты с помощью осмысленных названий и заголовков. В документе могут быть тонны информации, но от этого нет толку, если читатель не сможет найти что-то конкретное.

В хорошем документе пункты должны иметь заголовок, отражающий его содержание, а заголовок документа — вбирать внутреннее наполнение в короткой фразе. Содержание должно включать названия всех разделов, тем самым помогая читателю искать информацию.

Простой, но эффективный способ для выбора названий и заголовков — задать себе вопрос: «По какой фразе в поисковике должен найтись этот раздел?»

Используйте реалистичные примеры кода

Нереалистичные примеры кода усложняют понимание документации.

Например, если вы хотите показать какие-либо строковые литералы, то объяснять «на пальцах» нельзя. При необходимости показать читателю, как использовать ваш код, почему бы не сделать это на реальном примере? Желательно делать так, чтобы каждый пример кода можно было вырезать и вставить в настоящую программу.

Для демонстрации примера плохого использования предположим, что мы хотим показать, как задействовать функцию `parse()` из проекта `atomisator`, в котором реализуется синтаксический анализ RSS-канала. Вот пример применения с помощью нереального воображаемого кода:

```
>>> from atomisator.parser import parse
>>> # Используем это:
>>> stuff = parse('some-feed.xml')
>>> next(stuff)
{'title': 'foo', 'content': 'blabla'}
```

В хорошем примере использовался бы источник данных, похожий на настоящий URL для RSS-канала, и вывод напоминал бы реальную статью:

```
>>> from atomisator.parser import parse
>>> # Используем это:
>>> my_feed = parse('http://tarekziade.wordpress.com/feed')
>>> next(my_feed)
{'title': 'eight tips to start with python', 'content': 'The first tip
is..., ...'}
```

Такая небольшая разница может показаться излишней, но документация при этом станет гораздо более полезной. Читатель может скопировать эти строки

в оболочку, понять, что функция `parse()` ожидает в качестве параметра URL и возвращает итератор, который содержит веб-статьи.

Конечно, не всегда можно привести реалистичный пример. Это особенно верно для слишком общего кода, не сконцентрированного на конкретной задаче. Даже в нашей книге есть подобные примеры. Во всяком случае, вы всегда должны стремиться свести количество таких нереальных примеров до минимума.

Примеры кода должны подходить для реального использования в программе.

Пишите по минимуму, но достаточно

В большинстве гибких методологий документация не стоит на первом месте. Создать рабочее программное обеспечение важнее, чем составить подробную документацию. Таким образом, лучшей рекомендацией, как описывает в своей книге *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process* Скотт Амблер, будет определить в документации самое необходимое, а не пытаться вместить туда все.

Так, рассмотрим пример документации простого проекта, который доступен на GitHub. `ianitor` (github.com/clearcodehq/ianitor) — инструмент, помогающий регистрировать процессы в кластере Consuk и в основном предназначенный для системных администраторов. Если вы посмотрите на его документацию, то увидите там всего один документ (файл `README.md`). В нем изложено только то, как работает инструмент и как его использовать. С точки зрения администратора, этого достаточно. Ему важно знать, как настроить и запустить инструмент, а остальным `ianitor` и не нужен. Данный документ отвечает всего на один вопрос: «Как мне поднять `ianitor` на моем сервере?»

Используйте шаблоны

Почти все страницы «Википедии» выглядят одинаково. Справа есть блоки, в которых обобщается информация из документов, принадлежащих к одной и той же области. Первая часть статьи, как правило, содержит оглавление со ссылками, ведущими на якоря в этом же тексте. В конце всегда есть справочный раздел.

Пользователи привыкают к такому дизайну. Например, они знают, что им нужно посмотреть в содержание, и если желаемое там не найдется, то надо перейти к справке, чтобы увидеть, есть ли еще какой-нибудь сайт на данную тему. Это работает для любой страницы в «Википедии». Однажды усвоив формат ее статьи, вы сможете быстро и эффективно искать полезную информацию.

Таким образом, здесь подразумевается создание общего шаблона для документов, что обеспечивает более эффективный поиск данных. Пользователи привыкают к общей структуре информации и учатся читать быстрее.

Введение шаблона для каждого вида документа — это еще и быстрый старт для самого писателя.

Документация как код

Лучший способ сохранить документацию вашего проекта в актуальном состоянии — считать ее частью кода и хранить в репозитории, в котором содержится и исходный код. Совместное хранение кода и документации дает следующие преимущества.

- Имея правильную систему управления версиями, вы можете отслеживать все изменения, внесенные в документацию. Если встанет вопрос, чем является некое удивительное поведение кода: ошибкой или просто старой и забытой функцией, то вы сможете просмотреть историю документации для данной функции и найти ответы.
- Разрабатывать различные версии документации проще, если проект делится на несколько параллельных ветвей (например, для разных клиентов). Исходный код проекта отходит от основной ветви разработки, и документация следует за ним.
- Существует много инструментов, позволяющих создавать документацию API прямо из комментариев, включенных в исходный код. Это один из лучших способов составления документации для проектов, которые представляют собой API для других компонентов (например, в виде многократно используемых библиотек и удаленных сервисов).

У языка Python есть уникальные качества, которые делают процесс документирования простым и интересным. Сообщество Python также располагает огромным набором инструментов, позволяющих создать красивую и полезную документацию прямо из кода Python. Основа для этих инструментов — так называемые *строки документации* (docstrings).

Использование строк документации в Python

Строки документации — специальные строковые литералы Python, которые предназначены для документирования функций, методов, классов и модулей Python. Если первое определение функции, метода, класса или модуля представляет собой строковый литерал, то автоматически становится строкой документации и станет значением атрибута `__doc__` для этой функции, метода, класса или модуля.

Во многих примерах кода в этой книге уже есть строки документации, но для ясности приведем общий пример модуля, который содержит все возможные типы строк документации, а именно:

```
""" Пример модуля со строкой документации.

В этом модуле есть все четыре вида строк документации:
- для модуля;
- для функции;
- для метода;
- для класса.
"""

def show_module_documentation():
    """ Выводит документацию модуля.

    Документация модуля доступна в виде глобального атрибута __doc__.
    Этот атрибут может быть доступен и изменен в любое время.
    """
    print(__doc__)
class DocumentedClass:
    """ Класс документации метода.
    """

    def __init__(self):
        """ Инициализация экземпляра класса.
        Интересное замечание: строки документации являются валидным
        определением. Это означает, что если функция или метод ничего
        не делают, но строка документации есть, то других определений
        не нужно.

        Такие функции полезны для определения абстрактных методов или
        вставки заглушек в код на будущее.
        """
```

В Python также есть функция `help()`, которая является точкой входа для встроенной справочной системы. Она предназначена для интерактивного использования в рамках интерактивной сессии интерпретатора аналогично тому, как работает просмотр системных справочных страниц с помощью `command` в UNIX. Если вы укажете экземпляр модуля в качестве входного аргумента функции `help()`, то она отформатирует все строки документации объектов модуля в виде древовидной структуры. Ниже приведен пример.

```
Help on module docexample:

NAME
    docexample — Пример модуля со строкой документации.
```

```
FILE
    /Users/swistakm/docexample.py

DESCRIPTION
    В этом модуле есть все четыре вида строк документации:
    - для модуля;
    - для функции;
    - для метода;
    - для класса.

CLASSES
    DocumentedClass

    class DocumentedClass
     |  Класс документации метода.
     |
     |  Описанные методы:
     |
     |  __init__(self)
     |      Инициализация экземпляра класса.
     |
     |      Интересное замечание: строки документаций являются валидным
     |      определением. Это означает, что если функция или метод ничего
     |      не делают, но строка документации есть, то других определений
     |      не нужно.
     |
     |      Такие функции полезны для определения абстрактных методов или
     |      вставки заглушек в код на будущее.

FUNCTIONS
    show_module_documentation()
        Выводит документацию модуля.
        Документация модуля доступна в виде глобального атрибута __doc__.
        Этот атрибут может быть доступен и изменен в любое время.
```

Популярные языки разметки и стилей для документации

Внутри строк документации можно писать что угодно и как угодно. Разумеется, есть официальный документ PEP 257 (соглашение о строках документации), который представляет собой общее руководство по оформлению строк документации, но внимание в нем уделяется в основном нормированному форматированию многострочных строковых литералов для целей документирования, а рекомендаций по языку разметки не дано.

В любом случае, имея желание сделать красивую и полезную документацию, стоит выбрать для нее некий формализованный язык разметки, особенно если вы

планируете использовать инструменты генерации документации. Правильная разметка позволит генераторам раскрашивать код, делать расширенное форматирование текста, вставлять ссылки на другие документы и функции или даже включать нетекстовые ассеты, например образы автоматически генерируемых диаграмм классов.

Хорошим считается такой язык разметки, на котором легко писать и который удобно читать в сыром виде за пределами сгенерированной справочной документации. Лучше всего, если с ее помощью можно будет перейти на более длинные источники документации за пределами строк документации Python. Один из самых распространенных языков разметки, разработанный специально для Python с учетом этих целей, — reStructuredText. Он используется системой документации Sphinx и является языком разметки, служащим для создания официальной документации самого языка Python. Основные элементы синтаксиса данной разметки описаны в приложении в конце книги.

Другие популярные варианты простых текстовых языков разметки для строк документации — Markdown и AsciiDoc. Первый особенно популярен в сообществе пользователей GitHub и в целом является наиболее распространенным языком разметки документации. Кроме того, он часто поддерживается «из коробки» различными инструментами для самодокументирующихся веб-API.

Популярные генераторы документации для библиотек Python

Как мы уже говорили ранее, целевая аудитория у документации к программному обеспечению может быть разнообразной. Доступ к документации непосредственно из исходного кода проекта выглядит нормально для пользователей, если это сами разработчики. Но такой способ доступа удобен не для всех. Кроме того, в некоторых компаниях могут быть требования по доставке документации клиентам в печатной форме.

Именно поэтому средства генерации документации важны и нужны. Они позволяют извлечь выгоду из обращения с документацией, как с кодом, сохраняя при этом возможность получить готовый к распечатке документ, который можно просматривать, искать в нем что-то и читать, не имея доступа к исходному коду. В экосистеме Python есть множество удивительных инструментов с открытым кодом, которые позволяют создавать проектную документацию непосредственно из исходного кода. Два наиболее популярных инструмента создания документации в сообществе Python — Sphinx и MkDocs. Мы кратко поговорим о них ниже.

Sphinx

Sphinx (sphinx.pocoo.org) — набор скриптов и расширений `docutils`, которые можно использовать для создания структуры HTML из дерева текстовых документов, создаваемых с помощью языка синтаксиса reStructuredText (подробнее о нем — в приложении). Кроме того, Sphinx поддерживает множество других форматов вывода документации, даже PDF и LaTex. Этот инструмент применяется, например, для создания официальной документации Python и очень популярен в данной нише. В нем есть неплохая система просмотра, а также простенький, но хороший поисковый движок на JavaScript на стороне клиента. Он также задействует `pygments` для отрисовки приятной подсветки синтаксиса в примерах с кодом.

Sphinx может быть легко настроен таким образом, что это хорошо наложится на подход с документированием, описанный в предыдущем разделе. Устанавливается как пакет `Sphinx` через `pip`.

Самый простой способ начать работать с Sphinx — использовать скрипт `sphinx-quickstart`. Эта утилита вместе с `Makefile` генерирует скрипт, который можно будет при необходимости задействовать для генерации веб-документации. Он в интерактивном режиме задаст вам несколько вопросов, а затем выгрузит все исходное дерево документации и файл конфигурации. Как только это будет сделано, вы сможете в любой момент легко выполнить настройку. Предположим, что мы уже выгрузили все среду Sphinx и хотим посмотреть на его HTML-представление в деле. Это легко сделать с помощью команды `make html` следующим образом:

```
project/docs$ make html
sphinx-build -b html -d _build/doctrees . _build/html
Running Sphinx v1.3.6
making output directory...
loading pickled environment... not yet created
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: 1 added, 0 changed, 0 removed
reading sources... [100%] index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices... genindex
writing additional pages... search
copying static files... done
copying extra files... done
dumping search index in English (code: en) ... done
dumping object inventory... done
build succeeded.
Build finished. The HTML pages are in _build/html.
```

На рис. 11.1 ниже показан пример HTML-версии документации, созданной с помощью Sphinx.

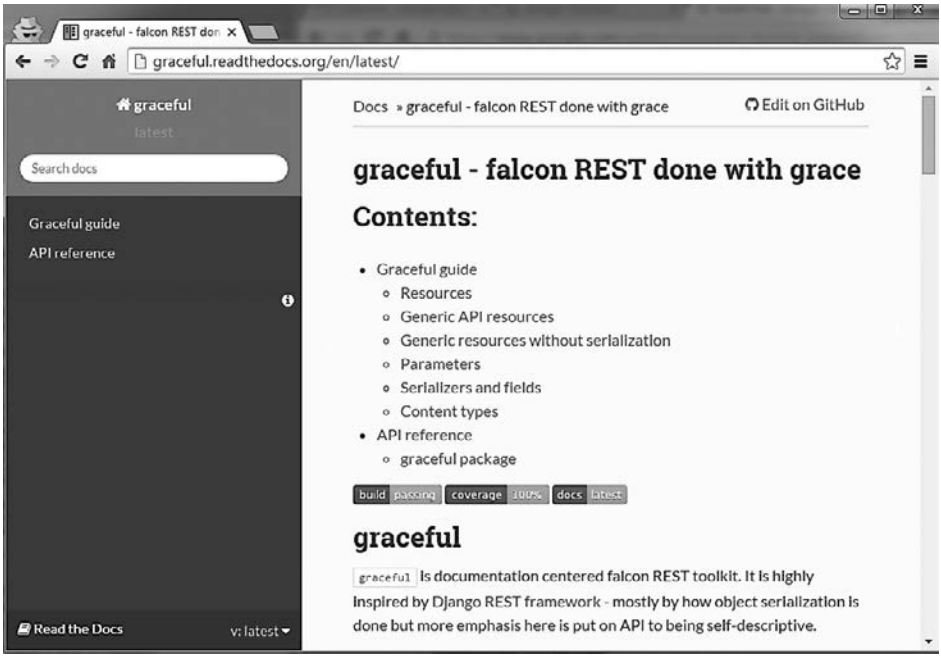


Рис. 11.1. HTML-версия документации, созданной с помощью Sphinx

Помимо HTML-версий документов, данный инструмент делает автоматические страницы, например список модулей и указатель. В Sphinx есть несколько расширений `docutils` для управления этими функциями. Основные из них таковы:

- директива, делающая оглавления;
- маркер, который может быть использован, чтобы зарегистрировать документ в качестве справки модуля;
- маркер добавления элемента в указатель.

Работа с индексными страницами (указателем)

В Sphinx есть директива `toctree`, которую можно использовать для вставки в документ оглавления со ссылками на другие документы. Каждая строка — это файл с относительным путем, начиная с расположения текущего документа. Кроме того, могут быть предусмотрены глобальные имена, чтобы добавить несколько файлов, соответствующих выражению.

Например, файл указателя в папке cookbook, которую мы определили ранее, может выглядеть следующим образом:

```
========
Cookbook
========

Welcome to the Cookbook.

Available recipes:

.. toctree::
   :glob:
   *
```

С помощью этого синтаксиса можно вывести на HTML-странице список всех документов reStructuredText, имеющихся в папке cookbook. Данная директива может использоваться во всех файлах указателей для создания просматриваемой документации.

Регистрация справки модуля

Для файлов справки добавляется особый маркер, чтобы он автоматически попал в список на странице указателя. Это делается следующим образом:

```
=======
session
=======

.. module:: db.session

The module session...
```

Обратите внимание: здесь используется префикс db, чтобы избежать конфликта имен. Sphinx будет применять его в качестве категории модуля и сгруппирует все модули, которые начинаются с db, в эту категорию.

Добавление маркеров указателя

Другой вариант заполнения указателя — связь документа точкой входа следующим образом:

```
=======
session
=======

.. module:: db.session
```

```
.. index::
   Database Access
   Session

The module session...
```

В указатель добавятся страницы `Database Access` и `Session`.

Перекрестные ссылки

Наконец, в Sphinx есть встроенная разметка для установки перекрестных ссылок. Например, ссылка на модуль делается следующим образом:

```
:mod:`db.session`
```

Здесь `:mod:` — префикс маркера модуля, а `` `db.session` `` — имя модуля, с которым устанавливается связь (он зарегистрирован ранее). Помните: `:mod:`, как и предыдущие элементы, — это специфические директивы, введенные Sphinx в reStructuredText.

Sphinx предоставляет еще множество возможностей, которые вы можете найти на их сайте. Например, функция autodoc — отличный вариант для автоматического извлечения ваших доктестов в целях создания документации. Получить дополнительную информацию можно на sphinx.pocoo.org.

MkDocs

MkDocs (www.mkdocs.org) — очень минималистичный генератор статических страниц, которые можно использовать для документирования проектов. В нем не хватает встроенных функций *autodoc*, аналогичных имеющимся в Sphinx, но применяется намного более простой и читабельный язык разметки Markdown. Плюс есть возможности к расширению. Намного легче написать плагин MkDocs, чем расширение Docutils, которое может быть использовано в Sphinx. Таким образом, если у вас есть весьма конкретные потребности в документации и для их удовлетворения не хватает имеющихся инструментов и их расширений, то MkDocs поможет вам создать что-то свое.

Сборка документации и непрерывная интеграция

Sphinx и аналогичные инструменты генерации документации действительно улучшают читабельность документации и удобство работы с ней для потребителя. Как мы уже отмечали ранее, это особенно полезно, если некоторые части документации

тесно связаны с кодом, например, в виде строки документации. Данный подход действительно позволяет быть уверенными, что версия документации совпадает с кодом, однако не гарантирует, что у читателей будет доступ к самой последней и уточненной версии.

Одного лишь голого кода тоже недостаточно, если целевые читатели документации не владеют инструментами командной строки и не знают, как превратить документ в просматриваемую и читабельную форму. Вот почему важно ориентировать документацию на потребителя и вовремя редактировать ее в соответствии с изменениями.

Лучший способ разместить документацию, собранную с помощью Sphinx, — это сгенерировать HTML-сборку и отправить ее в качестве статического ресурса на ваш веб-сервер. Sphinx использует нужный `Makefile` для сборки HTML-файлов с помощью команды `make html`. Поскольку это очень распространенная утилита, она должна легко интегрироваться с любой из систем непрерывной интеграции, которые мы обсуждали в главе 10.

Если вы документируете проект с открытым кодом с помощью Sphinx, то значительно упростите себе жизнь с помощью *Read the Docs* (https://readthedocs.org/). Это бесплатный сервис для размещения документации проектов Python с открытым исходным кодом. Конфигурация выполняется без проблем и отлично интегрируется с GitHub и Bitbucket. На практике, если ваши аккаунты правильно соединены и репозиторий кода установлен верно, разместить документацию на *Read the Docs* можно в два щелчка.

Документирование веб-API

Принципы документирования веб-API почти такие же, как и у других видов программного обеспечения. Нам нужно выбрать аудиторию, предоставить документацию в понятной для нее форме (в данном случае в виде веб-страницы) и прежде всего убедиться, что читатели имеют доступ к актуальной документации.

Из-за этого крайне важно, чтобы документация веб-API генерировалась из исходного кода, который предоставляет эти API. К сожалению, из-за сложной архитектуры большинства фреймворков классические инструменты документации наподобие Sphinx редко бывают полезны для документирования типичных конечных точек HTTP в веб-API. В связи с этим часто бывает так, что возможности автоматического документирования встроены в ваш веб-фреймворк. Подобного рода фреймворк может давать документацию, читаемую пользователем, сам по себе или выдавать стандартизированное описание API в машинно-читаемом формате, который можно далее обработать с помощью специализированного браузера документации.

Существует и совершенно другая философия документирования веб-интерфейсов, основанная на идее прототипирования API. Инструменты для прототипирования позволяют использовать документацию в качестве программного контракта, который можно применить в качестве заглушки API еще до начала разработки. Часто этот вид инструмента позволяет автоматически проверять, соответствует ли структура API тому, что реализовано в сервисе. При таком подходе документация может служить дополнительной функцией инструмента тестирования API.

Документация как прототип API с API Blueprint

API Blueprint — язык описания веб-API, сочетающий в себе читабельность и хорошее определение. Это нечто вроде Markdown для описания веб-сервиса. Он позволяет документировать что угодно из структуры путей URL через структуру HTTP-запроса/ответа и заголовки, вплоть до сложных обменов запросами и ответами. Ниже приведен пример воображаемой Cat API, описанной с помощью API Blueprint:

````
FORMAT: 1A
HOST: https://cats-api.example.com

# Cat API
This API Blueprint demonstrates example documentation of some imaginary
Cat API.

# Group Posts
This section groups Cat resources.

## Cat [/cats/{cat_id}]

A Cat is central and only resource utilized by Cat API.

+ Parameters
    + cat_id: `1` (string) - The id of the Cat.

+ Model (application/json)

    ```js
    {
        "data": {
            "id": "1", // note this is a string
            "breed": "Maine Coon",
            "name": "Smokey"
    ```

### Retrieve a Cat [GET]

Returns a specific Cat.
````

```
+ Response 200

     [Cat][]

### Create a Cat [POST]

Create a new Post object. Mentions and hashtags will be parsed out of the
Post text, as will bare URLs...

+ Request

     [Cat][]

+ Response 201

     [Cat][]
```

Сам по себе API Blueprint — всего лишь язык. Его сила исходит из того, что он может быть легко написан вручную и из огромного набора сред, поддерживающих этот язык. На момент написания книги на официальной странице API Blueprint было перечислено более 70 инструментов, которые поддерживают этот язык. Часть из них могут даже генерировать функциональные серверы API, предназначенные для сокращения времени разработки, поскольку можно использовать фиктивные серверы, например, с помощью внешнего интерфейса кода еще до того, как программисты начинают разработку бэкенд-API.

Самодокументирующиеся API со Swagger/OpenAPI

Хотя самодокументирующиеся API — более традиционный подход для документирования веб-API (по сравнению с документированием через прототипы API), можно заметить кое-какие интересные тенденции, которые появились в течение последних нескольких лет. В прошлом, когда API-фреймворки должны были поддерживать возможности автоматической документации, это почти всегда означало наличие у фреймворка встроенной структуры метаданных API с пользовательским движком рендеринга документации. Если кому-то нужно несколько автоматически задокументированных сервисов, то следует использовать одну и ту же структуру для каждого сервиса, либо получится противоречивая структура документации.

С появлением архитектуры микросервисов такой подход стал крайне неудобен и неэффективен. В настоящее время очень часто разные сервисы в рамках одних и тех же проектов пишутся с использованием различных механизмов, библиотек и даже совершенно разных языков программирования. Наличие различных библиотек документации для каждой структуры и языка дает очень противоречивую

документацию, поскольку у каждого инструмента будут свои сильные и слабые стороны.

Подход, который решает эту проблему, требует отделения отображения документации (рендеринг и просмотр) от фактического определения документации. Данный подход аналогичен прототипированию API, поскольку требует стандартизированного языка описания API. Но здесь разработчик редко использует этот язык явным образом. Именно фреймворк отвечает за создание машинно-читаемого определения API из структуры кода, написанного в данном фреймворке.

Один из таких машинно-читаемых языков описания веб-API — OpenAPI. Его спецификация — результат развития популярного инструмента документации Swagger. Поначалу это был внутренний формат метаданных инструмента Swagger, но сразу после его стандартизации вокруг данной спецификации появились и другие инструменты. Благодаря OpenAPI многие веб-фреймворки позволяют описать структуру их API, используя тот же формат метаданных, и потому вся документация будет иметь одинаковый вид.

Создание хорошо организованной системы документации

Простой способ направить читателей документации — дать каждому из них указания, о которых мы узнали в предыдущем разделе этой главы.

С точки зрения автора, это делается с помощью набора повторно используемых шаблонов вместе с указаниями, описывающими, как и когда применять их в проекте. Это называется *портфелем документации*.

Читателю же важно иметь возможность безболезненно просматривать документацию и эффективно находить информацию. Это делается путем создания *шаблона документа*.

Очевидно, начать стоит с рекомендаций для авторов, поскольку без них читателям будет нечего читать. Посмотрим, как выглядит подобный портфель, и создадим такой сами.

Создание портфеля документации

Существует много видов документов, которые может включать проект программного обеспечения: от документов низкого уровня, относящихся непосредственно к коду, до проектных документов, содержащих высокоуровневый обзор приложения.

Например, Скотт Амблер определяет обширный список типов таких документов в своей книге *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. В портфель входит все: от ранних спецификаций до документов по

эксплуатации. Сюда включены даже документы по управлению проектами, поэтому все необходимые документы построены по стандартному набору.

Поскольку полный портфель тесно связан с методологиями, используемыми для создания программного обеспечения, в этой главе основное внимание будет уделено только общему подмножеству, которое вы можете дополнить конкретно вашими потребностями. Создание эффективного портфеля занимает много времени, поскольку для этого нужно будет перестроить свои привычки.

Общий набор документов в области программных проектов можно разделить на следующие три категории:

- *проект* — включает в себя все документы с информацией об архитектуре и низкоуровневом проектировании, диаграммы классов и схемы баз данных;
- *использование* — включает все документы о том, как применять программное обеспечение. Может быть представлена в виде пособия, справки или по типу «книги рецептов»;
- *эксплуатация* — содержит рекомендации о том, как разворачивать, обновлять ПО или работать с ним.

Обсудим представленные категории.

Проект

При создании подобных документов важно убедиться, что целевая аудитория хорошо известна и объем содержимого ограничен. Таким образом, общий шаблон для проектной документации — это простая структура с небольшим количеством советов для писателя.

Структура документа может включать в себя следующее:

- название;
- имя автора;
- теги (ключевые слова);
- описание (аннотация);
- цель (кто должен это читать);
- содержание (со схемами);
- ссылки на другие документы.

Содержание при печати должно занимать три или четыре страницы, поэтому следует ограничить объем. Если он становится больше, то его нужно разделить на несколько документов или сократить.

В шаблон также входит имя автора и список тегов для управления эволюцией и облегчения классификации. Об этом будет рассказано далее в текущей главе.

Ниже приведен пример шаблона документа, написанного с помощью reStructuredText:

```
========================================
Название документа
========================================

: Автор: автор документа.
: Теги: теги документа, разделенные пробелами.

:аннотация:

    небольшая аннотация о вашем документе
.. Содержание ::


Аудитория
========

Объясните, кто является целевой читательской аудиторией.

Контент
=======

Сам документ. Здесь может быть несколько разделов.

Ссылки
==========

Здесь должны быть ссылки на другие документы и сайты.
```

Использование

Документация по использованию описывает, как работает конкретная часть ПО. Может содержать детали низкого уровня, например принцип работы функции, а также детали высокого уровня, такие как аргументы командной строки для вызова программы. Это наиболее важная часть документации во фреймворках, поскольку целевая аудитория — это в основном разработчики, которые собираются повторно применять код.

Три основных вида документов таковы:

- *инструкция (рецепт)* — короткий документ, который объясняет, как сделать что-то конкретное; ориентирован на одну читательскую аудиторию и фокусируется на одной конкретной теме;
- *руководство* — шаг за шагом объясняет, как использовать все программное обеспечение в целом; может давать ссылку на рецепты, и каждый экземпляр предназначен для одной читательской аудитории;

- *справка модуля* — документ низкого уровня, который объясняет, что входит в модуль; может быть показан, например, путем вызова `help` в модуле.

Рецепт. В документе этого вида рассматривается конкретная задача и приводится ее решение. Например, ActiveState — огромное хранилище рецептов Python, в которых разработчики описывают, как выполнить некие действия в этом языке (code.activestate.com/recipes/langs/python/). Такой набор рецептов, относящихся к одной области/проекту, часто называют *кулинарной книгой*.

Эти рецепты должны быть короткими и структурированными:

- название;
- отправитель;
- последнее обновление;
- версия;
- категория;
- описание;
- источник (исходный код);
- обсуждение (пояснения к коду);
- комментарий (из Интернета).

Часто этот документ умещается на одном экране и написан максимально кратко. Данная структура идеально подходит для потребностей программного обеспечения и может быть адаптирована в общей структуре, где добавляются целевая аудитория, а категория заменяется тегами:

- название (короткая фраза);
- автор;
- теги (ключевые слова);
- кому предназначена книга;
- предпосылки (другие документы для чтения, например);
- задача (краткое описание);
- решение (основной текст, один или два экрана);
- ссылки (ссылки на другие документы).

От даты и версии здесь нет пользы, поскольку проектная документация должна управляться как исходный код в проекте. Это значит, лучший способ работать с версиями — задействовать систему управления версиями. В большинстве случаев это точно такой же репозиторий кода, как тот, который применяется для кода проекта.

Простой многоразовый шаблон для рецептов может выглядеть примерно так:

```
Название документа
================================================================================
: Автор: автор рецепта
: Теги: теги документа, разделенные пробелами

:аннотация:

    небольшая аннотация о вашем документе

.. Содержание ::

Аудитория
================================================================================

Объясните, кто является целевой читательской аудиторией.

Что нужно установить
================================================================================

Список предпосылок для реализации данного рецепта. Это могут быть
дополнительные документы, программное обеспечение, специальные библиотеки,
параметры среды или просто все, что требуется, помимо интерпретатора.

Проблема
================================================================================

Объясните проблему, которую этот рецепт пытается решить.

Решение
================================================================================

Решение проблемы — самая важная часть.

Ссылки
================================================================================

Ссылки на другие документы.
```

Руководство. Отличие руководства от рецепта состоит в цели: не решение отдельно взятой проблемы, а скорее пошаговое описание того, как пользоваться приложением. Руководство может быть больше рецепта и охватывать многие части приложения. Например, Django выкладывают список учебников на своем сайте. В руководстве *Writing your first Django App, part 1* (docs.djangoproject.com/en/1.9/intro/tutorial01/) на паре снимков экрана объясняется, как сделать приложение на Django.

Структура такого документа будет выглядеть следующим образом:

- название (короткая фраза);
- автор;

- теги (ключевые слова);
- описание (аннотация);
- кому предназначена книга;
- предпосылки (другие документы для чтения, например);
- само руководство (основная часть);
- ссылки (на другие документы).

Справка модуля. Последний шаблон, который мы рассмотрим, — справка модуля. Она относится к одному модулю и содержит описание его содержимого, а также примеры использования.

Некоторые инструменты позволяют автоматически создавать подобные документы путем извлечения строк документации и вычисления с помощью `pydoc`, например Epydoc (epydoc.sourceforge.net). Таким образом можно сформировать обширную документацию, основанную на самоанализе API. Этот вид документации часто предоставляется в фреймворке Python. Так, у Plone есть сервер, который хранит актуальную коллекцию справок модуля. Прочитать об этом можно, перейдя по ссылке api.plone.org.

Ниже перечислены основные проблемы, связанные с этим подходом:

- нет хорошего варианта выбора модулей, которые были бы интересны для документирования;
- код может оказаться перегруженным документацией и стать сложным для понимания.

Кроме того, модуль документации содержит примеры, которые иногда ссылаются на несколько частей модуля и в которых трудно выделить строки документации классов и функций. Строку документации модуля можно использовать для этой цели, написав текст в верхней части модуля. В результате мы получаем гибридный файл, состоящий из смешанных блоков текста и кода. Читатель путается, когда код составляет менее 50 % от общей длины. Если вы автор, то это прекрасно. Но когда люди пытаются прочитать код (не документацию), им приходится буквально пролистывать строки документации.

Другой подход заключается в разделении текста в самом файле. Какой модуль Python будет иметь свой файл справки — решается с помощью ручного выбора. Строки документации могут быть отделены от кодовой базы и жить собственной жизнью, как мы увидим ниже. Python так и задокументирован, собственно говоря.

Многие разработчики не согласны с тем, что разделение документации и кода лучше, чем строки документации. Такой подход означает, что процесс документирования полностью интегрирован в цикл разработки, поскольку документация будет быстро утрачивать актуальность. Использование строк документации решает эту проблему ввиду того, что код находится близко к примерам применения, однако

не доходит до документа высокого уровня, который можно задействовать как часть обычной документации.

Следующий шаблон «Справки по модулю» очень прост, поскольку содержит лишь немного метаданных, а затем — только контент; цель не определена, так как именно разработчики хотят использовать модуль:

- название (имя модуля);
- автор;
- ключевые слова (слова);
- контент.

В следующей главе мы поговорим о разработке на основе тестирования (TDD) с использованием доктестов и справки модуля.

Эксплуатация. Такие документы используются для описания того, как может работать программное обеспечение:

- документы по установке и развертыванию;
- документы по администрированию;
- документы с FAQ;
- документы, объясняющие, как люди могут внести свой вклад, попросить о помощи или связаться с разработчиками.

Эти документы очень специфичны, но для них вполне подойдет шаблон руководства, описанный ранее.

Ваш собственный портфель документации

Шаблоны, которые мы обсуждали ранее, — лишь основа, которую можно использовать для документирования программного обеспечения. Со временем вы начнете разрабатывать собственные шаблоны и стили для создания документации. Но всегда нужно помнить соответствующий подход к проектной документации: каждый новый документ должен иметь четко определенную целевую аудиторию и удовлетворять реальную потребность. Документы, не решающие нужные проблемы, писать не следует.

Каждый проект уникален и имеет свои потребности в документации. Например, небольшим инструментам, простым в использовании, достаточно небольшого файла `README`. Такой минималистичный подход совершенно нормален, если целевые читатели точно определены и составляют некую группу (к примеру, системные администраторы).

Кроме того, не надо привязываться к этим шаблонам слишком строго. Некоторые дополнительные метаданные также могут оказаться очень полезными в крупных проектах или строго формализованных командах. Например, теги предназначены для улучшения поиска по тексту в больших документациях, но не будут представлять никакой ценности, если вся документация состоит всего из нескольких документов.

Указывать автора документа тоже не всегда хорошо. Подобный подход может быть особенно сомнителен в проектах с открытым исходным кодом. В таких проектах хочется, чтобы сообщество тоже могло внести свой вклад в документацию. В большинстве случаев эти документы постоянно обновляются всякий раз, когда кто-то вносит свою лепту в проект. Многие почему-то считают *автора* документа его *владельцем*. Из-за этого человек может постесняться обновить документацию, если у каждого документа будет указан автор. Как правило, программное обеспечение для управления версиями дает более четкую и прозрачную информацию об авторах реальных документов, чем явное их указание в метаданных. Однако авторов стоит указывать в проектных документах, особенно в проектах, где процесс проектирования строго формализован. В качестве примера можно привести серию документов PEP, в которых приведены предложения по улучшению языка Python.

Создание шаблона документации

Портфель документов, созданный нами в предыдущем разделе, обеспечивает структуру на уровне документа, но не предоставляет способ группировки и организации документа для будущих читателей. Андреас Рюпин называет это шаблоном документации, намекая на ментальную карту, которую читатели используют при просмотре документации. Он пришел к выводу, что лучший способ организовать документы — создать логическое дерево.

Иными словами, различные виды документов, входящие в портфель, следует разместить на своем месте в дереве каталогов. Это место должно быть очевидным и для авторов, когда они создают документ, и для читателей, поскольку им предстоит искать в нем информацию.

При просмотре документации хорошо помогает индексирование страниц на каждом уровне.

Создание шаблона документа делается в два этапа:

- построение дерева для производителей (авторов);
- построение дерева для потребителей (читателей) на вершине дерева для авторов.

Это различие между производителями и потребителями весьма важно, поскольку они получают доступ к документам в разных местах и форматах.

Шаблон для автора

С точки зрения автора, каждый документ — то же самое, что и модуль Python. Он должен храниться в системе управления версиями так же, как код. Писателей не волнует окончательный вид их творчества и его доступность. Они просто хотят написать документ, который будет являться единственным источником истины по данной теме. Файлы reStrcturedText, хранящиеся в дереве каталогов, доступны в системе контроля версий вместе с программным кодом, что служит удобным решением в вопросе создания шаблона документации для авторов.

По соглашению папка `docs` используется как корневой каталог дерева документации следующим образом:

```
$ cd my-project
$ find docs
docs
docs/source
docs/source/design
docs/source/operations
docs/source/usage
docs/source/usage/cookbook
docs/source/usage/modules
docs/source/usage/tutorial
```

Обратите внимание: дерево находится в папке `source`, поскольку папка `docs` будет использоваться в качестве корневой для установки специального инструмента в следующем подразделе.

Файл `index.txt` добавляется на каждом уровне (кроме корня) и включает описание того, какие документы содержит папка или подпапка. Эти файлы могут вмещать список документов. Например, в папке `operations` может лежать перечень документов по эксплуатации:

```
==========
Эксплуатация
==========
Этот раздел содержит список документов по эксплуатации:
- как установить и запустить проект;
- как установить и управлять базой данных проекта.
```

Важно знать, что люди, как правило, забывают обновлять такие перечни документов и оглавления. Поэтому лучше бы они обновлялись автоматически.

Шаблон для читателя

С точки зрения потребителя, важно разработать файлы указателей и представить всю документацию в удобном для чтения формате. Веб-страницы — лучший выбор. Они легко генерируются из файлов reStructuredText.

Резюме

В данной главе мы рассмотрели подход, позволяющий превратить управление документацией в организованный, легкий, эффективный и (надеемся) нескучный процесс. Одна из самых трудных задач при документировании проекта — поддержание документации в актуальном состоянии. Единственный способ сделать это — не считать документацию чем-то побочным, а обслуживать по высшему разряду. Хорошая документация всегда находится близко к коду. Если она является частью репозитория кода, то процесс значительно упрощается. В этом случае каждый раз, внося какие-либо изменения в код, разработчик должен изменить и соответствующую документацию.

Обеспечить точность также можно, объединив документацию с тестами через доктесты. Об этом поговорим в следующей главе, в которой рассказывается о принципах разработки на основе тестирования и о разработке на основе документов.

12 Разработка на основе тестирования

Разработка на основе тестирования (Test-Driven Development, TDD) — методология, направленная на производство высококачественного программного обеспечения путем создания автоматизированных тестов. Она широко используется в сообществе Python, но очень популярна и в других сообществах.

В Python тестирование особенно важно из-за его динамичного характера. В данном языке не хватает статической типизации, и из-за этого многие мелкие ошибки можно не заметить, если не запустить код и не проверить каждую его строку. Но проблема не только в том, как в Python работает типизация. Самые неприятные ошибки связаны не с плохим синтаксисом или использованием неправильного типа, а скорее с логическими и мелкими ошибками, которые могут привести к более серьезным проблемам.

Для эффективной проверки программного обеспечения часто бывает необходимо задействовать широкий спектр стратегий тестирования, которые будут выполняться на различных уровнях абстракции вашего приложения. К сожалению, не каждая стратегия подходит для любого проекта и не каждая система тестирования подходит для всех стратегий. Вот почему в крупных проектах часто используется несколько библиотек и стратегий тестирования, ориентированных на разные (зачастую дублирующие друг друга) задачи тестирования. Таким образом, чтобы лучше сориентировать вас в сложном мире тестирования ПО и разработки на основе тестирования, мы разделим эту главу на две части:

- *«Я не тестирую»* — содержит описание TDD и вкратце описывает, как реализовать это тестирование с помощью стандартной библиотеки;
- *«Я тестирую»* — предназначена для разработчиков, которые занимаются тестами и желают получить от них больше пользы.

Технические требования

Ниже перечислены пакеты Python, упомянутые в этой главе, которые можно скачать с PyPI:

- `pytest`;
- `nost`;
- `coverage`;
- `tox`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы примеров для этой главы можно найти по ссылке github.com/packtpublishing/expert-python-programming-third-edition/tree/chapter12.

Я не тестирую

Если вы уже уверовали в методологию TDD, то стоит перейти к следующему разделу. В нем мы поговорим о передовых методах и инструментах, упрощающих работу с тестами. Этот же раздел в основном предназначен для тех, кто не использует данный подход и тем самым совершает огромную ошибку.

Три простых шага разработки на основе тестирования

Процесс разработки на основе тестирования в своей простейшей форме состоит из таких трех этапов, как:

- написание автоматических тестов для новых функций или улучшений, которые еще не попали в релиз;
- написание минимального количества кода, выполняющего все заданные тесты;
- рефакторинг кода для достижения желаемых стандартов качества.

Самое важное, что следует помнить о данном цикле разработки, — тесты нужно писать до фактической реализации. Это довольно сложный для неопытных разработчиков подход, но он единственный надежно гарантирует, что код, который вы собираетесь писать, возможно протестировать.

Например, разработчик, который пишет функцию, проверяющую число на предмет того, является ли оно простым, может написать несколько примеров и привести ожидаемые результаты:

```
assert is_prime(5)
assert is_prime(7)
assert not is_prime(8)
```

Разработчик, реализующий данную функцию, не должен в одиночку думать о тестировании. Примеры могут быть предоставлены и другим человеком. Часто бывает так, что в официальных спецификациях сетевых протоколов или алгоритмов шифрования есть тестовые наборы, предназначенные для проверки правильности реализации. Это идеальная основа для тестирования кода, реализующего такие протоколы и алгоритмы.

Теперь можно работать над функцией, пока все примеры не заработают правильно:

```
def is_prime(number):
    for element in range(2, number):
        if number % element == 0:
            return False
    return True
```

Вполне возможно, что во время работы пользователи кода найдут баги или неожиданные результаты. Такие особые случаи тоже попадают в тесты, чтобы реализованная функция могла их обрабатывать:

```
>>> assert not is_prime(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Поскольку обнаружены новые проблемные примеры, функция постепенно улучшается следующим образом:

```
def is_prime(number):
    if number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

Этот процесс можно повторить несколько раз, поскольку зачастую бывает трудно предсказать все сложные примеры использования, допустим:

```
>>> assert not is_prime(-3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Суть этого подхода такова: реализация проходит через ряд постепенных улучшений. Данный процесс гарантирует, что функция правильно обрабатывает все возможные крайние случаи в пределах заданных ограничений по использованию:

```
def is_prime(number):
    if number < 0 or number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

Сбор известных или планируемых примеров использования выполняется для определения таких ограничений применения. Типичные примеры входят в состав тестов для реализованной функции, которая проверяет соответствие реализации всем известным требованиям. На практике наиболее часто встречающиеся примеры собираются в собственные именованные функции, чтобы те выполнялись каждый раз, когда код эволюционирует, следующим образом:

```
def test_is_prime():
    assert is_prime(5)
    assert is_prime(7)

    assert not is_prime(8)
    assert not is_prime(0)
    assert not is_prime(1)

    assert not is_prime(-1)
    assert not is_prime(-3)
    assert not is_prime(-6)
```

В предыдущем примере каждый раз, когда мы сталкиваемся с новым требованием, функция `test_is_prime()` должна обновляться, чтобы определить ожидаемое поведение функции `is_prime()`. Затем выполняется тест с целью проверить, дает ли новая реализация желаемые результаты. Если тесты не проходят, нужно обновить тестируемую функцию.

Разработка на основе тестирования имеет много преимуществ, включая следующие:

- помогает предотвратить регрессию программного обеспечения;
- улучшает качество программного обеспечения;
- дает своего рода низкоуровневую документацию о поведении кода;
- позволяет ускорить производство надежного кода и тем самым сократить циклы разработки.

Лучше всего работать с множественными тестами в Python так: собрать их все в одном модуле или пакете (как правило, под названием `tests`) и выработать способ запустить весь пакет с помощью одной команды оболочки. К счастью, нет

необходимости собирать всю цепочку тестирования самим. Стандартная библиотека Python и PyPI содержит много тестовых оболочек и утилит, которые позволяют создавать, открывать и запускать тесты в удобном виде. Далее в этой главе мы обсудим наиболее заметные примеры таких пакетов и модулей.

В следующих пунктах текста обсудим преимущества TDD.

Предотвращение регрессии программного обеспечения

Все разработчики рано или поздно сталкиваются с регрессией ПО. Это ошибка, возникшая после введения изменений. Она проявляется, когда функции, которые раньше работали нормально, приходят в негодность после внесения тех или иных изменений в проект.

Основная причина регрессии — высокая сложность программного обеспечения. В какой-то момент становится невозможно угадать, к чему приведет то или иное изменение в кодовой базе. Изменения в коде одной функции могут сломать некоторые другие функции, и иногда это приводит к плохим побочным эффектам, например к повреждению данных. Однако проблема высокой сложности касается не только огромных кодовых баз. Разумеется, очевидна корреляция между количеством кода и его сложностью, но даже небольшие проекты (несколько сотен/тысяч строк кода) могут иметь столь запутанную архитектуру, что бывает трудно предсказать все возможные последствия относительно небольших изменений.

Чтобы избежать регресса, нужно при каждом изменении проверять весь набор функций всей программы. Без этого вы не сможете надежно отличить уже существующие в коде ошибки от новых, которые появились вместе со свежими изменениями в коде.

Открытие кодовой базы сразу несколькими разработчиками усугубляет эту проблему, поскольку вероятно, что не все из них будут полностью в курсе всего происходящего в ходе разработки. И хотя наличие системы контроля версий предотвращает неожиданные конфликты, невозможно полностью предотвратить нежелательные взаимодействия.

TDD помогает уменьшить степень регрессии. Все программное обеспечение можно автоматически проверять после каждого изменения. Процесс будет работать, если у каждой функции есть собственный набор тестов. Когда TDD реализована правильно, набор тестов растет вместе с основной кодовой базой.

В результате выполнение полного теста может занять довольно много времени, и мы рекомендуем передать тестирование некой системе непрерывной интеграции, которая может выполнять эту работу в фоновом режиме. Мы уже обсудили такие решения в главе 10. Тем не менее каждый разработчик должен иметь возможность запускать тесты вручную, по крайней мере для соответствующих модулей. Если полностью положиться на систему непрерывной интеграции, то это плохо

скажется на производительности труда разработчиков. Программисты должны иметь возможность выполнять тесты в своей среде. Именно поэтому необходимо тщательно выбирать инструменты тестирования для проекта. Следует предпочитать тестовые фреймворки, которые позволяют легко выбирать и группировать нужные тесты.

Улучшение качества кода

При написании кода мы в основном думаем об алгоритмах, структурах данных и производительности, но часто забываем о точке зрения пользователя — как и когда станут применяться наши функция, класс или модуль? Действительно ли легко и логично используются аргументы? Правильные ли имена в этом новом API? Легко ли расширить код в будущем?

Такие качества легко обеспечить с помощью советов, описанных в предыдущих главах, скажем в главе 6. Но единственный способ сделать это эффективно — написать примеры использования. Именно в этот момент вы поймете, насколько ваш код логичен и прост в применении. Часто первый рефакторинг происходит сразу после завершения модуля, класса или функции.

Написание тестов, которые являются примерами применения кода, помогает не забывать о точке зрения пользователя. Обращение к самой этой методике позволяет вам производить лучший код. Трудно тестировать гигантские функции и огромные монолитные классы. Код, написанный с учетом тестирования, как правило, имеет более чистую модульную архитектуру.

Предоставление лучшей документации для разработчика

Тесты — лучший для разработчика способ узнать, как функционирует программа. Это именно те случаи использования, для которых и был создан код. Читая их, можно быстро, но в то же время глубоко понять, как работает код. Иногда такой пример дороже тысячи слов.

Если эти тесты всегда обновляются вместе с кодовой базой, то документация для разработчиков получается выше всяких похвал. Тесты всегда должны быть актуальными, в отличие от текстовой документации, иначе возможны проблемы.

Быстрая разработка надежного кода

Написание кода без тестирования приводит к лишним затратам времени на отладку. Последствия ошибки в одном модуле могут проявиться в совершенно других частях программы. Поскольку вы не знаете, кто виноват, слишком много времени уходит на отладку. Бороться с небольшими ошибками проще сразу после провала теста, поскольку тогда вы будете лучше понимать, где именно проблема.

Тестирование часто приносит больше удовольствия, чем отладка, ведь это тоже своего рода программирование.

Если измерить время, необходимое на исправление кода и на его написание, то обычно оно превышает время, затраченное на подход TDD. Когда вы начинаете писать новую часть кода, это неочевидно. Так происходит потому, что время, необходимое для создания тестовой среды и написания первых нескольких тестов, чрезвычайно велико по сравнению со временем, которое требуется для написания первой части кода.

О каких тестах речь

Существует несколько видов тестов, которые работают с любым программным обеспечением. Основные виды — *модульные*, *приемочные* и *функциональные тесты*. Именно о них идет речь во время тестирования программного обеспечения. Но есть и другие виды тестов, которые вы можете использовать в вашем проекте, например *интеграционные тесты*, *тесты нагрузки и производительности*, а также *тесты качества кода*. Мы поговорим о них далее.

Модульные тесты

Модульные тесты (юнит-тесты) — низкоуровневые тесты, которые идеально подходят для TDD. Как следует из названия, они сосредоточены на тестировании модулей программы. Под модулем можно понимать наименьшую тестируемую часть кода приложения. В зависимости от целей применения тестировать можно весь модуль или отдельные функции, но обычно модульные тесты пишутся для минимально возможных фрагментов кода. Модульные тесты, как правило, отделяют тестируемый фрагмент (модуль, класс, функцию и т. д.) от остальной части приложения и других элементов. Если при этом необходимы внешние зависимости, такие как веб-интерфейсы или базы данных, то они часто заменяются болванками.

Приемочные тесты

Приемочные тесты фокусируются на функциональности, считая саму программу черным ящиком. Они проверяют только, действительно ли программное обеспечение работает должным образом, применяя те же средства, что и у пользователей, и наблюдают за выводом приложения. Эти тесты иногда пишутся обособленно от основного цикла разработки, чтобы проверить соответствие приложения требованиям. Они обычно работают в виде контрольных листов для программы. Часто эти тесты не проходят через TDD и составляются менеджерами, сотрудниками по качеству или даже клиентами. В таком случае они часто называются *пользовательскими приемочными тестами*.

Тем не менее принципы TDD забывать не стоит. Приемочные тесты могут быть предоставлены до написания самой функции. Разработчики получают кучу приемочных тестов, составленных, как правило, из функциональных спецификаций, и работа этих людей состоит в получении уверенности в том, что код пройдет их все.

Средства, с помощью которых пишутся эти тесты, зависят от пользовательского интерфейса программного обеспечения. Некоторые популярные инструменты, применяемые разработчиками на Python, приведены в табл. 12.1.

Таблица 12.1. Популярные инструменты для написания приемочных тестов

Тип приложения	Инструмент
Веб-приложение	Selenium (для веб-интерфейсов с JavaScript)
Веб-приложение	zope.testbrowser (не тестирует JS)
Приложение WSGI	paste.test.fixture (не тестирует JS)
Приложение Gnome	dogtail
Приложение Win32	pywinauto

Большой список функциональных инструментов для тестирования приведен на странице PythonTestingToolsTaxonomy на «Вики»-сайте Python: wiki.python.org/moin/PythonTestingToolsTaxonomy.

Функциональные тесты

Функциональные тесты работают с большими частями функционала, а не с небольшими блоками кода. По назначению эти тесты похожи на приемочные. Основное отличие таково: функциональные тесты не обязательно должны задействовать тот же интерфейс, что и пользователь. Например, при тестировании веб-приложений некоторые действия пользователей (или их последствия) можно смоделировать с помощью синтетических запросов HTTP или прямого доступа к базе данных, а не имитируя реальную загрузку страницы и щелчки кнопкой мыши.

Такой подход часто работает проще и быстрее, чем тестирование с помощью инструментов, которые используются в *приемочных испытаниях пользователя*. Недостаток ограниченных функциональных тестов — они, как правило, не работают со всеми слоями и уровнями абстракции приложения. Тесты, которые сосредотачиваются на таких точках *пересечения*, часто называют *интеграционными*.

Интеграционные тесты

Эти тесты представляют собой более высокий уровень тестирования, чем модульные. В них тестируются большие части кода, и акцент ставится на ситуациях, когда несколько слоев или компонентов приложений встречаются и взаимодействуют. Форма и объем интеграционных тестов варьируется в зависимости от архитектуры и сложности проекта. Например, в небольших и монолитных проектах эти тесты не сложнее функциональных и позволяют им взаимодействовать с реальными бэкенд-сервисами (базами данных, кэшем и т. д.), а не болванками. Для сложных сценариев или продуктов, построенных из нескольких сервисов, реальные интеграционные тесты могут быть весьма обширными и порой даже требуют запуска всего проекта в большой распределенной среде, которая повторяет продакшен.

Интеграционные тесты часто очень похожи на функциональные, а граница между ними размыта. Достаточно часто интеграционные тесты затрагивают только отдельные функции.

Нагрузочные тесты и тесты производительности

Нагрузочные тесты и тесты производительности дают объективную информацию об эффективности кода, а не о его правильности. Сами термины «нагрузочные тесты» и «тесты производительности» часто взаимозаменяемы, но в первом случае имеется в виду ограниченная производительность. Нагрузочный тест призван измерить, как поведет себя код под некой искусственной нагрузкой. Это очень популярный способ тестирования веб-приложений, где под нагрузкой понимается веб-трафик от реальных пользователей или программных клиентов. Важно отметить, что нагрузочные тесты в веб-приложениях, как правило, используются для HTTP-транзакций. Это весьма сближает их с интеграционными и функциональными тестами. Таким образом, прежде чем приступать к нагрузочному тестированию, очень важно убедиться, что тестируемое приложение уже полностью проверено и работает правильно. Тесты производительности — это обычно тесты, которые направлены на измерение производительности кода и могут охватывать даже небольшие блоки кода. Следовательно, нагрузочные тесты — подмножество тестов производительности.

Оба вида особенные по-своему, поскольку не дают бинарного результата (успех/провал), а лишь возвращают некую меру качества производительности. Это значит, что полученные результаты нужно интерпретировать в сравнении с результатами других прогонов. В некоторых случаях требования проекта определяют жесткие ограничения на нагрузку или производительность кода, но это не меняет того факта, что в таком тестировании всегда существует доля произвольной интерпретации.

Нагрузочные тесты и тесты производительности — отличные инструменты разработки любого программного обеспечения, которое должно соответствовать требованиям некоего *соглашения об уровне обслуживания* (например, время работы в процентах, количество единовременно поддерживаемых соединений), поскольку это помогает снизить риск ухудшения производительности критически важных точек кода.

Тестирование качества кода

Не существует конкретной шкалы, которая бы позволила точно сказать, хорош код или плох. К сожалению, абстрактное понятие качества кода нельзя выразить в числах. Вместо этого мы можем измерять различные метрики программного обеспечения, насколько нам известно, коррелирующие с качеством кода. Ниже приведены некоторые из них:

- количество нарушений стиля кода;
- объем документации;
- сложность метрики, например цикломатическая сложность Маккейба;
- количество предупреждений при статическом анализе кода.

Во многих проектах тестирование качества кода используется в процессе непрерывной интеграции. Хороший и популярный подход заключается в том, чтобы тестировать по крайней мере базовые метрики (статический анализ и стиль) и не допускать слияния любого кода с основным потоком, если вследствие этого показатели снижаются.

В следующем разделе мы рассмотрим некоторые основные инструменты тестирования из стандартной библиотеки Python, позволяющие реализовать множество различных типов программных тестов.

Стандартные инструменты тестирования в Python

В стандартной библиотеке Python есть два простых модуля, которые позволяют писать автоматизированные тесты:

- `unittest` (docs.python.org/3/library/unittest.html) — стандартный и наиболее распространенный фреймворк тестирования Python на основе JUnit Java, написанный Стивом Перселлом (бывший `PyUnit`);
- `doctest` (docs.python.org/3/library/doctest.html) — инструмент для тестирования программ с интерактивными примерами использования.

Рассмотрим оба модуля.

Модуль unittest

Модуль `unittest` делает в основном то же, что JUnit выполняет для Java. В нем есть базовый класс под названием `TestCase`, имеющий обширный набор методов для проверки вывода функций и операторов. Это наиболее простая и часто используемая библиотека тестирования Python, которая нередко служит основой для более сложных механизмов тестирования.

Этот модуль был создан в расчете на модульные тесты, но вы можете применять его и для других видов тестов. Можно даже задействовать его при приемочном тестировании потоков с интеграцией слоя пользовательского интерфейса, поскольку в некоторых библиотеках тестирования есть справка по наложению инструментов, например Selenium поверх `unittest`.

Написание простого модульного теста с помощью `unittest` осуществляется подклассом `TestCase` и записью методов с префиксом `test`. Образец модуля с примерами из раздела о TDD будет выглядеть следующим образом:

```
import unittest

from primes import is_prime

class MyTests(unittest.TestCase):
    def test_is_prime(self):
        self.assertTrue(is_prime(5))
        self.assertTrue(is_prime(7))

        self.assertFalse(is_prime(8))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))

        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-3))
        self.assertFalse(is_prime(-6))

if __name__ == "__main__":
    unittest.main()
```

Функция `unittest.main()` — утилита, которая позволяет сделать весь модуль исполняемым в виде тестового набора следующим образом:

```
$ python test_is_prime.py -v
test_is_prime (__main__.MyTests) ... ok

----------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Функция `unittest.main()` просматривает контекст текущего модуля и ищет классы с подклассом `TestCase`. Модуль конкретизирует их, а затем запускает все методы, которые начинаются с префикса `test`.

Хороший набор тестов должен соблюдать соглашения об именовании. Например, если функция `is_prime` входит в модуль `primes.py`, то тестовый класс можно назвать `PrimesTests` и поместить в файл `test_primes.py`, как показано ниже:

```
import unittest

from primes import is_prime

class PrimesTests(unittest.TestCase):
    def test_is_prime(self):
        ...

if __name__ == '__main__':
    unittest.main()
```

Теперь при каждом изменении модуля `primes` в `test_primes` попадает больше тестов.

Модулю `test_primes` для работы нужен доступный в контексте модуль `primes`. Это реализуется либо помещением обоих модулей в один пакет, либо путем явного добавления тестируемого модуля в путь Python. На практике здесь бывает полезна команда `develop` из `setuptools` (см. главу 7).

Выполнение тестов на всем приложении предполагает наличие сценария, который строит *тестовую кампанию* из всех тестовых модулей. В `unittest` есть класс `TestSuite`, позволяющий агрегировать тесты и запускать их в качестве таковой кампании, если все они являются экземплярами `TestCase` или `TestSuite`.

В прошлом Python существовало соглашение о том, что тестовый модуль должен иметь функцию `test_suite`, которая возвращает `TestSuite`. Эта функция будет использоваться в разделе `__main__`, когда модуль вызывается в командной строке или автоматически выполняется тестером, как показано ниже:

```
import unittest

from primes import is_prime

class PrimesTests(unittest.TestCase):
    def test_is_prime(self):
        self.assertTrue(is_prime(5))

        self.assertTrue(is_prime(7))

        self.assertFalse(is_prime(8))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))
```

```
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-3))
        self.assertFalse(is_prime(-6))

class OtherTests(unittest.TestCase):
    def test_true(self):
        self.assertTrue(True)

def test_suite():
    """Сборка тестовой среды"""
    suite = unittest.TestSuite()
    suite.addTests(unittest.makeSuite(PrimesTests))
    suite.addTests(unittest.makeSuite(OtherTests))

    return suite

if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

При запуске модуля из оболочки вы получите следующий вывод теста:

```
$ python test_primes.py -v
test_is_prime (__main__.PrimesTests) ... ok
test_true (__main__.OtherTests) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Предыдущий подход использовался в более старых версиях Python, когда модуль `unittest` не располагал нормальными утилитами для обнаружения тестов. Как правило, запуск всех тестов выполнялся с помощью глобального сценария, который просматривал дерево кода и искал тесты для запуска. Этот процесс часто называли *поиском теста*, и он будет рассмотрен более подробно далее в главе. На данный момент вам достаточно знать только то, что в `unittest` есть простая команда, позволяющая обнаружить все тесты из модулей и пакетов, в имени которых есть префикс `test`:

```
$ python -m unittest -v
test_is_prime (test_primes.PrimesTests) ... ok
test_true (test_primes.OtherTests) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Если вы используете предыдущую команду, то нет необходимости вручную определять раздел `__main__` и вызывать функцию `unittest.main()`.

Модуль doctest

Модуль `doctest` извлекает тестовые фрагменты из строк документации или текстовых файлов в виде интерактивных быстрых сессий, а также воспроизводит их, чтобы проверить, совпадает ли заданный вывод с реальным.

Например, в качестве теста можно использовать текстовый файл со следующим содержимым:

```
Check addition of integers works as expected::

>>> 1 + 1
    2
```

Предположим, что этот файл документации хранится в файловой системе под именем `test.rst`. Модуль `doctest` предоставляет функции для извлечения и запуска тестов из таких файлов документации, а именно:

```
>>> import doctest
>>> doctest.testfile('test.rst', verbose=True)
Trying:
    1 + 1
Expecting:
    2
ok
1 items passed all tests:
   1 tests in test.rst
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=1)
```

Использование `doctest` дает следующие преимущества:

- пакеты можно документировать и тестировать через примеры;
- примеры документации всегда поддерживаются в актуальном состоянии;
- использование примеров в виде доктестов помогает не забывать о точке зрения пользователя.

Однако применение доктестов не означает, что модульные тесты устарели. Они должны применяться только для приведения читабельных примеров в документах или строках документации. Иными словами, когда тесты касаются вопросов низкого уровня или нуждаются в сложных испытательных тестовых установках, доктесты использовать не надо.

В ряде структур, таких как Python Zope, доктесты используются активно и порой не нравятся тем, кому программирование в новинку. Некоторые доктесты действительно трудно читать и понимать, поскольку примеры нарушают одно из правил технической документации — их нельзя просто взять и выполнить, ввиду

того что нужны обширные знания. Таким образом, документы, которые должны помогать новичкам, в действительности оказывается трудно читать, если примеры кода основаны на сложных текстах или API для тестирования.

При использовании доктестов, которые являются частью документации ваших пакетов, не забывайте соблюдать семь правил технической документации, рассмотренных в главе 11.

Я тестирую

Раздел «Я не тестирую» должен был познакомить вас с основами разработки на основе тестирования, но есть еще несколько вещей, которым вы должны научиться, прежде чем сможете эффективно использовать эту методологию.

В этом разделе мы опишем ряд проблем, с которыми сталкиваются разработчики при написании тестов, а также способы их решения. Здесь же приведен краткий обзор популярных тестеров и инструментов, доступных в сообществе Python.

Ловушки модуля unittest

Этот модуль появился в Python 2.1 и с тех пор массово используется разработчиками. Однако люди, разочарованные слабостями и ограничениями `unittest`, создали альтернативные механизмы тестирования.

Ниже приведены самые распространенные недостатки.

- Нужен обязательный префикс `test` у имен методов.
- Поощряется использовать методы, приведенные в `TestCase`, вместо простых операторов `assert`, поскольку существующие методы не охватывают каждый случай применения.
- Фреймворк плохо расширяется, поскольку требует много подклассов классов и других трюков, таких как декораторы.
- Тестовые «принадлежности» иногда трудно организовать, поскольку все привязано к уровню `TestCase`, хотя они выполняются всего один раз за тест. То есть в случае использования большого количества тестовых модулей организовать создание и очистку будет непросто.
- Неудобно запускать тестовую кампанию. Тестер по умолчанию (`python -m unittest`) действительно выполняет поиск тестов, но имеет мало возможностей фильтрации. Приходится писать дополнительные скрипты, чтобы собрать тесты и запустить их в удобном виде.

Для написания тестов без страданий от сложности всей этой структуры необходим более легкий подход, поскольку иначе мы получаем тот же JUnit. Поскольку

в Python не требуется работа на 100 % основанной на классах среде, желательно придумать фреймворк, не основанный на подклассах в первую очередь.

В хорошую структуру должно входить следующее:

- простой способ, чтобы назначить какую-либо функцию или класс тестом;
- расширяемость через систему плагинов;
- готовая и полная тестовая среда для всех уровней тестов: для всей кампании, на уровнях модуля и одного теста;
- тестер с поиском тестов и широким набором опций.

Альтернативы модулю unittest

В ряде инструментов сторонних производителей делались попытки решить проблемы, которые мы упоминали ранее, через дополнительные возможности в виде расширений `unittest`.

На «Вики»-сайте Python приведен длинный список различных тестовых утилит и фреймворков (wiki.python.org/moin/PythonTestingToolsTaxonomy), но лишь два проекта завоевали популярность:

- `nose` (задокументирован на nose.readthedocs.org);
- `py.test` (задокументирован на pytest.org).

Тестер nose

Это тестер с мощными поисковыми функциями. Его возможности позволяют выполнять все виды тестов в приложениях Python.

Тестер не является частью стандартной библиотеки, но доступен на PyPI и легко устанавливается с помощью одной команды `pip`:

```
pip install nose
```

Далее мы рассмотрим тестер `nose`, его систему, а также интеграцию с инструментами установки и систему плагинов.

Тестирование. После установки `nose` у вас в командной строке появится новая команда под названием `nosetests`. Выполнять тесты, представленные в первой части этой главы, можно напрямую оттуда:

```
$ nosetests -v
test_true (test_primes.OtherTests) ... ok
test_is_prime (test_primes.PrimesTests) ... ok
builds the test suite. ... ok

----------------------------------------------------------------
Ran 3 tests in 0.009s

OK
```

Поиск тестов в nose реализован через рекурсивный поиск по текущему рабочему каталогу и автоматическое создание тестового набора. На первый взгляд, предыдущий пример мало чем лучше команды `python -m unittest`. Реальная разница становится заметной, когда вы выполняете `nosetests` с переключателем `--help`. Вы увидите, что nose предоставляет десятки параметров, которые позволяют очень тонко настраивать выполнение теста.

Написание тестов. Тестер nose выполняет тесты еще более эффективно по сравнению с UnitTest, когда проходит через все классы и функции, чье имя совпадает с регулярным выражением `((?:^|[b_.-])[Tt]est)`. Он ищет в модулях блоки, чьи имена совпадают с этим выражением. Грубо говоря, все вызываемые объекты с именами, начинающимися с `test`, и расположенные в модуле, который соответствует этому шаблону, также будут выполнены в виде теста.

Например, такой модуль `test_ok.py` вполне подойдет для nose:

```
$ cat test_ok.py
def test_ok():
    print('my test')

$ nosetests -v
test_ok.test_ok ... ok

----------------------------------------------------------
Ran 1 test in 0.071s

OK
```

Обычные классы `TestCase` и `doctests` тоже будут работать нормально.

И последнее, но не по значению: nose предоставляет функции утверждения, похожие на те, что предусмотрены в методах класса `unittest.TestCase`. Но у этих функций имена следуют соглашениям об именовании PEP 8, а не Java, как в `unittest`.

Написание тестовой установки. Фреймворк nose поддерживает следующие три уровня тестовых механизмов:

- *уровень пакета* — функции `setup` и `teardown` могут быть добавлены в модуль `__init__` тестового пакета, содержащего все тестовые модули;
- *уровень модуля* — у модуля могут быть свои функции `setup` и `teardown`;
- *уровень теста* — вызываемые объекты также могут иметь свои такие функции при наличии декоратора `@with_setup()`.

Например, чтобы установить тест на уровне модуля и теста, используйте этот код:

```
def setup():
    # код установки, запускаемый для всего модуля
    ...

def teardown():
    # код удаления, запускаемый для всего модуля
    ...

def set_ok():
    # код установки, используемый по требованию с помощью декоратора
    ...

@with_setup(set_ok)
def test_ok():
    print('my test')
```

Интеграция с setuptools и системой плагинов. Фреймворк `nose` хорошо интегрируется с `setuptools`, и вы можете задействовать команду `setup.py test` для вызова всех тестов. Такая интеграция чрезвычайно полезна в экосистемах пакетов, которые требуют общей точки входа в тест, но могут применять различные фреймворки.

Такая интеграция осуществляется путем добавления метаданных `test_suite` в скрипт `setup.py`:

```
setup(
    ...
    test_suite='nose.collector',
)
```

В `nose` также используется точка входа `setuptools` для разработчиков плагинов. Это позволяет переопределить или изменить основные аспекты инструмента, например алгоритм поиска тестов или выходное форматирование.

Подытожим. Модуль `nose` — полноценный набор инструментов для тестирования, в котором устранены многие из проблем `unittest`. В нем по-прежнему используются неявные имен префиксов для испытаний, что остается сдерживающим фактором для некоторых разработчиков. Этот префикс можно изменить, но соглашение об именовании все равно должно соблюдаться.

Это соглашение в конфигурации является не таким уж плохим и намного лучше шаблонного кода `unittest`. Но использование явных декораторов может быть хорошим способом избавиться от префикса `test`.

Кроме того, возможность расширения с помощью плагинов делает `nose` очень гибким и позволяет настроить этот инструмент под свои потребности.

Если ваш рабочий процесс тестирования требует переопределения многих параметров `nose`, то вы можете легко добавить файл `.noserc` или `nose.cfg` в корневой

каталог вашего проекта. В этом файле определен набор параметров по умолчанию для команды `nosetests`. Так, рекомендуется автоматически искать `doctests` во время тестового прогона. Пример файла конфигурации `nose`, который позволяет выполнять доктесты, выглядит следующим образом:

```
[nosetests] with-doctest=1 doctest-extension=.txt
```

Модуль py.test

Модуль `py.test` очень похож на `nose`. На самом деле последний был вдохновлен именно `py.test`, поэтому мы остановимся лишь на их различиях. Инструмент был рожден как часть более крупного пакета под названием `py`, но теперь они разрабатываются отдельно.

Как и любой сторонний пакет, упомянутый в этой книге, `py.test` доступен на PyPI и устанавливается с помощью команды `pip` как `pytest`:

```
$ pip install pytest
```

После этого у вас появится новая команда `py.test`, доступная в вашей оболочке, которая работает так же, как `nosetests`. Инструмент использует похожие алгоритмы сопоставления модели и обнаружения тестов и ищет тесты в вашем проекте. Поисковик работает более строго и ищет только следующее:

- классы, начинающиеся со слова `Test`, в файле, который начинается со слова `test`;
- функции, начинающиеся со слова `test`, в файле, который начинается со слова `test`.

Важно использовать правильный регистр. Если функция начинается с заглавной T, то будет принята за класс и проигнорирована. Если же класс начинается со строчной t, то py.test сломается, поскольку попытается обработать класс как функцию.

Преимущества `py.test` над другими системами таковы:

- возможность легко отключать выбранные классы тестов;
- гибкий и оригинальный механизм для работы с установкой;
- встроенная возможность распределять тесты между несколькими компьютерами.

В следующих подпунктах мы рассмотрим написание тестовых установок, отключение тестовых функций и классов, а также автоматизированные распределенные тесты в `py.test`.

Написание тестовой установки. В `py.test` поддерживается два механизма работы с установками. Первый из них, по образцу фреймворка XUnit, похож на

nose. Конечно, семантика немного отличается. Модуль py.test просматривает три уровня в каждом тестовом модуле, как показано в следующем примере, взятом из официальной документации:

```
def setup_module(module):
    """Установка состояния, присущего выполнению
       данного модуля
    """

def teardown_module(module):
    """Удаление состояния, ранее заданного методом
       setup_module
    """

def setup_class(cls):
    """Установка состояния, присущего выполнению
       данного класса (обычно с тестами)
    """

def teardown_class(cls):
    """Удаление состояния, ранее заданного вызовом
       setup_class
    """

def setup_method(self, method):
    """Установка состояния с привязкой к выполнению данного
       метода в классе setup_method вызывается
       для каждого тестового метода класса
    """

def teardown_method(self, method):
    """Удаление состояния, ранее заданного методом
       вызовом setup_method
    """
```

Каждая функция получает приемный модуль, класс или метод в качестве аргумента. Тогда тестовая установка будет иметь возможность работать с контекстом, не прибегая к необходимости искать его, как это происходит в случае с nose.

Другой механизм для написания установок с помощью py.test построен на концепции внедрения зависимостей и позволяет поддерживать тестовое состояние более модульным и масштабируемым способом. Установки в стиле, отличном от XUnit (процедуры установки/демонтажа), всегда имеют уникальные имена и должны явно активироваться через объявление в тестовых функциях, методах и модулях.

Простейшая реализация установки имеет форму именованной функции, объявленной с декоратором pytest.fixture(). Чтобы обозначить установку как используемую в тесте, ее следует объявить как функцию или аргумент метода.

С целью упростить понимание рассмотрим предыдущий пример тестового модуля для функции `is_prime`, переписанный следующим образом с применением `py.test`:

```
import pytest

from primes import is_prime

@pytest.fixture()
def prime_numbers():
    return [3, 5, 7]

@pytest.fixture()
def non_prime_numbers():
    return [8, 0, 1]

@pytest.fixture()
def negative_numbers():
    return [-1, -3, -6]

def test_is_prime_true(prime_numbers):
    for number in prime_numbers:
        assert is_prime(number)

def test_is_prime_false(non_prime_numbers, negative_numbers):
    for number in non_prime_numbers:
        assert not is_prime(number)

    for number in negative_numbers:
        assert not is_prime(number)
```

Отключение тестовых функций и классов. В `py.test` предусмотрен простой механизм, позволяющий отключить некоторые тесты при определенных условиях. Он называется *пропуском теста*, и в пакете `pytest` для этого есть декоратор `@mark.skipif`. Если при определенных условиях нужно пропустить одну функцию или целый декоратор класса, то вы должны определить его с помощью этого декоратора и некоего выражения, которое проверяет, было ли выполнено ожидаемое условие. Вот пример из официальной документации, где пропускается целый класс, если тестовый набор выполняется на Windows:

```
import pytest

@pytest.mark.skipif(
    sys.platform == 'win32',
    reason="does not run on windows"
)
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

Вы можете предопределить условие пропуска, чтобы использовать тест на всех модулях:

```
import pytest

skipwindows = pytest.mark.skipif(
    sys.platform == 'win32',
    reason="does not run on windows"
)

@skip_windows
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

Если тест пометить подобным образом, то выполняться не будет. Однако иногда вам хочется выполнить конкретный тест, который выдает ошибку при известных условиях. Для этой цели служит другой декоратор. Он называется `@mark.xfail` и гарантирует, что тест запускается всегда, но должен потерпеть неудачу при определенном условии:

```
import pytest

@pytest.mark.xfail(
sys.platform == 'win32',
    reason="does not run on windows"
)
class TestPosixCalls:

    deftest_function(self):
        "it must fail under windows"
```

Правила использования `xfail` гораздо более строги в сравнении с `skipif`. Тест выполняется всегда, и если не терпит неудачу, когда это от него ожидается, то все тестирование в целом будет считаться ошибочным.

Автоматизированные распределенные тесты. Интересная особенность `py.test` — его способность распределять тесты на несколько компьютеров. Если компьютер доступен через SSH, то `py.test` будет иметь возможность управлять каждым компьютером, посылая на него нужные тесты.

Тем не менее эта функция зависит от наличия сети, и при разрыве соединения ведомый не сможет продолжать работу, поскольку полностью управляется ведущим.

BuildBot и другие инструменты непрерывной интеграции будут более предпочтительны, если в проекте предусмотрены большие и длительные тестовые кампании. Однако распределенная модель `py.test` может быть использована для динамического распределения тестов при отсутствии налаженной системы непрерывной интеграции.

Подытожим. Инструмент `py.test` очень похож на `nose`, поскольку его работа не требует наличия шаблонного кода для сбора тестов воедино. В нем также есть хорошая система плагинов и большое количество расширений на PyPI.

Модуль `py.test` фокусируется на том, чтобы тесты выполнялись быстро, и в этом он действительно куда эффективнее всех прочих инструментов в данной области. Другая примечательная особенность — оригинальный подход к созданию тестовой установки, который помогает управлять многоразовой библиотекой установок. Некоторым кажется, что здесь чересчур много магии, но это действительно упрощает разработку тестовых наборов. Данное преимущество `py.test` покорило наши сердца, и мы очень рекомендуем этот модуль.

В следующем подразделе вы узнаете о способах определения того, какую часть кода охватывать.

Охват тестирования

Охват кода — очень полезный показатель, который дает объективную информацию о том, насколько тщательно тестируется код проекта. Это мера того, сколько и каких именно строк кода выполняется во время теста. Показатель часто выражается в процентах, и охват 100 % означает, что был выполнен абсолютно весь код.

Самый популярный инструмент в этой области — `coverage`, и его можно свободно скачать с PyPI. В использовании он весьма прост — всего два шага. Первый заключается в выполнении команды `coverage run` в вашей оболочке с указанием пути к скрипту, который запускает все тесты:

```
$ coverage run --source . `which py.test` -v
==================== test session starts ======================
platformdarwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1 --
/Users/swistakm/.envs/book/bin/python3 cachedir: .cache rootdir:
/Users/swistakm/dev/book/chapter10/pytest, inifile:  plugins:
capturelog-0.7, codecheckers-0.2, cov-2.2.1, timeout-1.0.0 collected 6
items  primes.py::pyflakes PASSED
primes.py::pep8 PASSED
test_primes.py::pyflakes PASSED
test_primes.py::pep8 PASSED
test_primes.py::test_is_prime_true PASSED
test_primes.py::test_is_prime_false PASSED
========= 6 passed, 1 pytest-warnings in 0.10 seconds ==========
```

Команда также принимает параметр `-m`, в котором можно задать имя исполняемого модуля вместо пути программы, что бывает удобно для отдельных фреймворков тестирования:

```
$ coverage run -m unittest
$ coverage run -m nose
$ coverage run -m pytest
```

Следующий шаг — генерация читабельного отчета в виде результатов в файле `.coverage`. Пакет `coverage` поддерживает несколько форматов вывода, и самый простой из них — таблица ASCII в консоли:

```
$ coverage report
Name             StmtsMiss  Cover
----------------------------------
primes.py            7      0   100%
test_primes.py      16      0   100%
----------------------------------
TOTAL               23      0   100%
```

Еще один полезный формат отчета — HTML, который можно открыть в вашем браузере:

```
$ coverage html
```

Выходная папка для HTML-отчета по умолчанию — `htmlcov/` в рабочем каталоге. Реальное преимущество вывода `coverage html` заключается в том, что вы можете просматривать код проекта с аннотациями, причем будут выделены части, для которых отсутствует тестовое покрытие (как показано на рис. 12.1).

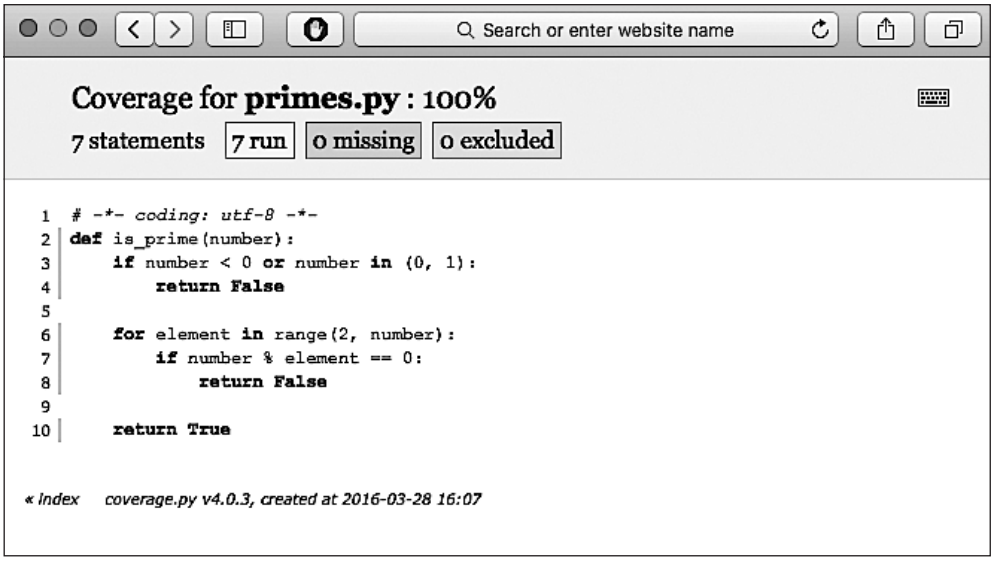


Рис. 12.1. Пример аннотированного охвата кода в отчете HTML

Вам стоит знать, что, хоть желаемый охват и стремится к 100 %, это никогда не гарантирует безупречной проверки и бесперебойной работы кода. Это лишь означает, что каждая строка кода выполнилась во время теста, но не обязательно

были проверены все возможные условия. На практике бывает относительно легко реализовать полный охват кода, но действительно сложно добиться тестирования каждой ветви кода. Это особенно верно в отношении проверки функций, в которых есть несколько комбинаций операторов `if` и специфичных языковых конструкций, таких как `list/dist/set`-включения. Всегда стоит стремиться к хорошему охвату тестирования, однако не стоит в то же время думать, что одного этого показателя достаточно для идеального кода.

Далее мы рассмотрим другие методы тестирования и инструменты, которые позволяют заменить существующие сторонние зависимости во время тестов на объекты, имитирующие их поведение.

Подделки и болванки

Написание тестов предполагает, что вы изолируете блок кода, который нужно протестировать. В тестах, как правило, функции или методу передаются некие данные и проверяется возвращаемое значение или иной результат выполнения. Такой подход позволяет убедиться, что тесты:

- работают с мельчайшими частями приложения, то есть функциями, методами, классами или интерфейсами;
- выдают детерминированные и воспроизводимые результаты.

Иногда неочевидно, как правильно изолировать компоненты. Например, в случае отправки кодом сообщений по электронной почте, вероятно, используется модуль `smtplib`, который будет работать с SMTP-сервером через сетевое соединение. Если мы хотим, чтобы наши тесты были воспроизводимыми, и проверяем только контент письма, то реальное сетевое подключение нам, возможно, и не требуется. В идеале модульные тесты должны работать на любом компьютере без каких-либо внешних зависимостей и побочных эффектов.

Благодаря динамической природе языка Python вы можете использовать специальные *заплатки*, чтобы модифицировать код во время выполнения прямо из тестовой установки (то есть изменять ПО динамически, не трогая при этом исходный код) с целью сымитировать поведение стороннего кода или библиотеки. Далее мы узнаем, как создавать такие объекты.

Займемся подделкой

Подделку для тестов можно создать путем воспроизведения минимального набора взаимодействий, необходимых для работы тестируемого кода с внешними частями. Затем выход возвращается вручную или через реальный пул данных, которые были записаны ранее.

Начать можно с создания пустого класса или функции, которую затем можно использовать в качестве замены для имитируемого нами компонента. Затем вы можете шаг за шагом обновить определение класса, пока подделка не начнет вести себя как задумано. Это возможно благодаря природе системы типов в Python. Объект считается совместимым с данным типом, если тот ведет себя ожидаемым образом, и, как правило, связь с этим типом с помощью подклассов не требуется. Такой подход к типизации в Python называется утиной типизацией: *если что-то ходит как утка и говорит как утка — значит, это утка*.

Рассмотрим пример модуля mailer с функцией `send`, который отправляет электронную почту с помощью библиотеки `smtplib`:

```
import smtplib
import email.message

def send(
    sender, to,
    subject='None',
    body='None',
    server='localhost'
):
    """Отправить сообщение"""
    message = email.message.Message()
    message['To'] = to
    message['From'] = sender
    message['Subject'] = subject
    message.set_payload(body)

    server = smtplib.SMTP(server)
    try:
        return server.sendmail(sender, to, message.as_string())
    finally:
        server.quit()
```

Для иллюстрации подделок мы будем использовать py.test.

Соответствующий тест можно записать следующим образом:

```
from mailer import send

def test_send():
    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

Этот тест будет работать, если на локальном хосте поднят SMTP-сервер. В противном случае ничего не выйдет:

```
$ py.test --tb=short
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: /Users/swistakm/dev/book/chapter10/mailer, inifile:
plugins: capturelog-0.7, codecheckers-0.2, cov-2.2.1, timeout-1.0.0
collected 5 items
mailer.py ..
test_mailer.py ..F
============================== FAILURES ==============================
_____________________________ test_send _____________________________
test_mailer.py:10: in test_send
    'body'
mailer.py:19: in send
    server = smtplib.SMTP(server)
.../smtplib.py:251: in __init__
    (code, msg) = self.connect(host, port)
.../smtplib.py:335: in connect
    self.sock = self._get_socket(host, port, self.timeout)
.../smtplib.py:306: in _get_socket
    self.source_address)
.../socket.py:711: in create_connection
    raise err
.../socket.py:702: in create_connection
    sock.connect(sa)
E   ConnectionRefusedError: [Errno 61] Connection refused
======== 1 failed, 4 passed, 1 pytest-warnings in 0.17 seconds ========
```

Тогда в подделку класса SMTP можно добавить небольшой патч:

```
import smtplib
import pytest
from mailer import send

class FakeSMTP(object):
    pass

@pytest.yield_fixture()
def patch_smtplib():
    # Установка: monkey patch smtplib
    old_smtp = smtplib.SMTP
    smtplib.SMTP = FakeSMTP

    yield
    # Шаг удаления: возврат smtplib
    # в исходное состояние
    smtplib.SMTP = old_smtp
```

```
def test_send(patch_smtplib):
    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

В предыдущем коде мы использовали новый декоратор `@pytest.yield_fixture()`. Он позволяет применять синтаксис генератора, чтобы установку и разборку можно было выполнять в одной функции. Теперь наш тестовый набор можно снова запустить с исправленной версией `smtplib` следующим образом:

```
$ py.test --tb=short -v
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1 --
/Users/swistakm/.envs/book/bin/python3
cachedir: .cache
rootdir: /Users/swistakm/dev/book/chapter10/mailer, inifile:
plugins: capturelog-0.7, codecheckers-0.2, cov-2.2.1, timeout-1.0.0
collected 5 items

mailer.py::pyflakes PASSED
mailer.py::pep8 PASSED
test_mailer.py::pyflakes PASSED
test_mailer.py::pep8 PASSED
test_mailer.py::test_send FAILED

============================ FAILURES ============================
_____________________________ test_send _____________________________
test_mailer.py:29: in test_send
    'body'
mailer.py:19: in send
    server = smtplib.SMTP(server)
E   TypeError: object() takes no parameters
======= 1 failed, 4 passed, 1 pytest-warnings in 0.09 seconds =======
```

Как видно из листинга выше, наша реализация класса `FakeSMTP` еще не готова. Нам нужно обновить интерфейс, чтобы достичь соответствия исходному классу SMTP.

Согласно принципу утиной типизации, нам достаточно предоставлять интерфейсы, необходимые функции `send()`:

```
class FakeSMTP(object):
    def __init__(self, *args, **kw):
        # Аргументы для нашего примера не важны
        pass
```

```
    def quit(self):
        pass

    def sendmail(self, *args, **kw):
        return {}
```

Разумеется, поддельный класс может развиваться по мере появления новых тестов, моделируя более сложное поведение. Но в то же время класс должен быть максимально коротким и простым. Тот же принцип можно использовать с более сложными выходами, записывая и пропуская их через поддельный API. Часто это делается для сторонних серверов, таких как базы данных LDAP или SQL.

Особое внимание следует уделять таким заплаткам во встроенных и сторонних модулях. Если сделать это неправильно, то подобный подход может оставить нежелательные побочные эффекты, которые будут проявляться во всех тестах. К счастью, многие механизмы тестирования и библиотек предоставляют соответствующие утилиты, которые делают процесс создания заплаток на блоки кода более безопасным и легким. В нашем примере мы делали все вручную и использовали функцию `patch_smtplib()` с разделенными шагами установки и демонтажа. Типичное решение в `py.test` гораздо проще. Эта система поставляется с встроенным устройством `monkeypatch`, которого для наших задач должно хватить:

```
import smtplib
from mailer import send

class FakeSMTP(object):
    def __init__(self, *args, **kw):
        # Аргументы для нашего примера не важны
        pass

    def quit(self):
        pass

    def sendmail(self, *args, **kw):
        return {}

    def test_send(monkeypatch):
        monkeypatch.setattr(smtplib, 'SMTP', FakeSMTP)
        res = send(
            'john.doe@example.com',
            'john.doe@example.com',
            'topic',
            'body'
    )
    assert res == {}
```

Стоит иметь в виду, что у *подделок* есть свои ограничения. Если вы решили подделать внешнюю зависимость, то могут появиться ошибки или нежелательное поведение, которых не будет у реального сервера, и наоборот.

Использование имитаторов

Имитаторы — это подделки, которые можно задействовать для изоляции тестируемого кода. Они автоматизируют процесс сборки ввода и вывода подделки. Такие объекты более широко применяются в статически типизированных языках, где заплатки делать труднее, но полезны и в Python, поскольку позволяют сократить код, который имитирует внешние API.

Существует много библиотек имитаторов в Python, но самая известная из них — `unittest.mock`, которая есть в стандартной библиотеке. Изначально это был сторонний пакет, но вскоре он был включен в стандартную библиотеку в качестве временного пакета (docs.python.org/dev/glossary.html#term-provisional-api). В версиях Python старше 3.3 придется установить ее с PyPI таким образом:

```
$ pip install Mock
```

В нашем следующем примере использовать `unittest.mock` для исправления SMTP проще, чем создать подделку с нуля:

```
import smtplib
from unittest.mock import MagicMock
from mailer import send

def test_send(monkeypatch):
    smtp_mock = MagicMock()
    smtp_mock.sendmail.return_value = {}

    monkeypatch.setattr(
        smtplib, 'SMTP', MagicMock(return_value=smtp_mock)
    )

    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

Аргумент `return_value` имитатора или метода позволяет определить, какое значение будет возвращено после вызова. Когда используется имитатор, при каждом вызове атрибута для текущего атрибута из кода создается новый имитатор. Таким образом, исключения не выбрасываются. Это тот случай, например, для метода `quit`, который мы описали ранее, в предыдущем пункте. При наличии имитатора подобный метод будет уже не нужен.

В предыдущем примере мы фактически создали следующие два имитатора.

- Первый имитирует объект типа SMTP (класс), а не его экземпляр. Это позволяет легко создать новый объект, независимо от ожидаемого метода `__init__()`.

Имитатор по умолчанию возвращает новый объект `Mock()`, если рассматривать его как вызываемый. Поэтому нам нужно создать еще один имитатор в качестве именованного аргумента `return_value`, чтобы управлять интерфейсом экземпляра.

- Второй имитатор — экземпляр, который возвращается при вызове `smtplib.SMTP()`. В этом имитаторе мы управляем поведением метода `sendmail()`.

В предыдущих примерах мы использовали инструменты из модуля `py.test`. Однако `unittest.mock` предоставляет собственные утилиты для данной цели. В некоторых ситуациях (например, в патчах для классов) может оказаться проще и быстрее задействовать эти утилиты вместо специфичных для фреймворка инструментов. Ниже приведен пример с менеджером контекста `patch()` из модуля `unittest.mock`:

```
from unittest.mock import patch
from mailer import send

deftest_send():
    with patch('smtplib.SMTP') as mock:
        instance = mock.return_value
        instance.sendmail.return_value = {}
        res = send(
            'john.doe@example.com',
            'john.doe@example.com',
            'topic',
            'body'
        )
        assert res == {}
```

В следующем подразделе мы рассмотрим тестирование приложений в разных средах и в разных версиях зависимостей.

Совместимость среды тестирования и зависимостей

О важности изоляции среды мы говорим в этой книге постоянно. Выполняя изоляцию среды на уровне приложения (виртуализация среды) и на уровне системы (виртуализация системы), вы можете убедиться, что ваши тесты работают под воспроизводимыми условиями. Так вы обезопасите себя от редких и малоизвестных проблем, вызванных неполадками с зависимостями или трудностями совместимости систем.

Лучший способ реализовать надлежащую изоляцию тестовой среды — использовать подходящую систему непрерывной интеграции, поддерживающую виртуализацию или контейнеризацию системы. Есть хорошие системы непрерывной интеграции, такие как Travis CI (для Linux и macOS) или AppVeyor (для Windows), в которых такие возможности для проектов с открытым исходным кодом реали-

зованы бесплатно. Но если вам нужен подобный инструмент для тестирования закрытого программного обеспечения, то, вероятно, придется прибегнуть к платному сервису или разместить его в вашей собственной инфраструктуре с помощью CI с открытым исходным кодом (например, GitLab CI, Jenkins или BuildBot).

Матричное тестирование зависимостей. Матричное тестирование для проектов Python с открытым исходным кодом в большинстве случаев касается разных версий Python и реже — различных операционных систем. Отсутствие тестирования в различных системах — нормально для простых проектов на чистом Python, поскольку проблем с совместимостью систем быть не должно. Однако некоторые проекты, особенно распространяемые в виде компилируемых расширений Python, требуют тестирования в различных целевых операционных системах. Для ряда проектов с открытым исходным кодом приходится использовать несколько систем CI с целью обеспечения сборки для трех популярных ОС (Windows, Linux и macOS). Если вам нужен хороший пример, то посмотрите на небольшой проект pyrilla (github.com/swistakm/pyrilla) — простое расширение C для Python для работы с аудио. В нем используется Travis CI и AppVeyor, что позволяет создать сборки для Windows, macOS под большое количество версий CPython.

Размерности матриц тестирования не заканчиваются на разных ОС и версиях Python. Если используются пакеты, которые обеспечивают интеграцию с другими программными продуктами, например кэшами, базами данных или системными сервисами, то надо тестировать и различные версии интегрированных приложений. Существует хороший инструмент — tox (tox.readthedocs.org). Он предоставляет простой способ настройки нескольких сред тестирования и запуска всех тестов с помощью одной команды `tox`. Это очень эффективный и гибкий инструмент и вместе с тем простой в использовании. Приведем наглядный пример конфигурационного файла, который представляет собой ядро tox. Это файл `tox.ini` из проекта `django-userena` (github.com/bread-and-pepper/django-userena):

```
[tox]
downloadcache = {toxworkdir}/cache/

envlist =
    ; py26 support was dropped in django1.7
    py26-django{15,16},
    ; py27 still has the widest django support
    py27-django{15,16,17,18,19},
    ; py32, py33 support was officially introduced in django1.5
    ; py32, py33 support was dropped in django1.9
    py32-django{15,16,17,18},
    py33-django{15,16,17,18},
    ; py34 support was officially introduced in django1.7
    py34-django{17,18,19}
    ; py35 support was officially introduced in django1.8
    py35-django{18,19}
```

```
[testenv]
usedevelop = True
deps =
    django{15,16}: south
    django{15,16}: django-guardian<1.4.0
    django15: django==1.5.12
    django16: django==1.6.11
    django17: django==1.7.11
    django18: django==1.8.7
    django19: django==1.9
    coverage: django==1.9
    coverage: coverage==4.0.3
    coverage: coveralls==1.1

basepython =
    py35: python3.5
    py34: python3.4
    py33: python3.3
    py32: python3.2
    py27: python2.7
    py26: python2.6

commands={envpython} userena/runtests/runtests.py userenaumessages
{posargs}

[testenv:coverage]
basepython = python2.7
passenv = TRAVIS TRAVIS_JOB_ID TRAVIS_BRANCH
commands=
    coverage run --source=userena userena/runtests/runtests.py
userenaumessages {posargs}
    coveralls
```

Данная конфигурация позволяет тестировать `django-userena` на пяти различных версиях Django и шести версиях Python. Не каждая версия Django будет работать на всех версиях Python, и файл `tox.ini` позволяет относительно легко выявить эти случаи. На практике вся матрица состоит из 21 уникальной среды (включая специальную среду для вычисления охвата кода). Создание такого количества сред тестирования потребует огромных усилий при отсутствии инструмента, подобного tox.

Tox — очень эффективный инструмент, но его использование усложняется, если мы хотим изменить элементы среды тестирования, которые не относятся к простым зависимостям Python. В такой ситуации нам требуется выполнять тестирование под управлением различных версий системных пакетов и бэкенд-сервисов. Лучший способ решить эту проблему — снова задействовать хорошую систему непрерывной интеграции, которая позволяет легко определить матрицы переменных среды и устанавливать системное ПО на виртуальных машинах.

Хороший пример такого подхода с использованием Travis CI можно найти в проекте `ianitor` (https://github.com/ClearcodeHQ/ianitor/), и мы уже упоминали об этом в главе 11. Это простая утилита для сервиса обнаружения проекта Consul. Его сообщество очень активное, и каждый год выпускается много новых версий кода. Поэтому весьма разумно выполнять тестирование в различных версиях данного сервиса. Такой подход гарантирует, что проект `ianitor` будет максимально актуален и при этом не будет нарушена совместимость с предыдущими версиями Consul.

Ниже приведено содержание файла конфигурации `.travis.yml` для Travis CI, который позволяет тестировать три различные версии Consul и четыре версии интерпретатора Python:

```
language: python

install: pip install tox --use-mirrors
env:
  matrix:
    # consul 0.4.1
    - TOX_ENV=py27     CONSUL_VERSION=0.4.1
    - TOX_ENV=py33     CONSUL_VERSION=0.4.1
    - TOX_ENV=py34     CONSUL_VERSION=0.4.1
    - TOX_ENV=py35     CONSUL_VERSION=0.4.1

    # consul 0.5.2
    - TOX_ENV=py27     CONSUL_VERSION=0.5.2
    - TOX_ENV=py33     CONSUL_VERSION=0.5.2
    - TOX_ENV=py34     CONSUL_VERSION=0.5.2
    - TOX_ENV=py35     CONSUL_VERSION=0.5.2

    # consul 0.6.4
    - TOX_ENV=py27     CONSUL_VERSION=0.6.4
    - TOX_ENV=py33     CONSUL_VERSION=0.6.4
    - TOX_ENV=py34     CONSUL_VERSION=0.6.4
    - TOX_ENV=py35     CONSUL_VERSION=0.6.4

    # проверка охвата и стилевых правил
    - TOX_ENV=pep8     CONSUL_VERSION=0.4.1
    - TOX_ENV=coverage CONSUL_VERSION=0.4.1

before_script:
  - wget https://releases.hashicorp.com/consul/${CONSUL_VERSION}/consul_${CONSUL_
    VERSION}_linux_amd64.zip
  - unzip consul_${CONSUL_VERSION}_linux_amd64.zip
  - start-stop-daemon --start --background --exec `pwd`/consul -- agent
  -server -data-dir /tmp/consul -bootstrap-expect=1

script:
  - tox -e $TOX_ENV
```

Этот пример позволяет получить 14 уникальных тестовых сред (в том числе сборки `pep8` и `coverage`) для кода `ianitor`. В данной конфигурации используется tox для создания тестовых виртуальных окружений на виртуальных машинах Travis. Это очень популярный подход к интеграции tox с различными системами CI. Перебирая много тестовых сред с помощью tox, вы уменьшаете риск привязки к одному поставщику CI. Такие вещи, как установка новых сервисов или определение переменных среды, поддерживаются большинством конкурентов Travis CI, поэтому будет относительно легко переключиться на другого поставщика услуг, если по той или иной причине захочется выбрать другой продукт.

Разработка на основе документации

Доктесты в Python — хорошая вещь, и они есть во многих других языках программирования. Тот факт, что в документации содержатся работоспособные примеры кода, которые можно запускать как тесты, изменяет весь подход к TDD. Так, часть документации можно прогонять через доктесты во время цикла разработки. Кроме того, подобный подход гарантирует, что приведенные в документации примеры всегда будут актуальными и действительно рабочими.

Создание программного обеспечения через доктесты вместо обычных модульных тестов может быть частью процесса *разработки на основе документации* (document-driven development, DDD). Разработчики объясняют функционирование кода на простом английском языке и в то же время реализуют его.

Написание истории. Написание доктестов в подходе DDD — рассказ о том, как работает и применяется фрагмент кода. Принцип работы описывается на простом английском языке, а затем по всему тексту приводится несколько примеров использования кода. Рекомендуется сначала писать о том, как работает код, а затем вставлять его примеры.

Чтобы увидеть пример доктеста на практике, посмотрим на пакет `atomisator` (bitbucket.org/tarek/atomisator). Документация для подпакета `atomisator.parser` выглядит следующим образом:

```
=================
atomisator.parser
=================

The parser knows how to return a feed content, with
the `parse` function, available as a top-level function::

>>> from atomisator.parser import Parser

This function takes the feed url and returns an iterator
over its content. A second parameter can specify a maximum
number of entries to return. If not given, it is fixed to 10::
```

```
>>> import os
>>> res = Parser()(os.path.join(test_dir, 'sample.xml'))
>>> res
<itertools.imap ...>

Each item is a dictionary that contain the entry::
>>> entry = res.next()
>>> entry['title']
u'CSSEdit 2.0 Released'

The keys available are:

>>> keys = sorted(entry.keys())
>>> list(keys)
    ['id', 'link', 'links', 'summary', 'summary_detail', 'tags',
     'title', 'title_detail']

Dates are changed into datetime::

>>> type(entry['date'])
>>>
```

Затем доктесты развиваются, вбирая в себя новые элементы или необходимые изменения. Получается хорошая документация для разработчиков, которые хотят использовать этот пакет, и данную документацию можно разрабатывать с учетом опыта применения.

Типичная ошибка в написании тестов в документе — когда текст из-за них становится нечитабельным. Если это происходит, то такие тесты уже не следует рассматривать как часть документации.

Тем не менее некоторые разработчики, работающие исключительно через доктесты, часто группируют их в две категории: читабельные, пригодные для использования и являющиеся частью документации, и нечитабельные, применяемые для сборки и тестирования программного обеспечения.

Многие разработчики считают, что по этой причине от доктестов стоит отказаться в пользу обычных модульных тестов. Некоторые даже используют выделенные доктесты для исправления ошибок. Таким образом, баланс между доктестами и обычными тестами — дело вкуса, и решение остается за разработчиками. В современных тестовых фреймворках, таких как `nose` или `py.test`, очень легко поддерживать сборные доктесты и классические функции или тесты на основе классов одновременно.

Когда в проекте используется метод DDD, следует уделить большое внимание читабельности и решить, какие доктесты имеют право быть частью публикуемой документации.

Резюме

В этой главе мы обсудили основы методологии TDD и привели более подробную информацию о том, как эффективно писать содержательные автоматизированные тесты для программного обеспечения. Мы рассмотрели несколько способов структурирования и поиска тестов. Кроме того, поговорили о популярных инструментах тестирования, которые превращают процесс написания тестов в веселое и легкое дело.

Мы уже упоминали, что тесты можно использовать не только для проверки работоспособности приложения, но и для оценки производительности. Это хорошая отправная точка для двух следующих глав, где мы обсудим, как диагностировать проблемы производительности приложения Python, а также рассмотрим эффективные принципы и методы оптимизации.

Часть IV
Жажда скорости

Эта часть полностью посвящена скорости и ресурсам, от простых методов оптимизации, благодаря которым можно выжать максимум из процесса одной программы, до различных моделей параллелизма, позволяющих распределять и масштабировать систему на несколько ядер процессора или даже несколько компьютеров. Вы узнаете, как заставить код работать быстрее и как организовать распределенную обработку данных.

13 Оптимизация — принципы и методы профилирования

> Около 97 % времени вообще не стоит думать об эффективности: преждевременная оптимизация есть корень всех зол.
>
> *Дональд Кнут*

Эта глава посвящена оптимизации, ее основным принципам и общим методам профилирования. Мы обсудим базовые правила оптимизации, которые должен знать каждый разработчик. Кроме того, научимся выявлять узкие места в производительности приложений и использовать общие средства профилирования.

В этой главе:

- три правила оптимизации;
- стратегии оптимизации;
- поиск узких мест.

Рассмотрим три правила оптимизации.

Технические требования

Многим утилитам профилирования Python, которые описаны в этой главе, требуется пакет Graphviz. Его можно скачать по ссылке www.graphviz.org/download.

Ниже перечислены пакеты Python, упомянутые в этой главе, которые можно скачать с PyPI:

- `gprof2dot`;
- `memprof`;
- `memory_profiler`;
- `pympler`;
- `objgraph`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы примеров для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter13.

Три правила оптимизации

Оптимизация имеет цену, независимо от ее результатов. Самая важная цена, помимо очевидных временных затрат, — увеличение сложности программного обеспечения и снижение удобства сопровождения. Когда фрагмент кода работает, иногда стоит просто оставить его в покое, не пытаясь ускорить его любой ценой. Оптимизация должна быть экономически эффективной, то есть проводить ее стоит разумно. Ниже представлены три самых основных правила, которые обязательно нужно иметь в виду при выполнении какой-либо оптимизации:

- сначала — функционал;
- работа с точки зрения пользователя;
- код должен оставаться читабельным во что бы то ни стало.

В следующих подразделах мы объясним эти правила более подробно.

Сначала — функционал

Очень распространенная ошибка, которую допускают многие разработчики, — оптимизация кода с самого начала. Часто это будут бессмысленные и напрасные усилия, поскольку реальные узкие места могут оказаться в неожиданных точках программы.

Даже, казалось бы, простое приложение, как правило, состоит из множества сложных взаимодействий, и часто невозможно угадать, насколько хорошо оно будет вести себя, пока дело не дойдет до реального использования в продакшене.

Конечно, это не повод бросаться в крайности и писать все функции и алгоритмы, полностью забыв о производительности приложения. Некоторые ее *«горячие зоны»* могут быть очевидны с самого начала, и потому имеет смысл использовать эффективное решение на раннем этапе разработки. Но это нужно делать осторожно и стараться сохранить код максимально простым. Он должен работать — вот первоочередная цель, и она не должна отходить на второй план из-за фанатичной оптимизации.

Если говорить конкретно о коде, то философия Python утверждает, что у любой задачи есть один и только один способ решения. Таким образом, пока вы придерживаетесь синтаксиса Python, о котором мы подробно говорили в главах 3 и 4,

ваш код (в микромасштабе) будет оптимален. Часто лучше и быстрее написать малый объем кода, чем большой.

Пока ваш код не достиг работоспособного состояния, нужно избегать следующих методов:

- любых видов кэширования или запоминания значений, даже если это простой глобальный словарь;
- экстернализации части кода на C или гибридных языках, таких как Cython;
- использования специализированных внешних библиотек, ориентированных в основном на оптимизацию производительности.

Это не такие уж и строгие правила. Если перечисленные выше действия в конечном итоге упростят ваш код и сделают его более читабельным, то их стоит выполнить. Иногда использование библиотеки NumPy позволяет облегчить разработку отдельно взятых функций и производить более простой и быстрый код. Кроме того, не следует писать функцию с нуля при наличии хорошей библиотеки, в которой это уже сделано. Кроме того, в некоторых узкоспециализированных областях, таких как научные исследования или компьютерные игры, применение специализированных библиотек и модулей на разных языках может оказаться неизбежным с самого начала.

Так, например, в игровом движке Soya 3D на OpenGL (home.gna.org/oomadness/en/soya3d/index.html) используется C и Pyrex для быстрой реализации матричных операций при трехмерной визуализации в реальном времени.

Оптимизация проводится над программами, которые уже работают. Как говорит Кент Бек, «сначала должно работать правильно, а уже потом быстро».

В следующем подразделе мы рассмотрим то, как все функционирует с точки зрения пользователя.

Работа с точки зрения пользователя

Мы видели одну команду, тратившую много времени и усилий на оптимизацию времени запуска сервера приложений, который после запуска работал нормально. После завершения оптимизации они объявили об этом достижении своим клиентам. Представьте, сколь велико было их удивление, когда оказалось, что клиентам вообще нет до этого дела. Выяснилось, что за всеми усилиями по оптимизации не стояло никакой обратной связи от пользователей. Это была голая инициатива разработчиков. Дело в том, что они запускали сервер несколько раз в день, и время запуска для них имело значение, в отличие от клиентов.

В целом задачу-то они выполнили полезную, но, расставляя приоритеты работ по оптимизации, команда должна задавать себе следующие вопросы.

- Нас кто-нибудь просил делать это быстрее?
- Кому программа кажется медленной?
- Программа действительно медленная или работает приемлемо?
- Во сколько нам обойдется оптимизация?
- Стоит ли оно того?
- Какие именно части приложения должны быть быстрыми?

Помните, что оптимизация имеет цену, и точка зрения разработчика, как правило, бессмысленна для клиентов (если вы не пишете фреймворк или библиотеку, поскольку в этом случае клиент тоже является разработчиком).

Оптимизация не игра. К ней следует прибегать только при необходимости.

В следующем подразделе мы узнаем, как сохранить код читабельным и удобным в сопровождении.

Поддержание читабельности и удобства сопровождения

В целом шаблоны кода Python и так быстрые, однако методы оптимизации могут запутать код и сделать его трудным для чтения, понимания и развития. Приходится сохранять баланс между читабельностью/удобством сопровождения и производительностью.

Помните: оптимизация обычно не имеет границ. Всегда будет что-то, благодаря чему можно ускорить ваш код на несколько миллисекунд. Таким образом, если вы выполнили 90 % от запланированной оптимизации, а оставшиеся 10 % сделают ваш код совершенно нечитабельным и неудобным в сопровождении, то стоит прекратить работу и поискать другие решения.

Оптимизация не должна делать код нечитабельным. Если это произойдет, то нужно будет найти альтернативные решения, такие как экстернализация или полная переработка. Ищите хороший компромисс между читабельностью и скоростью.

Стратегии оптимизации мы рассмотрим в следующем разделе.

Стратегии оптимизации

Допустим, у вашей программы реальные проблемы с производительностью, которые вам необходимо решить. Не пытайтесь угадать, как сделать это быстрее. Часто трудно найти узкие места, просто посмотрев на код, и вам, вероятно, придется воспользоваться набором специализированных инструментов, чтобы найти настоящие причины проблемы.

Хорошая стратегия оптимизации начинается со следующих трех этапов.

- *Попробуйте свалить вину на другого*. Убедитесь, что проблема не в стороннем сервере или ресурсе.
- *Масштабируйте оборудование*. Убедитесь, что ресурсов достаточно.
- *Напишите тест скорости*. Создайте скрипт с учетом целей скорости.

Опишем эти стратегии в следующих подразделах.

Пробуем свалить вину на другого

Часто проблемы производительности возникают на уровне продакшена, и клиент предупреждает вас, что все работает не так, как раньше, во время тестирования. Проблемы производительности могут возникнуть из-за отсутствия тестирования приложения в реальной среде, с большим количеством пользователей и постоянно растущими объемами данных.

Но если приложения взаимодействуют с другими приложениями, то первое, что нужно сделать, — проверить наличие узких мест в этих самых взаимодействиях. Например, в случае использования сервера баз данных или какого-либо внешнего сервиса, работающего по сети, вполне возможно возникновение проблем производительности в результате неправильного применения сервисов (например, тяжелые запросы SQL) или множества последовательных соединений, которые легко распараллелить.

Физические связи между приложениями тоже следует учитывать. Сетевое соединение между сервером приложений и другим сервером в локальной сети может замедляться из-за неправильной настройки или перезагрузки.

Для понимания общей картины системы нужна хорошая и актуальная документация по архитектуре проектирования, которая содержит схемы всех взаимодействий и характер каждого звена. Эта картина имеет значение при попытке решить проблемы с производительностью, которые возникают на границах многих сетевых компонентов.

Если ваше приложение использует сторонние ресурсы, то каждое взаимодействие следует проверять для получения гарантий того, что узкое место заключается не в нем.

Масштабирование оборудования

Когда процесс требует больше физической памяти, чем имеется в данный момент, ядро системы может решить скопировать несколько страниц памяти в устройство подкачки. Когда некий процесс пытается получить доступ к такой перемещенной странице, она копируется обратно в оперативную память. Данный процесс называется подкачкой. Этот вид управления памятью часто встречается в современных операционных системах, и важно понимать его работу, поскольку в большинстве системных конфигураций по умолчанию для данной функции используются жесткие диски. А они, даже SSD-накопители, работают очень медленно по сравнению с оперативной памятью.

В целом подкачка весьма неплоха. В некоторых системах она используется для малодоступных страниц памяти, даже при большом наличии свободной памяти, просто для экономии ресурсов. Но если давление на память очень высоко и все процессы начинают использовать больше памяти, чем имеется, то производительность резко падает. В таких ситуациях система начинает перекачивать одни и те же страницы памяти туда-сюда, затрачивая на это много времени. С точки зрения пользователя, система на данном этапе считается мертвой. Таким образом, если ваше приложение тратит много памяти, то чрезвычайно важно масштабировать оборудование для предотвращения подобных ситуаций.

Важно иметь достаточный объем памяти в системе, но также важно убедиться, что приложения не потребляют ее в чрезмерных объемах (привет, Google). Например, если программа работает с большими видеофайлами, которые могут весить несколько сотен мегабайт, то их нужно не загружать в память полностью, а разбивать на фрагменты.

Обратите внимание: масштабирование аппаратного обеспечения (вертикальное масштабирование) имеет некоторые очевидные ограничения. Вы не можете поместить в серверную стойку бесконечное количество оборудования. Кроме того, производительное аппаратное обеспечение чрезвычайно дорого (закон убывающей доходности), то есть деньги — тоже ограничение. С этой точки зрения всегда лучше иметь систему, которую можно масштабировать, добавляя новые узлы или станции (горизонтальное масштабирование). Это позволяет масштабировать сервис программным обеспечением, имеющим лучшее соотношение цены/производительности.

К сожалению, разработка и сопровождение масштабируемых распределенных систем — сложные и дорогие процессы. Если ваша система плохо масштабируется по горизонтали или попросту быстрее и дешевле использовать вертикальное масштабирование, то может быть лучше прибегнуть к вертикальному, а не тратить время и ресурсы на рефакторинг архитектуры системы. Помните, что аппаратное обеспечение со временем неизменно дешевеет и ускоряется. Многие продукты долго находятся в точке, когда потребности масштабирования совпадают со скоростью роста производительности аппаратных средств без увеличения стоимости.

Написание теста скорости

Начиная оптимизацию, важно соблюдать рабочий процесс, похожий на разработку на основе тестирования, а не заниматься проверкой вручную. Рекомендуется выделить в приложении тестовый модуль с функциями для тестирования, которые бы тестировали код, требующий оптимизации. Такой подход поможет отслеживать прогресс в оптимизации приложения.

Можно даже добавить специальные операторы, если необходимо достичь конкретных показателей скорости. Чтобы избежать ухудшения скорости, можно не убирать тесты после оптимизации. Разумеется, измерение времени выполнения зависит от мощности используемого процессора, поэтому крайне трудно получить объективное и повторяемое измерение при любых условиях. Как следствие, тесты скорости выполняются лучше, если запускаются в тщательно подготовленной и изолированной среде. Важно также убедиться, что единовременно выполняется только один такой тест. Кроме того, лучше сосредоточиться на наблюдении трендов в изменении производительности, а не каких-то конкретных временных условий. К счастью, во многих популярных фреймворках для тестирования, таких как `pytest` и `nose`, есть плагины, которые могут автоматически измерять время выполнения теста и даже сравнивать результаты нескольких прогонов.

В следующем разделе рассмотрим механизм поиска узких мест.

Поиск узких мест

Такой поиск обычно производится следующим образом:

- профилированием использования процессора;
- профилированием применения памяти;
- профилированием использования сети;
- трассировкой.

Под *профилированием* понимается наблюдение за поведением кода или конкретными показателями производительности в рамках одного процесса или потока выполнения, работающих на одном хосте, и это обычно делает сам процесс. Добавление в приложение кода, который позволяет записывать и измерять различные показатели производительности, называется *инструментацией*. *Трассировка* — это обобщение профилирования, позволяющее наблюдать и измерять сетевые процессы, запущенные на нескольких узлах.

Подробнее о профилировании использования процессора — в следующем подразделе.

Профилирование использования ЦП

Первый источник узких мест — ваш собственный код. В стандартной библиотеке есть все инструменты, необходимые для профилирования кода. Они основаны на детерминированном подходе.

Детерминированный профайлер измеряет время, которое код находится в каждой функции, при этом подсчет ведется на самом низком уровне. Сам подсчет — тоже затраты, но зато позволяет понять, какая из функций тратит время. *Статистический профайлер*, с другой стороны, ориентируется на указатель инструкции, не инструментируя сам код. Последний подход менее точен, но позволяет запустить целевую программу на полной скорости.

Существует два способа профилирования кода:

- *макропрофилирование* — профилирует всю программу и генерирует статистику;
- *микропрофилирование* — анализирует отдельно взятую часть программы вручную.

В следующих пунктах текста обсудим эти способы.

Макропрофилирование

Макропрофилирование реализуется путем запуска приложения в специальном режиме, когда интерпретатор собирает статистику использования кода. Python предоставляет несколько инструментов для этой задачи:

- `profile` — реализация на чистом Python;
- `cProfile` — реализация на C с тем же интерфейсом, что и у `profile`, но с меньшими затратами.

Рекомендуемым выбором для большинства программистов Python является `cProfile`, поскольку затраты в нем меньше. Однако если вам необходимо каким-то

образом расширить профайлер, то `profile` подойдет лучше, ввиду того что в нем не используются расширения C.

Оба инструмента имеют одинаковые интерфейс и принципы использования, поэтому мы будем применять только один из них. Ниже показан модуль `myapp.py` с основной функцией, которую мы станем профилировать с помощью модуля `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 вызывается непосредственно из командной строки, а результаты приведены ниже:

```
$ python3 -m cProfile myapp.py
         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}
```

Расшифровка значений столбцов:

- `ncalls` — общее количество вызовов;
- `tottime` — общее время, проведенное в функции, за исключением времени, проведенного в вызовах подфункций;

- `cumtime` — общее время, проведенное в функции, включая время, затраченное на вызовы вспомогательных функций.

Столбец `percall` слева от `ttime` содержит значения `tottime/ncalls`, а справа от `cumtime` — `cumtime/ncalls`.

Это вывод на печать статического объекта, который создается профайлером. Вы также можете создавать и просматривать данный объект в интерактивной сессии Python следующим образом:

```
>>> 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}
```

Статистические данные также можно сохранить в файл для будущего считывания модулем `pstats`. В этом модуле есть класс, который знает, как обрабатывать файлы профайлера, и справка, позволяющая легко анализировать результаты профилирования. В листинге ниже показано, как получить информацию об общем количестве вызовов и отображать первые три из них, упорядоченные 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)
```

Здесь вы можете просматривать код, выводя результаты вызовов 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 сортировки вывода позволяет работать с данными и эффективно искать узкие места. Например, рассмотрим следующие сценарии.

- Когда количество мелких вызовов (нижнее значение `percall` для столбца `tottime`) очень высоко (верхнее значение `ncalls`) и занимает большую часть времени работы, то функция или метод, вероятно, попадает в длинный цикл. В качестве оптимизации можно переместить этот вызов в другую область, чтобы уменьшить количество операций.
- Когда вызов одной функции занимает очень много времени, возможно, стоит использовать кэш.

Еще один отличный способ визуализировать узкие места из данных профилирования — превратить их в схемы (рис. 13.1). Модуль *gprof2dot* (github.com/jrfonseca/gprof2dot) помогает превратить данные профайлера в точечный график. Вы можете скачать этот простой скрипт PyPI с помощью `pip` и использовать его в файле статистики, который был создан с помощью модуля cProfile (вам также потребуется ПО Graphviz с открытым исходным кодом, www.graphviz.org). Ниже приведен пример вызова `gprof2dot.py` для Linux:

```
$ gprof2dot.py -f pstats myapp.stats | dot -Tpng -o output.png
```

Преимущество gprof2dot заключается в том, что он мало зависит от языка. Он не ограничивается `profile` или `cProfile` и может считывать данные из нескольких других профилей, таких как Linux perf, Xperf, gprof, Java HPROF и др.

На схеме, сгенерированной с помощью gprof2dot, показаны различные пути кода, которые были выполнены программой, а также время, затраченное на каждом пути. Модуль отлично подходит для изучения моделей производительности

крупных приложений. Макропрофилирование — хороший способ найти функцию, в которой есть проблема, или по крайней мере очертания этой проблемы. Когда эта функция найдена, можно переходить к микропрофилированию.

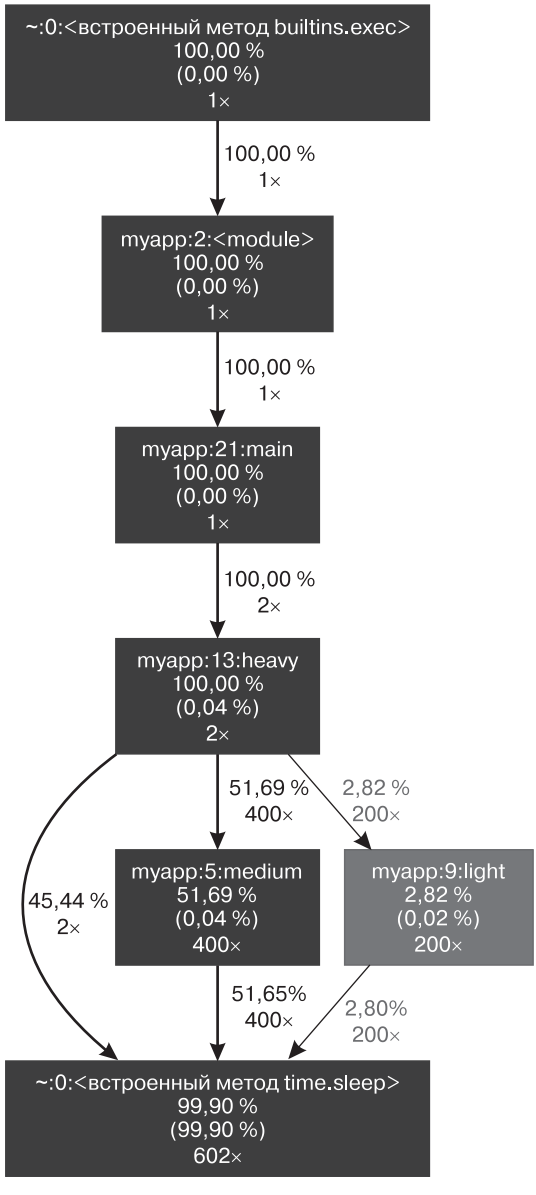


Рис. 13.1. Пример схемы профилирования, построенной gprof2dot

Микропрофилирование

Когда медленная функция найдена, иногда бывает необходимо перейти на более тонкий уровень и проанализировать часть программы. Это делается через ручное инструментирование части кода в тесте скорости.

Например, модуль `cProfile` можно использовать в виде декоратора, как показано в следующем примере:

```
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)

@profile('time')
def main():
    for i in range(2):
        heavy()

if __name__ == '__main__':
    main()
```

Данный подход позволяет выполнять тестирование выбранных частей приложения и делает вывод статистики более конкретным. Таким образом, вы можете собрать много точных и изолированных профилей, запустив приложение всего раз:

```
$ python3 cprofile_decorator.py
===== 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)

===== main() profile =====
Wed Apr 10 03:11:57 2019
/var/folders/jy/wy13kx0s7sb1dx2rfsqdvzdw0000gq/T/tmp6c0y2oxj

         62 function calls in 8.663 seconds

   Ordered by: internal time
   List reduced from 27 to 3 due to restriction <3>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1 8.662 8.662 8.662 8.662 {method 'enable' of '_lsprof.Profiler' objects}
        1 0.000 0.000 0.000 0.000 {built-in method posix.lstat}
        8 0.000 0.000 0.000 0.000
/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/l
ib/python3.7/random.py:224(_randbelow)
```

На данном этапе сам список вызываемых методов нам уже неинтересен, поскольку медленная функция уже найдена. Теперь нас интересует ее скорость.

Модуль `timeit` весьма полезен. Он позволяет измерять время выполнения небольшого фрагмента кода с помощью лучшего таймера в хостовой системе (`time.time` или `time.clock`), как показано в следующем примере:

```
>>> from myapp import light
>>> import timeit
>>> t = timeit.Timer('main()')
>>> t.timeit(number=5)
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
5.6196951866149902
```

Данный модуль позволяет повторить вызов несколько раз и может быть легко использован, чтобы протестировать выделенные фрагментов кода. Это очень полезно вне контекста приложения, но не слишком удобно внутри него.

Детерминированный профайлер выдает результаты в зависимости от того, чем в данный момент занят компьютер, и поэтому результаты могут отличаться каждый раз. Если выполнить тест несколько раз и взять среднее значение, то вы получите более точные результаты. Кроме того, ряд компьютеров имеет специальные функции центрального процессора, такие как SpeedStep, которые могут повлиять на результаты в случае простоя компьютера во время теста. Таким образом, для небольших фрагментов кода лучше выполнять тест несколько раз. Кэши тоже играют роль, например кэш DNS или процессора.

Результаты модуля `timeit` следует использовать с осторожностью. Это очень хороший инструмент объективного сравнения двух коротких фрагмента кода. Однако он же позволяет легко совершить опасные ошибки, которые приведут вас к неправильным выводам. Ниже данный модуль был применен для сравнения двух простых фрагментов кода, исходя из которого можно подумать, что конкатенация строк работает быстрее, чем `str.join()`:

```
$ python3 -m timeit -s 'a = map(str, range(1000))' '"".join(a)'
1000000 loops, best of 3: 0.497 usec per loop

$ python3 -m timeit -s 'a = map(str, range(1000)); s=""' 'for i in a: s +=
i'
10000000 loops, best of 3: 0.0808 usec per loop
```

Из главы 3 мы уже знаем, что конкатенация — это не очень хорошо. Несмотря на некоторые незначительные микрооптимизации CPython, которые были разработаны именно для таких случаев, время выполнения все равно выходит квадратичным. Проблема заключается в нюансах настройки аргумента `timeit()` (или параметра `-s` в командной строке) и в том, как в Python 3 работает тип `range`. Подробности оставим вам в качестве зарядки для мозга. Правильный способ сравнения конкатенации с методом `str.join()` в Python 3 таков:

```
$ python3 -m timeit -s 'a = [str(i) for i in range(10000)]' 's="".join(a)'
10000 loops, best of 3: 128 usec per loop
$ python3 -m timeit -s 'a = [str(i) for i in range(10000)]' '
  s = ""
  for i in a:
      s += i
  '
1000 loops, best of 3: 1.38 msec per loop
```

В следующем подразделе обсудим профилирование применения памяти.

Профилирование использования памяти

Еще одна проблема, с которой вы можете столкнуться при оптимизации приложения, — это потребление памяти. Если программа начинает затрачивать так много памяти, что система постоянно активирует подкачку, то, вероятно, в приложении есть место, в котором создается слишком много объектов, либо плохо работает удаление ненужных объектов. Такое нерациональное расходование ресурсов сложно обнаружить с помощью обычных методов профилирования процессора. Иногда потребление памяти, приводящее к использованию подкачки, также связано с большой нагрузкой процессора, которую легко обнаружить, прибегнув к обычным методам профилирования. Но падение производительности, как правило, возникает внезапно, в неожиданный момент и не связано с ошибками программиста. Часто возникают утечки памяти, и память потребляется непрерывно в течение длительного времени.

Посмотрим на то, как Python работает с памятью.

Как Python работает с памятью

Применение памяти, вероятно, самая трудная для профилирования вещь в Python при использовании реализации CPython. Такие языки, как C, позволяют узнать объем занимаемой любым элементом памяти, а в Python подобной возможности нет. Это связано с динамической природой языка и с тем фактом, что управление памятью напрямую пользователю недоступно.

Кое-какие сырые механизмы управления памятью мы уже рассматривали в главе 9. Мы уже знаем, что в CPython для управления распределением объектов используется подсчет ссылок. Это детерминированный алгоритм, который гарантирует, что будет срабатывать открепление объекта, когда счетчик ссылок на объект становится равен нулю. Несмотря на детерминированность, данный процесс сложно отслеживать и анализировать (особенно в сложных кодовых базах). Кроме того, открепление объектов на уровне счетчика ссылок не обязательно означает, что память действительно освободится. В зависимости от флагов компиляции интерпретатора CPython, системного окружения или контекста выполнения

внутренний менеджер памяти может оставить некоторые блоки памяти «про запас» и не освобождать память полностью.

Дополнительные микрооптимизации в реализации CPython еще сильнее усложняют предсказание фактического использования памяти. Например, две переменные, которые указывают на одну и ту же короткую строку или малое целое значение, могут указывать или не указывать на один и тот же экземпляр объекта в памяти (этот механизм называется интернированием).

Несмотря на кажущуюся сложность, управление памятью в Python очень хорошо прописано в официальной документации Python (docs.python.org/3/c-api/memory.html). Обратите внимание: микрооптимизациями памяти наподобие интернирования в большинстве случаев при отладке проблем с памятью можно пренебречь. Кроме того, подсчет ссылок основан на простом принципе: если на объект нет ссылок, то он удаляется. Таким образом, в контексте выполнения функции каждый локальный объект в конечном счете удаляется, если интерпретатор:

- выходит из функции;
- узнает, что объект больше не используется.

Таким образом, следующие объекты дольше остаются в памяти:

- глобальные объекты;
- объекты, на которые в том или ином роде есть ссылки.

Подсчет ссылок в Python удобен и освобождает вас от необходимости вручную отслеживать ссылки на объекты, и поэтому вам не придется уничтожать их вручную. Однако, поскольку разработчики не думают об уничтожении объектов, использование памяти может бесконтрольно расти, если не обращать внимания на то, как применяются структуры данных.

Ниже приведены типичные пожиратели памяти.

- Бесконтрольно растущий кэш.
- «Производители» объектов, которые регистрируют глобальные экземпляры и не отслеживают их использование, например интерфейс базы данных, который применяется всякий раз при запросе к базе данных.
- Незавершенные ветви кода.
- Объекты с методом `__del__` и участвующие в циклах тоже пожирают много памяти. В более старых версиях Python (до версии 3.4) сборщик мусора не прерывает ссылочный цикл, поскольку не уверен в том, какой именно объект нужно удалять первым. Возникает утечка памяти. Использование этого метода в большинстве случаев плохая идея.

К сожалению, при написании расширений C с помощью Python/C API управление подсчетом ссылок выполняется вручную с помощью макросов `Py_INCREF()` и `Py_DECREF()`. Мы обсудили особенности работы с подсчетом ссылок и владение ими в главе 9, следовательно, вы уже должны знать, что данная тема сложна и полна ловушек. Именно поэтому большинство проблем памяти связано с плохо написанными расширениями C.

Профилирование памяти описывается ниже.

Профилирование памяти

Прежде чем мы начнем искать проблемы памяти в Python, вы должны знать, что природа утечки памяти в Python довольно необычна. В ряде компилируемых языках, таких как C и C++, утечка памяти почти всегда вызвана выделенными блоками памяти, на которые ничто больше не ссылается. Не имея ссылки на память, вы не можете ее освободить, и именно эта ситуация называется *утечкой памяти*. В Python у пользователя нет доступа к управлению памятью низкого уровня, поэтому возникает другая проблема — утечка ссылок, то есть ссылки на объекты, которые больше не нужны, но не были удалены. Интерпретатор перестает высвобождать ресурсы, но это не то же самое, что утечки памяти в C. Конечно, всегда существует исключительный случай расширений C, но это совершенно другая история, для которой требуются совершенно другие инструменты для диагностики, и из кода Python с проблемой утечек памяти работать трудно.

Итак, проблемы памяти в Python в основном вызваны неожиданными или незапланированными шаблонами использования ресурсов. Довольно редко бывает, что неправильное распределение памяти и открепление подпрограмм вызвано реальными ошибками. Подобные процедуры доступны разработчику только в CPython при написании расширений C с Python/C API, и вероятность столкнуться с этим мала. Подытожим: большинство так называемых утечек памяти в Python в основном вызвано раздутой сложностью программы и тонкими взаимодействиями между ее компонентами, которые очень трудно отследить. Чтобы найти такие недостатки в программе, нужно представлять, как выглядит в ней использование памяти.

Получение информации о том, сколько объектов находится под управлением интерпретатора, и вычисление реального размера — непростая задача. Чтобы узнать, сколько памяти данный объект занимает в байтах, придется просмотреть все его атрибуты, работать с перекрестными ссылками, а затем суммировать все. Это довольно сложная задача, если учесть, как объекты часто связаны между собой. Встроенный модуль `gc`, который является интерфейсом сборщика мусора в Python, не имеет высокоуровневых функций для этой задачи, и получение информации потребует компиляции в режиме отладки.

Часто программисты просто запрашивают у системы данные об использовании памяти приложением до и после выполнения какой-то конкретной операции. Но эта мера является приближенной и во многом зависит от управления памятью на уровне системы. Так, применение команды `top` в Linux или диспетчера задач в Windows позволяет обнаружить проблемы с памятью, если они очевидны. Но данный подход трудоемок и усложняет поиск неисправного блока кода.

К счастью, существует несколько инструментов, позволяющих делать снимки памяти, а также рассчитать количество и размер загружаемых объектов. Однако не будем забывать, что в Python затруднено освобождение памяти, поскольку он любит сохранять объекты на случай «а вдруг пригодятся».

Когда-то одним из самых популярных инструментов для исправления проблем с памятью в Python был Guppy-PE и компонент Heapy. К сожалению, он не поддерживается в Python 3. Но зато есть альтернативные варианты, в некоторой степени совместимые с Python 3:

- *Memprof* (jmdana.github.io/memprof/) — работает на Python 2.6, 2.7, 3.1, 3.2 и 3.3, а также в некоторых POSIX-совместимых системах (macOS и Linux). Последнее обновление вышло в декабре 2016 года;
- *memory_profiler* (pypi.python.org/pypi/memory_profiler) — работает с теми же версиями Python, что и Memprof, но репозиторий кода тестируется на Python 3.6. Активно поддерживается;
- *Pympler* (pythonhosted.org/Pympler/) — работает с Python 2.7, со всеми версиями Python 3, от 3.3 до 3.7, и не зависит от ОС. Активно поддерживается;
- *objgraph* (mg.pov.lt/objgraph/) — работает с Python 2.7, 3.4, 3.5, 3.6 и 3.7 и не зависит от ОС. Активно поддерживается.

Обратите внимание: эта информация о совместимости основана исключительно на поисковых классификаторах, которые используются последними версиями пакетов, а также на документации и проверке определений сборки. Пока вы читаете эту книгу, все могло измениться.

Как видите, разработчикам на Python доступно много инструментов профилирования памяти. Каждый из них имеет ограничения. В этой главе мы остановимся только на проекте, который хорошо работает с последней версией Python (то есть Python 3.7) в разных операционных системах. Это инструмент *objgraph*. Его API кажется неуклюжим и функционально ограниченным, однако он работает, выполняет свои задачи и при этом очень прост в использовании. Работа с памятью — временный процесс и не добавляется в код навсегда, поэтому даже такого инструмента будет достаточно. Его поддержка множества версий Python и независимость от ОС — причина того, что мы остановимся именно на `objgraph`.

Другие инструменты, упомянутые здесь, тоже хороши, но ими вы можете заняться самостоятельно.

А теперь рассмотрим модуль `objgraph`.

Модуль objgraph. Это простой модуль для создания схем ссылок на объекты, полезный во время охоты на утечки памяти в Python. Он доступен на PyPI, но к нему в дополнение требуется модуль Graphviz для создания схем использования памяти. В таких системах, как macOS или Linux, его легко скачать, задействуя менеджер пакетов (например, `brew` для macOS, `apt-get` для Debian/Ubuntu). В Windows вам будет необходимо скачать программу установки Graphviz со страницы проекта (www.graphviz.org) и установить его вручную.

Модуль `objgraph` предоставляет несколько утилит, которые позволяют объединять в список и выводить статистические данные о применении памяти и подсчете объектов. Пример использования такой утилиты показан в сессии интерпретатора ниже:

```
>>> 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: номера выделенных объектов отображаются objgraph уже высокими из-за того, что многие встроенные функции и типы Python — это обычные объекты Python, живущие в одной и той же памяти процесса. Вдобавок сам objgraph создает некоторые объекты, включенные в данный вывод.

Как мы уже упоминали ранее, `objgraph` позволяет создавать схемы моделей использования памяти и перекрестных ссылок, связывающие все объекты в заданном пространстве имен. Самые полезные утилиты этой библиотеки — `objgraph.show_refs()` и `objgraph.show_backrefs()`. Они берут на вход ссылки на проверяемый объект и сохраняют схемы в файл, задействуя пакет Graphviz. Примеры таких графиков показаны на рис. 13.2 и 13.3. Ниже приведен код, который применялся для создания этих схем:

```
from collections import Counter
import objgraph

def graph_references(*objects):
    objgraph.show_refs(
        objects,
        filename='show_refs.png',
        refcounts=True,
        # Дополнительная фильтрация для 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 = """
    Люди которые считают, что знают все, очень раздражают тех,
    кто действительно что-то знает.
    """
    words = quote.lower().strip().split()
    counts = Counter(words)
    graph_references(words, quote, counts)
```

На рис. 13.2 показана схема всех ссылок на объекты `words`, `quote` и `counts`.

На следующей схеме (рис. 13.3) показаны только объекты, ссылающиеся на объекты, которые мы передавали в функцию `show_backrefs()`. Они называются *обратными ссылками* и очень полезны в поиске объектов, которые не дают высвободить другие объекты.

Базовая установка пакета objgraph не включает модуль Graphviz, который требуется для создания схем в виде растрового изображения. Без Graphviz схемы будут выводиться в специальном формате. Вообще, Graphviz — очень популярная программа, часто встречающаяся в репозиториях пакетов операционных систем. Вы также можете скачать ее, пройдя по ссылке www.graphviz.org.

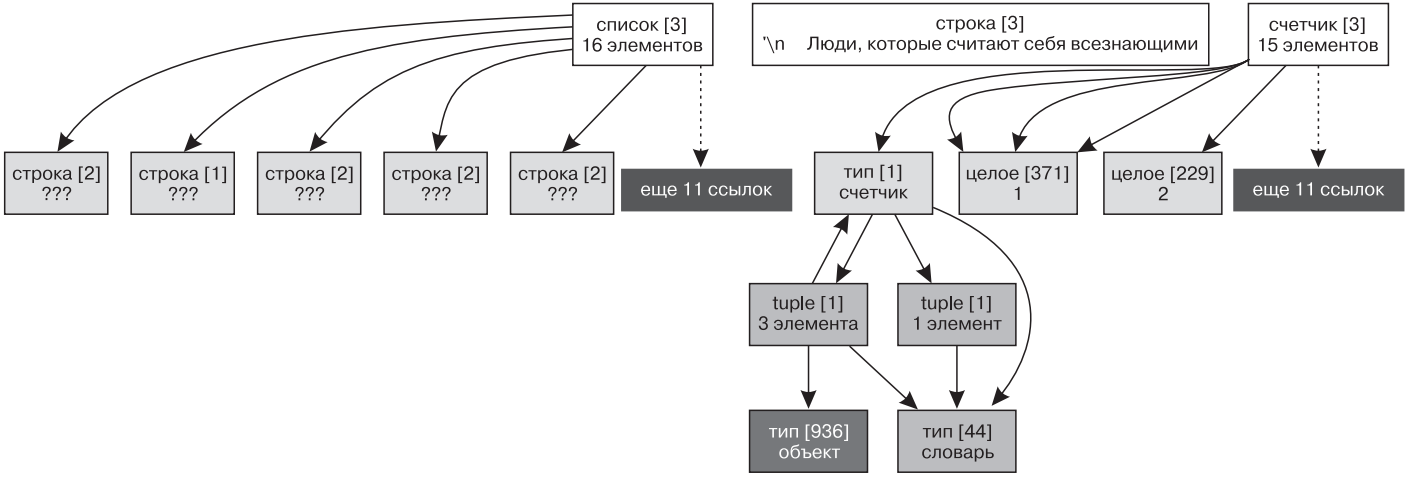


Рис. 13.2. Пример схемы show_refs(), построенной из функции graph_references()

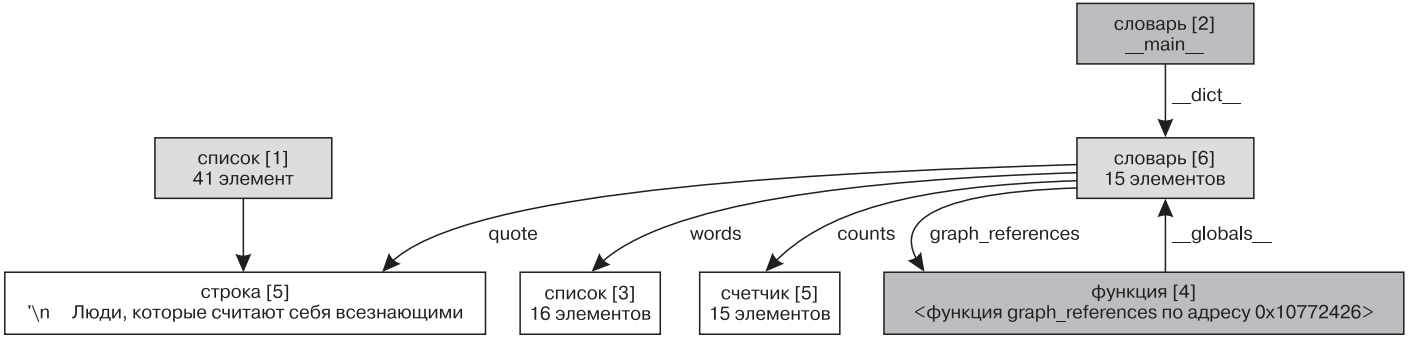


Рис. 13.3. Пример схемы show_backrefs(), построенной из функции graph_references()

Чтобы показать применение `objgraph` на практике, рассмотрим пример кода, который в определенных версиях Python может создать проблемы с памятью. Как мы уже отмечали несколько раз в этой книге, у CPython есть свой собственный сборщик мусора, существующий независимо от его механизма подсчета ссылок. Он не применяется для управления памятью общего назначения, и его единственная цель состоит в решении проблем циклических ссылок. Часто объекты могут ссылаться друг на друга таким образом, что их становится невозможно удалить, используя простые методы, основанные на отслеживании количества ссылок. Вот простейший пример:

```
x = []
y = [x]
x.append(y)
```

Визуально данная ситуация представлена на следующей схеме (рис. 13.4). Даже если все внешние ссылки на объекты `x` и `y` будут удалены (например, путем возврата из локальной области видимости функции), эти два объекта не будут удалены через подсчет ссылок, поскольку ссылаются друг на друга. И здесь начинает работать сборщик мусора Python. Он находит циклические ссылки на объекты и открепляет их, если нет никаких внешних ссылок на эти объекты за пределами цикла.

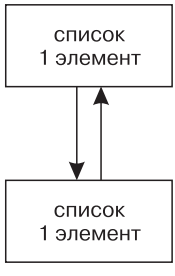


Рис. 13.4. Пример циклической ссылки между двумя объектами

Проблемы начинаются, когда у одного из объектов такого цикла есть пользовательский метод `__del__`. Это пользовательский обработчик открепления, который будет вызываться в момент обнуления счетчика ссылок объекта. Он может выполнять произвольный код на Python и таким образом создавать новые ссылки на объекты. Именно поэтому сборщик мусора в версии Python 3.4 не может «ломать» циклические ссылки, если по крайней мере у одного из объектов есть пользовательская реализация метода `__del__`. В PEP 442 введено безопасное удаление объектов в Python, ставшее частью стандарта языка, начиная с Python 3.4. Однако проблема сохраняется для пакетов, имеющих проблемы совместимости и нацеленных на широкий спектр версий интерпретатора Python. Следующий фрагмент кода позволяет показать, чем отличается поведение циклического сборщика мусора в разных версиях Python:

```
import gc
import platform
import objgraph

class WithDel(list):
    """Наследник класса list со своей имплементацией метода __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("unreachable prior collection: %s" % gc.collect())
    print("unreachable after collection: %s" % len(gc.garbage))
    print("WithDel objects count:        %s" %
          objgraph.count('WithDel'))

if __name__ == "__main__":
    print("Python version: %s" % platform.python_version())
    print()
    main()
```

Приведенный ниже вывод этого кода, выполненный на Python 3.3, показывает, что циклический сборщик мусора в старых версиях Python не удаляет объекты с методом `__del__()`:

```
$ python3.3 with_del.py
Python version: 3.3.5
unreachable prior collection: 3
unreachable after collection: 1
WithDel objects count:        1
```

В новых версиях Python сборщик мусора может безопасно обрабатывать завершение объектов, даже если для них определен метод `__del__()`:

```
$ python3.5 with_del.py
Python version: 3.5.1

unreachable prior collection: 3
unreachable after collection: 0
WithDel objects count:        0
```

В последних версиях Python пользовательская очистка памяти больше не является проблемой, но сложность сохраняется в приложениях, которые должны работать в различных средах. Как мы уже упоминали ранее, функции `objgraph.show_refs()` и `objgraph.show_backrefs()` позволяют легко определить проблемные объекты, которые участвуют в неразрывных циклических ссылках. Например, мы можем легко изменить функцию `main()`, чтобы она выводила все обратные ссылки на экземпляры `WithDel` с целью найти утечки ресурсов:

```
def main():
    x = WithDel()
    y = []
    z = []
```

```
x.append(y)
y.append(z)
z.append(x)

del x, y, z

print("unreachable prior collection: %s" % gc.collect())
print("unreachable after collection: %s" % len(gc.garbage))
print("WithDel objects count:        %s" %
    objgraph.count('WithDel'))

objgraph.show_backrefs(
    objgraph.by_type('WithDel'),
    filename='after-gc.png'
)
```

Запуск предыдущего примера под Python 3.3 даст схему, на которой видно, что `gc.collect()` не может удалить экземпляры объектов `x`, `y` и `z`.

Кроме того, `objgraph` выделяет все объекты, у которых определен метод `__del__()`, выделенный красным (рис. 13.5), чтобы облегчить обнаружение таких проблем.

Далее обсудим утечки памяти в коде C.

Утечки памяти в коде, написанном на C

Если код Python выглядит правильно, но потребление памяти в изолированной функции все равно увеличивается, утечка может быть на стороне C. Это происходит, например, когда в критической части некоего импортируемого расширения C отсутствует макрос `Py_DECREF`.

Код на C интерпретатора CPython довольно надежен и проверен на наличие утечек памяти, поэтому здесь искать проблемы с памятью не стоит. Но если вы используете пакеты, имеющие собственные расширения C, то, возможно, стоит исследовать именно их. Поскольку вам придется работать с кодом, работающим на гораздо более низком уровне абстракции, чем Python, придется задействовать различные инструменты для решения таких проблем с памятью.

Отладка памяти на C — непростая задача, поэтому прежде, чем погрузиться во внутреннее устройство расширений, убедитесь, что вы правильно диагностировали источник проблемы. Популярный метод — изолировать подозрительный пакет с кодом, имитирующим тесты. Чтобы диагностировать источник вашей проблемы, вы должны выполнить следующие действия:

- написать отдельный тест для каждого блока API или функционала расширения, которое подозреваете в утечках памяти;
- выполнить тест в цикле в длительной изоляции (один тест за прогон);
- понаблюдать, какие из функций потребляют все больше и больше памяти.

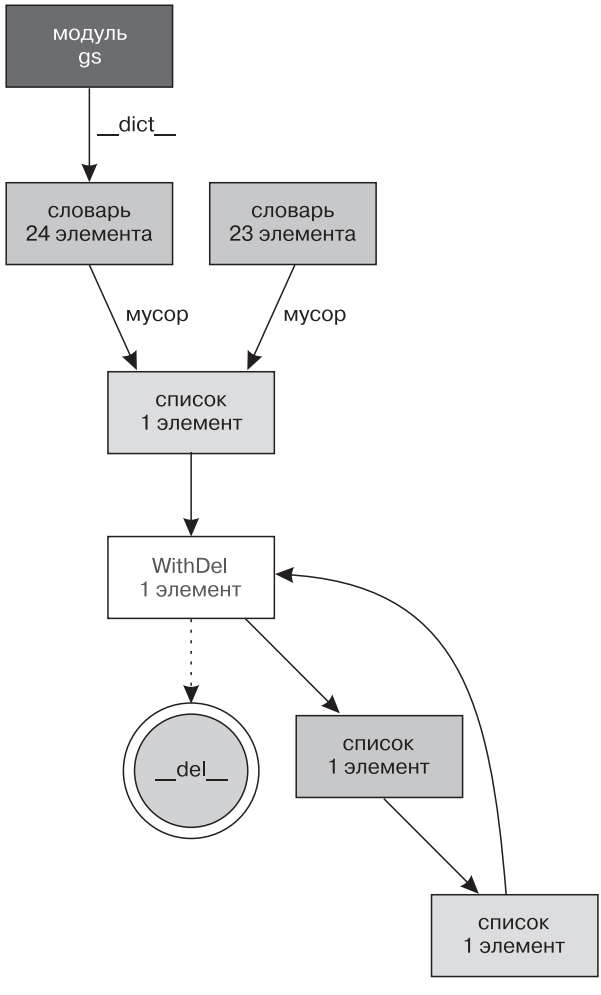


Рис. 13.5. Пример циклических ссылок, которые не удаляются сборщиком мусора Python до версии 3.4

С помощью такого подхода вы в конечном итоге найдете неисправную часть расширения, и это позволит сократить затраты времени на поиск и исправление проблемы. Данный процесс может показаться обременительным, поскольку требует много дополнительного времени и написания кода, но в долгосрочной перспективе действительно окупается. Вы всегда можете облегчить вашу работу за счет многократного использования некоторых инструментов тестирования, представленных в главе 12. Такие утилиты, как `pytest` и `tox`, возможно, не вполне подходят для

этого случая, но все-таки могут сократить время, необходимое для запуска тестов в изолированных средах.

Если вы успешно нашли часть расширения, которая тратит память, то можно приступить к самой отладке. При удачном положении дел простой и банальный просмотр кода даст желаемые результаты. Часто проблема решается добавлением вызова `Py_DECREF`. Но в большинстве случаев такой простоты ждать не приходится, и нужно будет нечто более эффективное. Один из наиболее известных инструментов для поиска утечек памяти в скомпилированном коде, который должен быть в арсенале каждого программиста, — Valgrind. Это полноценный фреймворк для создания инструментов динамического анализа, ввиду чего его трудно освоить, но вам стоит обязательно ознакомиться с основами его применения.

Профилирование использования сетевых ресурсов будет рассмотрено в следующем разделе.

Профилирование использования сети

Как мы уже говорили ранее, приложение, обменивающееся данными с программами сторонних производителей, например с базами данных, кэшами, веб-сервисами или серверами аутентификации, замедляет свою работу. Это можно отследить с помощью обычного профилирования кода на стороне приложения. Но если ПО сторонних производителей само по себе работает отлично, то проблема может крыться в сети.

Источником проблемы может быть неправильно настроенное сетевое оборудование, сетевое звено с низкой пропускной способностью или большое количество столкновений трафика, что заставляет компьютеры отправлять одни и те же пакеты по нескольку раз.

Вот несколько рекомендаций, которые вам помогут. Чтобы понять происходящее, сделайте следующее:

- отсмотрите сетевой трафик, используя такие инструменты, как:
  - ntop (только Linux, www.ntop.org);
  - Wireshark (www.wireshark.org);
- отследите нездоровые или неправильно сконфигурированные устройства с помощью средств мониторинга на основе широко используемого протокола SNMP (www.net-snmp.org);
- оцените пропускную способность между двумя компьютерами с помощью статистического инструмента, такого как Pathrate (www.cc.gatech.edu/~dovrolis/bw-est/pathrate.html).

Если хотите еще больше углубиться в проблемы производительности сети, то почитайте книгу Ричарда Блюма *Network Performance Toolkit Open Source*. Там опи-

саны стратегии для настройки приложений, которые используют много трафика, а также приведено руководство по поиску сложных проблем в сети.

Есть еще одна хорошая книга, *High Performance MySQL* Джереми Зоодни, в которой также говорится о написании приложений, использующих MySQL.

А теперь кратко поговорим о трассировке сетевых транзакций.

Трассировка сетевых транзакций. В настоящее время появление архитектуры микросервисов и современных систем контейнерной оркестровки значительно облегчило строительство больших распределенных систем. Довольно часто распределенные приложения работают с задержками не из-за медленной сети, а потому, что компоненты приложения слишком много общаются между собой. Сложные распределенные системы могут иметь десятки или даже сотни взаимодействий и микросервисов. Очень часто эти сервисы реплицируются на многих вычислительных узлах с различными аппаратными характеристиками. Они часто общаются с несколькими сервисами через множество промежуточных и программных слоев, таких как кэширующие прокси-серверы и серверы аутентификации. Часто простое взаимодействие с пользователем, например запрос HTTP API или загрузка веб-страницы, может запускать многоуровневую связь между несколькими серверами.

В таких сильно распределенных системах труднее всего определить тот самый сервис, создающий узкое место в производительности. Классические инструменты для профилирования кода обычно работают в изолированных средах и мониторят поведение одного процесса. Некоторые программы для мониторинга могут выполнять недетерминированное профилирование на коде в продакшене, но это полезно для общего статистического анализа производительности, а конкретные проблемы тут обнаруживаются разве что случайно. Если вам необходимо диагностировать проблемы производительности одного четко определенного взаимодействия, то придется использовать совершенно другой подход.

Чрезвычайно полезный в проверке сложных сетевых транзакций в распределенной системе метод называется *трассировкой*. Для выполнения трассировки каждый компонент в распределенной системе должен иметь схожий измерительный код, который помечает все входящие и исходящие связи с уникальными идентификаторами транзакций. Если измеряющий сервис получает запрос с неким идентификатором транзакции (или несколькими) и должен во время данной обработки запроса опросить другие сервисы, то добавляет эти идентификаторы своих собственных запросов и создает новый идентификатор на каждый запрос. В системах, где большинство коммуникаций происходит через протокол HTTP, естественный механизм передачи этих идентификаторов транзакций — HTTP-заголовки. Благодаря этому каждую транзакцию можно расчленить на несколько подтранзакций и становится возможным отследить трафик, который потребовался на каждое взаимодействие с пользователем (рис. 13.6).

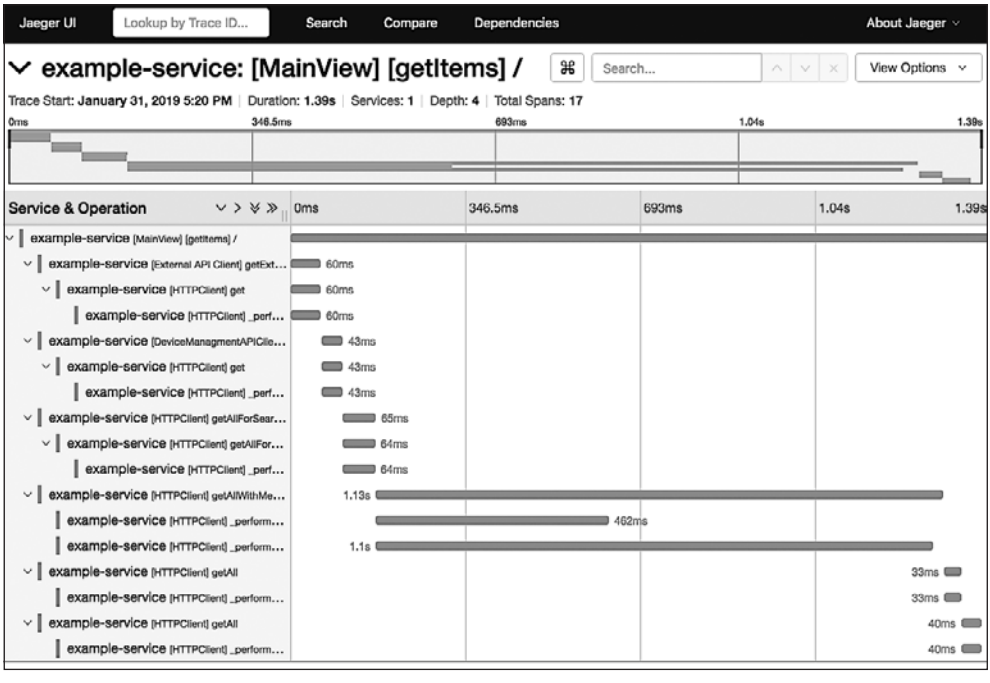


Рис. 13.6. Пример трассировки некоего сервиса в трассере Jaeger

Обычно каждый сервис регистрирует все обработанные транзакции на вторичном сервисе, отвечающем за сбор данных трассировки, а также добавляет метаданные, например время начала и окончания транзакции, имя хоста и количество переданных байтов. Транзакции с метками времени и тегами часто называют промежутками. Многие системы трассировки позволяют определить пользовательские метаданные, которые будут включены в промежутки для облегчения отладки и мониторинга.

Наиболее важным аспектом в любом решении трассировки — тщательный выбор протокола трассировки и системы сбора/агрегации. Неплохим выбором будет инструмент OpenTracing (opentracing.io), который рекламировали как «последовательный, выразительный, не зависящий от поставщика API для распределенной трассировки и распространения контекста». Он предоставляет официальные библиотеки для девяти языков программирования (Python, Go, JavaScript, Java, Ruby, PHP, Objective-C, C++, C#), поэтому подходит даже для команд, которые строят свои продукты с использованием различных технологических стеков. OpenTracing — ни стандарт, ни полноценная программа. Это спецификация API, коллекция фреймворков и библиотек, а также документации. Библиотеки OpenTracing позволяют измерять ваш код и подключаться к *трассерам*, которые

являются фактическими системами сбора и отображения результатов трассировки. Так, хорошая реализация трассера — Jaeger (www.jaegertracing.io). Установка занимает всего пару минут с помощью готового образа Docker, опубликованного на Docker Hub под названием `jaeger`.

Резюме

В данной главе мы узнали три основных правила оптимизации:

- сначала — функционал;
- работа с точки зрения пользователя;
- код должен оставаться читабельным во что бы то ни стало.

Основываясь на этих правилах, мы рассмотрели некоторые из инструментов и методов, позволяющих выявить узкие места в производительности в том или ином аспекте. Теперь, зная, как диагностировать и выявлять проблемы производительности, вы сможете поработать над ними. В следующей главе мы рассмотрим популярные и эффективные методы и стратегии, применимые к подавляющему большинству проблем оптимизации.

14 Эффективные методы оптимизации

Оптимизация — процесс повышения эффективности приложения с сохранением функциональности и точности работы. В предыдущей главе мы узнали, как выявить узкие места и отследить использование ресурсов в коде. Здесь мы узнаем, как с помощью этих знаний можно ускорить работу приложения и более эффективно применять ресурсы.

Оптимизация не магия, а всего лишь простой алгоритм, предложенный Стефаном Шварцером на EuroPython 2006. Оригинальный псевдокод этого примера выглядит следующим образом:

```
def optimize():
    """Recommended optimization"""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                    profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True,
                                    new_bugs=None)
        if not is_faster:
            undo_last_code_change()

# By Stefan Schwarzer, EuroPython 2006
```

Возможно, этот пример не столь яркий и иллюстративный, но зато охватывает почти все важные аспекты процедуры оптимизации. Основные моменты, которые нам следует из него уяснить, таковы:

- оптимизация — итерационный процесс, в котором не каждая итерация улучшает ваши результаты;
- код обязательно проверяется с помощью тестов;
- оптимизация узких мест является ключевой.

Заставить код работать быстрее нелегко. В случае с абстрактными математическими задачами часто все зависит от выбора правильного алгоритма и подходящей

структуры данных. Однако трудно выделить какие-то общие или универсальные советы и приемы, подходящие для любой алгоритмической задачи. Есть, конечно, некоторые общие методики разработки нового алгоритма или даже метаэвристики, применяемые к большому разнообразию задач, но они, как правило, зависят от языка и вследствие этого выходят за рамки данной книги.

Существует множество проблем производительности, связанных с дефектами качества кода или с контекстом использования приложений. Такого рода проблемы часто решаются с помощью общих подходов к программированию или конкретных ориентированных на производительность библиотек и сервисов либо за счет применения правильной архитектуры программного обеспечения. К типичным неалгоритмическим причинам плохой производительности приложений относятся:

- неправильное использование основных встроенных типов;
- чрезмерная сложность;
- применение моделей аппаратных ресурсов, которые не соответствуют среде выполнения;
- длительное время отклика от сторонних API или сервисов;
- слишком большие затраты времени в критических частях приложения.

Чаще всего решение таких проблем с производительностью требует не передовых научных знаний, а лишь хорошего ПО и опыта — ведь большая часть мастерства заключается в правильном и своевременном использовании подходящих инструментов. К счастью, существуют хорошо известные модели и способы для решения проблем с производительностью.

В этой главе мы обсудим некоторые популярные и многоразовые решения, которые позволяют алгоритмически оптимизировать вашу программу, и рассмотрим следующие темы:

- определение сложности;
- снижение сложности;
- использование архитектурных компромиссов;
- кэширование.

Технические требования

Ниже представлен пакет Python, упомянутый в этой главе, который можно скачать с PyPI:

- `pymemcached`.

Установить этот пакет можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter14.

Определение сложности

Прежде чем углубляться в методы оптимизации, разберемся, с чем имеем дело. Из введения к этой главе мы узнали, что акцент на улучшении узких мест приложения имеет решающее значение для успешной оптимизации. Под узким местом понимается единичный компонент, который существенно ограничивает производительность системы или компьютерной программы. У низкопроизводительного кода обычно есть всего одно узкое место. В предыдущей главе мы обсудили некоторые методы профилирования, поэтому вы уже должны знать, какие инструменты используются для обнаружения и изоляции проблемных частей кода. Если результаты профилирования показывают, что в коде есть несколько мест, требующих немедленного улучшения, то стоит попытаться рассматривать каждую часть как отдельный компонент и оптимизировать их независимо друг от друга.

Если в вашем приложении нет явного или очевидного узкого места, но проблемы с производительностью присутствуют, то вы можете оказаться в затруднительном положении. Успех процесса оптимизации пропорционален качеству оптимизации узкого места, и потому бессмысленно оптимизировать каждый крошечный компонент, особенно при отсутствии существенного влияния его на производительность или потребление ресурсов. Если в приложении не видно явных узких мест, то вы, возможно, пропустили что-то во время профилирования. Попробуйте использовать различные стратегии или инструменты профилирования или взглянуть на производительность приложения с другой стороны, например с точки зрения применения процессора, памяти, операций ввода/вывода или пропускной способности сети. Если это не помогает, то следует поразмыслить о пересмотре архитектуры программного обеспечения.

После успешного обнаружения того самого компонента, который ограничивает производительность вашего приложения, можно начинать процесс оптимизации. Велика вероятность того, что даже небольшое улучшение кода значительно ускорит выполнение кода или уменьшит использование ресурсов. Как мы уже упоминали ранее, преимущества оптимизации пропорциональны размеру узкого места.

Первое и самое очевидное, на что нужно обратить внимание при попытке улучшить производительность приложения, — его сложность. Но само это понятие определяется по-разному. Некоторые метрики сложности позволяют получить объективную информацию о том, как ведет себя код, и такая информация часто экстраполируется на производительность. Опытный программист может даже достоверно предположить, как будут работать на практике две разные реализации, если знает об их сложности и контексте выполнения.

Два наиболее популярных способа определить сложность приложения таковы:

- *цикломатическая сложность*, которая очень часто коррелирует с производительностью приложения;
- *обозначение Ландау*, или *асимптотическое обозначение*, — метод классификации алгоритма, полезный при объективной оценке производительности кода.

Оптимизация иногда понимается как процесс снижения сложности. В следующих подразделах мы более подробно рассмотрим определение этих двух типов сложности кода.

Цикломатическая сложность

Цикломатическая сложность — метрика, разработанная Томасом Маккейбом в 1976 году, известная также как *сложность Маккейба*. Цикломатическая сложность измеряет количество линейных путей через фрагмент кода. Смысл такой: все точки ветвления (операторы `if`) и циклы (`for` и `while`) увеличивают сложность кода.

В зависимости от значения измеряемой цикломатической сложности коду можно присвоить определенный класс сложности. В табл. 14.1 приведены часто используемые классы сложности Маккейба.

Таблица 14.1. Часто используемые классы сложности Маккейба

Цикломатическая сложность	Что означает
От 1 до 10	Код несложный
От 11 до 20	Умеренно сложный
От 21 до 50	Действительно сложный
Более 50	Слишком сложный

Цикломатическая сложность — скорее показатель качества кода, нежели объективная метрика производительности. Эта метрика не заменяет необходимость в профилировании кода при анализе узких мест в производительности. В коде с высокой цикломатической сложностью, как правило, используются довольно сложные алгоритмы, которые плохо работают с большими входными данными.

Хотя цикломатическая сложность не слишком надежный способ оценки работы приложения, у нее есть важное преимущество: это метрика исходного кода, которую можно измерить с помощью соответствующих инструментов. Другие канонические способы измерения сложности, включая асимптотическое обозначение, такого преимущества не имеют. Благодаря своей измеримости цикломатическая сложность может быть полезным дополнением к профилированию, поскольку дает

дополнительную информацию о проблемных участках вашего программного обеспечения. Именно со сложных участков кода следует начинать при рассмотрении вопроса о рефакторинге архитектуры.

Измерение сложности Маккейба в Python выполняется относительно просто, поскольку выводится из абстрактного синтаксического дерева. Конечно, вам не нужно делать это самостоятельно, поскольку есть инструмент `pycodestyle` (с плагином `mccabe`), о котором мы говорили в главе 6.

Нотация «О большое»

Наиболее канонический метод определения сложности функции — *нотация «О большое»*. Эта метрика определяет, как работа алгоритма зависит от размера входных данных, например линейно или квадратично.

Вычисление данного показателя алгоритма вручную — лучший способ понять, как его производительность зависит от размера входных данных. Знание сложности компонентов вашего приложения позволяет сфокусироваться на моментах, которые значительно замедляют код.

Для измерения асимптотического обозначения удаляются все константы и части выражения более низкого порядка (термы), чтобы сосредоточиться на той части, которая сильно зависит от объема входных данных. Идея заключается в том, чтобы попытаться классифицировать алгоритм в одну из следующих категорий, даже если это приближение (табл. 14.2).

Таблица 14.2. Категории

Обозначение	**Тип**
O(1)	Постоянная, не зависит от входных данных
O(n)	Линейно растет по мере роста n
O(n log n)	Квазилинейная
$O(n^2)$	Квадратичная сложность
$O(n^3)$	Кубическая сложность
O(n!)	Факториальная сложность

Например, из главы 3 мы уже знаем, что просмотр `dict` имеет среднюю сложность $O(1)$ и считается не зависящим от того, сколько элементов содержится в `dict`. Однако просмотр списка в поисках конкретного элемента имеет сложность $O(n)$.

Чтобы лучше понять эту концепцию, рассмотрим следующий пример:

```
>>> def function(n):
...     for i in range(n):
...         print(i)
...
```

В этой функции оператор `print` будет выполняться n раз. Скорость цикла будет зависеть от `n`, поэтому его сложность будет равна $O(n)$.

Если в функции есть условие, то правильное обозначение будет высшим из имеющихся:

```
>>> def function(n, print_count=False):
...     if print_count:
...         print(f'count: {n}')
...     else:
...         for i in range(n):
...             print(i)
...
```

В данном примере функция может иметь сложность $O(1)$ или $O(n)$ в зависимости от значения аргумента `print_count`. Худший случай — $O(n)$, поэтому вся функция тоже имеет сложность $O(n)$.

Говоря о сложности асимптотического обозначения, мы обычно думаем о наихудшем сценарии. Это лучший способ определить сложность при сравнении двух независимых алгоритмов, однако он не универсален для каждой конкретной ситуации. Многие алгоритмы меняют производительность во время выполнения, в зависимости от статистической характеристики исходных данных, или амортизируют стоимость наихудших операций с помощью различных ухищрений. Именно поэтому часто бывает лучше рассмотреть вашу реализацию с точки зрения *средней сложности* или *амортизированной сложности*.

В качестве примера рассмотрим операцию добавления одного элемента в список Python. Мы знаем, что список в CPython — это массив с перераспределением для внутреннего хранения, а не связанный список (см. главу 3). Если массив уже заполнен, то добавление нового элемента требует выделения нового массива и копирования всех имеющихся элементов (ссылок) в новое место в памяти. Посмотрев на это с точки зрения *наихудшего случая сложности*, мы поймем, что метод `list.append()` имеет сложность $O(n)$, что очень дорого по сравнению с типичной реализацией связанного списка. Мы также знаем, что реализация списков в CPython использует механизм избыточного выделения (выделяет больше места, чем требуется в данный момент времени), чтобы смягчить сложность случайного перераспределения. Если оценивать эту сложность по последовательности операций, то станет очевидно, что средняя сложность `list.append()` стремится к $O(1)$, и это на самом деле отличный результат.

При решении задач мы обычно знаем, какими будут входные данные, включая их количество и статистическое распределение. При оптимизации приложения всегда стоит использовать каждую крупицу имеющихся знаний о входных данных. Существует еще одна проблема — наихудший случай сложности. Асимптотическое обозначение предназначено для анализа предельного поведения функции, когда входные данные стремятся к большим значениям или бесконечности, что не всегда справедливо в реальной жизни. Асимптотическое обозначение — отличный

инструмент для определения скорости роста функции, но не даст прямой ответ на простой вопрос о том, какая именно реализация займет минимум времени. Наихудший случай сложности учитывает все подробности реализации и характеристики данных, показывая вам, как программа будет вести себя асимптотически. Он работает при сколь угодно больших входных данных, которые, возможно, даже и не существуют для вашей задачи.

Например, предположим, у вас есть проблемы с данными, состоящими из n независимых элементов. Допустим также, у вас есть два разных способа решения этой проблемы: *программа А* и *программа Б*. Вы знаете, что *А* для решения задачи потребует $100n^2$ операций, а *Б* — $5n^3$. Какой вариант вы бы выбрали?

При больших объемах данных *программа А* будет лучшим выбором, поскольку асимптотически ведет себя лучше, имеет $O(n^2)$ сложность по сравнению с $O(n^3)$ для *программы Б*. Однако, решая простое неравенство $100\ n^2 > 5\ n^3$, можно обнаружить, что при значениях n меньше 20 *программа Б* работает быстрее. Таким образом, зная количество входных данных, можно принять более удачное решение.

В следующем разделе мы рассмотрим, как уменьшить сложность путем выбора соответствующих структур данных.

Уменьшение сложности через выбор подходящей структуры данных

Чтобы уменьшить сложность кода, важно подумать, как именно хранятся данные. Нужно тщательно выбрать структуру данных. В следующем подразделе приведено несколько примеров того, как можно улучшить производительность простых фрагментов за счет правильных типов данных.

Поиск в списке

Реализация типа `list` в Python такова, что поиск определенного значения в списке — затратная операция. Сложность метода `list.index()` равна $O(n)$, где n — количество элементов списка. Это не будет проблемой, если вам не приходится совершать множество проходов по списку, но может негативно сказаться на производительности в критических участках кода, особенно если действие производится на очень больших списках.

Если вам нужно быстро и часто выполнять поиск по списку, то можете попробовать модуль `bisect` из стандартной библиотеки Python. Функции этого модуля в основном предназначены для вставки или поиска индексов вставки для заданных значений таким образом, чтобы сохранить порядок уже отсортированной последо-

вательности. Этот модуль используется для эффективного поиска индекса элемента с помощью алгоритма половинного деления. Функция ниже из официальной документации функции находит индекс элемента с помощью двоичного поиска:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
```

Обратите внимание: всем функциям модуля `bisect` требуется отсортированная последовательность. В случае неотсортированного списка сортировка будет иметь сложность не менее $O(n \log n)$. Это хуже, чем $O(n)$, и потому использование модуля не окупится. Тем не менее, если вам необходимо выполнить несколько поисков по редко изменяющемуся большому списку, `bisect` — то, что вам нужно.

Если у вас уже есть отсортированный список, то можете также добавлять в него новые элементы с помощью `bisect`, не прибегая к повторной сортировке.

Далее мы увидим, как использовать множества вместо списка.

Использование множеств. Если вам необходимо построить последовательность различных значений из данной последовательности, то первый алгоритм, который может прийти вам на ум, выглядит следующим образом:

```
>>> 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 предыдущем примере сложность вводится просмотром в списке `result` с оператором `in`, имеющим временную сложность $O(n)$. Затем он используется в цикле, который стоит $O(n)$. Таким образом, общая сложность квадратичная, то есть $O(n^2)$.

Использовать тип `set` для той же задачи будет быстрее, поскольку сохраненные значения ищутся с помощью хеша, как и в `dict`. Тип `set` также обеспечивает уникальность элементов, поэтому не нужно ничего делать, кроме создания множества из объекта `sequence`. Другими словами, для каждого значения в `sequence` затрачиваемое на поиск объекта время будет одинаковым:

```
>>> sequence = ['a', 'a', 'b', 'c', 'c', 'd']
>>> unique = set(sequence)
>>> unique
set(['a', 'c', 'b', 'd'])
```

Это снижает сложность до $O(n)$, то есть до сложности создания множества. Еще одно преимущество использования множеств — простота организации уникальности элементов.

Пытаясь уменьшить сложность алгоритма, тщательно рассмотрите ваши структуры данных.

В следующем разделе мы рассмотрим коллекции.

Использование модуля collections

Модуль `collections` предоставляет альтернативные высокопроизводительные встроенные типы контейнеров. Основные типы, имеющиеся в этом модуле, таковы:

- `deque` — похожий на список тип с дополнительными функциями;
- `defaultdict` — словарь со встроенной функцией по умолчанию;
- `namedtuple` — кортеж с ключами для элементов.

Мы обсудим эти типы ниже.

Тип deque

Тип `deque` — альтернативная реализация списков. Встроенный тип `list` основан на обычных массивах, а `deque` — это дважды связанный список. Следовательно, `deque` работает гораздо быстрее, когда нужно вставить что-то в середину или с краю, но куда медленнее, когда надо получить доступ к произвольному индексу.

Конечно, благодаря чрезмерному выделению внутреннего массива в типе `list` Python не каждый вызов `list.append()` требует перераспределения памяти, а средняя сложность этого метода равна $O(1)$. Тем не менее удаление и присоединение, как правило, работает быстрее в связанных списках, а не в массивах. Ситуация резко меняется, когда нужно добавить элемент в произвольную точку последовательности. Поскольку приходится сдвигать все элементы справа от нового, сложность `list.insert()` равна $O(n)$. Если вам нужно выполнить много удалений и вставок, то использование `deque` вместо `list` существенно улучшит производительность. Всегда профилируйте код перед переключением с `list` на `deque`, поскольку есть вещи, которые в массивах работают быстро (например, доступ к произвольному индексу), а в связанных списках — ужасно медленно.

Например, если мы измерим время, необходимое для добавления одного элемента и его удаления из последовательности, с помощью `timeit`, то разница между `list` и `deque` может быть незаметной:

```
$ python3 -m timeit \
> -s 'sequence=list(range(10))' \
> 'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
$ python3 -m timeit \
> -s 'from collections import deque; sequence=deque(range(10))' \
> 'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
```

Однако если мы захотим добавить и удалить первый элемент последовательности, то разница в производительности будет громадной:

```
$ 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
$ python3 -m timeit \
> -s 'from collections import deque; sequence=deque(range(10))' \
> 'sequence.appendleft(0); sequence.popleft()'
10000000 loops, best of 3: 0.172 usec per loop
```

Видно, что разница увеличивается по мере роста размера последовательности. Ниже приведено выполнение того же теста в списках из 10 000 элеметов:

```
$ python3 -m timeit \
> -s 'sequence=list(range(10000))' \
> 'sequence.insert(0, 0); sequence.pop(0)'
100000 loops, best of 3: 14 usec per loop
$ python3 -m timeit \
> -s 'from collections import deque; sequence=deque(range(10000))' \
> 'sequence.appendleft(0); sequence.popleft()'
10000000 loops, best of 3: 0.168 usec per loop
```

Благодаря эффективности методов `append()` и `pop()`, которые работают с одинаковой скоростью с обоих концов последовательности, `deque` прекрасно подходит для реализации очередей. Например, очередь FIFO (first in first out) будет гораздо более эффективной, если будет реализована с помощью `deque`, а не `list`.

Модуль deque хорошо работает при реализации очередей. Начиная с Python 2.6, в стандартной библиотеке Python есть модуль queue, который обеспечивает базовую реализацию FIFO, LIFO и приоритетных очередей. Если вы хотите использовать очереди как механизм связи между потоками, то применяйте классы из модуля queue, а не collections.deque. Это связано с тем, что в данных классах есть вся необходимая семантика. Если вы не используете многопоточность и предпочитаете не применять очереди в качестве механизма связи, то deque для вашей задачи хватит.

Тип defaultdict

Тип `defaultdict` аналогичен типу `dict`, за исключением того, что содержит функцию по умолчанию для новых ключей. Это избавляет от необходимости писать дополнительные тесты, чтобы инициализировать отображение, а также более эффективно, чем метод `dict.setdefault`.

Тип `defaultdict` выглядит как синтаксический сахар для `dict`, который позволяет писать более короткий код. Тем не менее возврат к заранее определенному значению при неправильном ключе работает чуть быстрее, чем метод `dict.setdefault()`, следующим образом:

```
$ python3 -m timeit \
> -s 'd = {}'
> 'd.setdefault("x", None)'
10000000 loops, best of 3: 0.153 usec per loop
$ python3 -m timeit \
> -s 'from collections import defaultdict; d=defaultdict(lambda: None)' \
> 'd["x"]'
10000000 loops, best of 3: 0.0447 usec per loop
```

Разница в предыдущем примере невелика, поскольку вычислительная сложность не изменилась. Метод `dict.setdefault` состоит из двух этапов (поиск ключа и его установка), и оба имеют сложность $O(1)$, как мы видели в главе 3, в пункте «Словари» раздела «Встроенные типы языка Python». Класса сложности ниже $O(1)$ не бывает, но даже небольшое улучшение скорости в отдельно взятых случаях важно при оптимизации критических участков кода.

Тип `defaultdict` принимает значение по умолчанию в качестве параметра и, следовательно, может быть использован со встроенными типами или классами, конструкторы которых не принимают аргументы. Следующий фрагмент кода — это пример из официальной документации, показывающий, как использовать `defaultdict` для выполнения подсчета:

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> list(d.items())
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

Тип namedtuple

Тип `namedtuple` — конструктор класса, который принимает имя типа и список атрибутов и создает из него класс. Класс может быть использован для создания экземпляра кортежеподобного объекта, а также обеспечивает механизм доступа к его элементам, а именно:

```
>>> from collections import namedtuple
>>> Customer = namedtuple(
...     'Customer',
...     'firstname lastname'
... )
>>> c = Customer('Tarek', 'Ziade')
>>> c.firstname
'Tarek'
```

Как показано в предыдущем примере, этот механизм можно применять для создания записей, которые легче писать по сравнению с пользовательским классом, для инициализации значений в котором требуется шаблонный код. С другой стороны, в основе лежит `tuple`, поэтому получить доступ к его элементам по индексу можно быстро. Сгенерированный класс также может быть субклассифицирован для добавления операций.

Преимущество использования `namedtuple`, по сравнению с другими типами данных, на первый взгляд неочевидно. Основное преимущество заключается в том, что данный тип легче применять, понимать и интерпретировать, чем обычные кортежи. Индексы кортежей не несут никакой семантики, поэтому хорошо бы иметь возможность доступа к элементам кортежей по атрибутам. Обратите внимание: того же эффекта можно добиться с помощью словарей, которые имеют сложность операций `get` и `set` $O(1)$.

Первое преимущество `namedtuple` с точки зрения производительности заключается в том, что это кортеж. То есть он является неизменяемым, вследствие чего выделенный массив будет иметь нужный размер. В словарях, с другой стороны, приходится задействовать чрезмерное выделение памяти для снижения затратности операций. Поэтому `namedtuple` превосходит `dict` с точки зрения эффективности использования памяти.

Тот факт, что `namedtuple` — кортеж, также может послужить для повышения производительности. К его элементам можно обращаться с помощью целочисленного индекса, как и у других подобных типов. Эта операция простая и быстрая. В случае экземпляров `dict` или пользовательских классов, которые применяют словари для хранения атрибутов, доступ к элементу требует поиска в хеш-таблице. Этот тип хорошо оптимизирован, чтобы гарантировать хорошую производительность независимо от размера коллекции, но, как уже упоминалось, сложность $O(1)$ — *средняя*. Фактически амортизированный наихудший случай сложности для операций `set/get` — $O(n)$. Реальный объем работ, необходимый для выполнения такой операции, зависит от размера и истории коллекции. В разделах кода, которые имеют решающее значение для производительности, может быть разумно использовать списки или кортежи, а не словари. И в такой ситуации `namedtuple` — отличное решение, сочетающее преимущества словарей и кортежей:

- в разделах, где важнее читабельность, предпочтителен доступ по атрибутам;
- в критических по производительности фрагментах к элементам можно обращаться по индексам.

Снижение сложности достигается за счет хранения данных в эффективной структуре данных, которая хорошо работает для этой задачи. Тем не менее, когда решение неочевидно, следует просто переписать код заново, а не пытаться «вылечить» его, тем самым уничтожая читабельность.

Код Python обычно хорошо читается и быстро работает, поэтому постарайтесь найти хороший способ выполнить работу, вместо того чтобы пытаться обойти несовершенный дизайн.

В следующем разделе мы рассмотрим, как применять архитектурные компромиссы.

Использование архитектурных компромиссов

Когда улучшить код за счет уменьшения сложности или выбора правильной структуры данных больше не получается, стоит пойти на компромисс. Если мы рассматриваем проблемы пользователей и определили действительно важные для них аспекты, то можем слегка снизить некоторые требования приложения. Производительность улучшится, если:

- заменить алгоритмы точного решения эвристическими или приближенными;
- отложить выполнение некоторых задач;
- использовать вероятностные структуры данных.

Рассмотрим эти методы более подробно.

Использование эвристических алгоритмов или приближенных вычислений

Часть алгоритмических задач попросту не имеет хороших современных решений, которые выполнялись бы в течение времени, приемлемого для пользователя. В качестве примера рассмотрим программу, выполняющую сложные задачи оптимизации, такие как *задача коммивояжера* (Traveling Salesman Problem, TSP) или *задача маршрутизации транспорта* (Vehicle Routing Problem, VRP). Это сложные задачи комбинаторной оптимизации. Точных алгоритмов с низкой сложностью для таких задач не существует, и объем доступной памяти для решения задачи в значительной степени ограничен. При больших объемах входных данных маловероятно, что вы сможете вычислить правильное решение за достаточно короткое время.

К счастью, для пользователя важно не идеальное решение, а его своевременность при нормальном качестве. В таких случаях имеет смысл применить *эвристические* или *приближенные алгоритмы*, если они дают приемлемые результаты.

- Эвристика позволяет решить задачу, жертвуя оптимальностью, полнотой, точностью ради скорости. Она концентрируется на скорости, но при этом может быть трудно доказать качество решений по сравнению с результатом точных алгоритмов.
- Приближенные алгоритмы похожи на эвристические, но позволяют оценить и доказать качество решения.

Существует много известных методов эвристики и приближения, которые позволяют решить крупные задачи TSP в течение разумного периода времени. Они также имеют высокую вероятность получения результатов в диапазоне 2–5 % от оптимального решения.

Еще один положительный аспект: эвристику не всегда нужно создавать для каждой задачи заново. Более высокоуровневые варианты эвристик, называемые *метаэвристиками*, обеспечивают стратегии решения математических задач оптимизации, которые не всегда привязаны к конкретной задаче и могут быть применены во многих ситуациях. Ниже приведены некоторые популярные метаэвристические алгоритмы:

- имитация отжига;
- генетические алгоритмы;
- поиск табу;
- оптимизация колонии муравьев;
- эволюционное вычисление.

Применение очереди задач и отложенная обработка

Иногда важнее сделать не как можно больше, а вовремя. Типичный пример, который часто упоминается в литературе, — отправка сообщений электронной почты в веб-приложении. В этом случае увеличение времени отклика HTTP-запросов не обязательно переводить в реализацию кода. Время отклика может управляться сторонним сервисом, например удаленным сервером электронной почты. А возможно ли тогда успешно оптимизировать приложение, если большую часть оно ожидает чьих-то ответов?

Ответ: и да и нет. Если сервис, который является основным источником временных затрат, не в вашей власти, то вы не можете ускорить работу. Простой

пример обработки HTTP-запроса, приводящего к отправке по электронной почте, представлен на следующей схеме (рис. 14.1). Здесь вы не можете сократить время ожидания, но можете повлиять на восприятие пользователя.

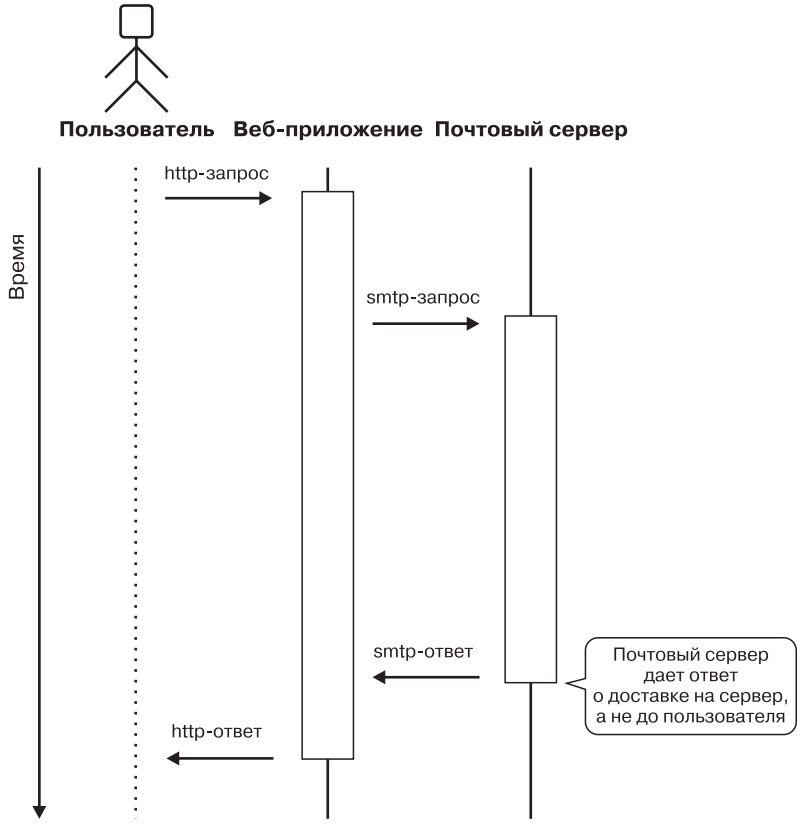


Рис. 14.1. Пример синхронной доставки электронной почты в веб-приложении

Обычное решение подобной проблемы — использование очереди сообщений или задач. Когда вам нужно выполнить некие действия, которые могут занять неопределенное количество времени, задача добавляется в очередь, а пользователь получает сообщение, что запрос был принят. Поэтому отправка сообщений электронной почты — хороший пример, ведь электронная почта так и устроена! Если вы отправляете новое сообщение на сервер электронной почты с помощью протокола SMTP, то успешный ответ означает доставку письма не адресату, а этому серверу. Если ответ от сервера не гарантирует доставку электронной почты, то вам не нужно ждать сигнала, чтобы сгенерировать ответ для пользователя.

Такая обработка задач представлена на схеме ниже (рис. 14.2).

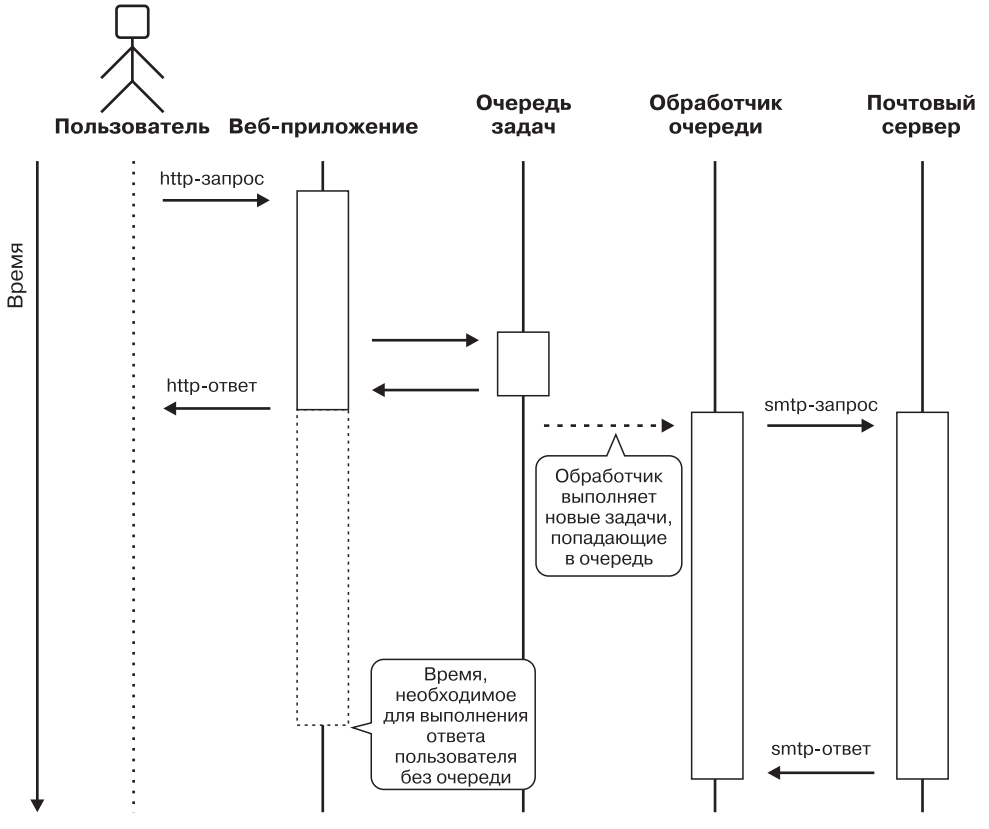


Рис. 14.2. Пример асинхронной доставки электронной почты в веб-приложении

Даже если ваш почтовый сервер отвечает мгновенно, возможно, потребуется еще некоторое время, чтобы сгенерировать сообщение, которое нужно отправить. Вы составляете годовые отчеты в формате XLS? Или передаете счета в виде PDF-файлов? При использовании транспортной системы электронной почты, которая сама по себе работает асинхронно, вы можете поместить всю задачу формирования сообщений в систему обработки сообщений. Если вы не можете гарантировать точное время доставки, то не стоит думать и о синхронности результатов.

Правильное использование очередей задач и сообщений в критических участках приложения также может дать следующие преимущества.

- Веб-работники, обслуживающие HTTP-запросы, будут освобождены от дополнительной работы и смогут обрабатывать запросы быстрее. Это значит, что вы

сможете обрабатывать больше запросов с теми же затратами ресурсов и, таким образом, бóльшую нагрузку.

- Очереди сообщений, как правило, более устойчивы к переходным отказам внешних сервисов. Например, если база данных или сервер электронной почты время от времени выходит из строя, то вы всегда можете перезапустить очередь и продолжить обработку позже.
- Имея хорошую реализацию очереди сообщений, вы можете легко распределить работу на несколько компьютеров. Такой подход позволяет улучшить масштабируемость компонентов вашего приложения.

Как было видно на предыдущей схеме (см. рис. 14.2), добавление асинхронного выполнения задач неизбежно увеличивает сложность архитектуры системы в целом. Вам потребуется установить дополнительные сервисы (очереди сообщений, такие как RabbitMQ) и создать работников, которые смогут действовать асинхронно. К счастью, для построения распределенных очередей задач уже есть готовые инструменты. Самый популярный среди разработчиков на Python — *Celery* (www.celeryproject.org). Это полноценная структура очереди задач с поддержкой нескольких брокеров сообщений, позволяющая реализовать запланированное выполнение задач. Она даже может заменить `cron`. Если вам нужно что-то попроще, то может подойти RQ (python-rq.org). Он намного проще, чем Celery, и в качестве брокера сообщений использует ключ/значение Redis (*RQ* так и расшифровывается — *Redis Queue*).

Многие инструменты уже проверены и показали себя на практике, но вам следует внимательно продумать свой подход к построению очереди задач. Не любую задачу можно обрабатывать в очередях. Они, безусловно, хороши в решении ряда вопросов, но у них бывают и проблемы:

- рост сложности архитектуры системы;
- возможно *более одной* передачи;
- нужны дополнительные сервисы для сопровождения и мониторинга;
- увеличивается задержка обработки;
- затрудняется ведение журнала.

Использование вероятностной структуры данных

Вероятностными называются структуры данных, предназначенные для хранения коллекций значений таким образом, который позволяет отвечать на конкретные вопросы в течение малого времени в условиях ограничений. Самая важная особенность вероятностных структур данных заключается в том, что ответы, которые они дают, *могут быть* правильными. То есть это не истина, а приближение реальных значений. Однако вероятность правильного ответа можно легко оценить. Несмотря

на то что он дается не всегда, вероятностные структуры данных по-прежнему полезны, если возможны ошибки.

Существует много структур данных с вероятностными свойствами. Каждая из них решает конкретные проблемы, но из-за стохастического характера эти структуры годятся не для всех задач. В качестве практического примера мы поговорим об одной из наиболее популярных структур — *HyperLogLog*.

HyperLogLog — это алгоритм, который аппроксимирует количество различных элементов в мультимножестве. В обычных множествах, если вы хотите узнать количество уникальных элементов, вы должны хранить их все. Для больших наборов данных это, очевидно, нецелесообразно. HLL отличается от классического пути реализации множества. Не углубляясь в детали реализации, скажем просто, что акцент здесь делается на аппроксимации эффективности множества, при этом реальные значения не сохраняются. Обратиться к ним нельзя, проитерировать нельзя, проверить на членство тоже нельзя. HyperLogLog жертвует точностью и правильностью ради снижения затрат времени и потребления памяти. Например, реализация Redis занимает всего 12 Кбайт со стандартной погрешностью 0,81 %, при этом практически не имеет ограничений на размер коллекции.

Использование вероятностной структуры данных — интересный способ решать проблемы с производительностью. В большинстве случаев мы жертвуем некоторой точностью ради более быстрой обработки и более эффективного применения ресурсов. Однако это требуется не всегда. Вероятностные структуры данных часто задействуются в системах хранения пар «ключ — значение» для ускорения поиска по ключам. Один из наиболее популярных методов, используемых в подобных системах, — *приближенный запрос* (approximate member query, AMQ). Существует также фильтр Блума — интересная вероятностная структура данных, которая часто используется для этой цели.

В следующем разделе мы поговорим о кэшировании.

Кэширование

Если вычисление некоторых функций приложения занимает слишком много времени, то стоит подумать о кэшировании. Оно сохраняет возвращаемые значения вызовов функций, запросы к базам данных, HTTP-запросы и т. д. «про запас» на будущее. Результат «дорогой» функции или метода можно кэшировать в случае выполнения следующих требований:

- функция детерминированная, то есть при одинаковых входных данных всегда дает одинаковые результаты;
- возвращаемое значение функции недетерминировано, но остается полезным и актуальным в течение некоторого периода времени.

Другими словами, детерминированная функция всегда возвращает один и тот же результат для одного и того же набора аргументов, тогда как недетерминированная функция возвращает разные результаты. Кэширование обоих типов результатов, как правило, значительно сокращает время вычисления и позволяет сэкономить много ресурсов компьютера.

Самое важное требование для системы кэширования таково: это должна быть система хранения данных, которая позволяет извлекать сохраненные значения значительно быстрее, чем они вычисляются. Примеры того, для чего можно использовать кэширование, представлены ниже:

- результаты вызываемых объектов, которые делают запросы к базам данных;
- результаты вызываемых объектов, обрабатывающие статические значения, такие как содержимое файла, веб-запросы или PDF;
- результаты детерминированных функций со сложными вычислениями;
- глобальные отображения, отслеживающие временные значения, такие как объекты веб-сессии;
- результаты, которые нужны часто и быстро.

Другой важный пример использования кэширования — сохранение результатов от сторонних API, работающих через Интернет. Это может значительно повысить производительность приложений, сводя на нет задержки сети, а также позволяет сэкономить деньги, если вам выставляют счет за каждый запрос к API.

В зависимости от архитектуры приложения кэш может быть реализован разными способами и с различными уровнями сложности. Существует много способов предоставления кэша, и в сложных приложениях могут использоваться различные подходы на разных уровнях архитектуры приложения. Иногда кэш реализован просто как глобальная структура данных (обычно `dict`), хранящаяся в памяти процесса. Иногда вы также можете создать специальный сервис кэширования, который будет работать на тщательно настроенном оборудовании. В этом разделе будет приведена основная информация о наиболее популярных подходах к кэшированию, мы рассмотрим несколько общих случаев их применения, а также распространенные ошибки.

Итак, рассмотрим, что такое детерминированное кэширование.

Детерминированное кэширование

Детерминированные функции проще и безопаснее всего с точки зрения кэширования. Они всегда возвращают одно и то же значение, если им подаются одинаковые данные, поэтому хранить результаты можно сколь угодно долго. Единственное ограничение данного подхода — объем памяти запоминающего устройства.

Самый простой способ кэшировать результаты — поместить их в память процесса, поскольку это, как правило, самое быстрое место для извлечения данных. Такой метод часто называют *меморизацией*.

Меморизация очень полезна при оптимизации рекурсивных функций, которые могут потребоваться для оценки одних и тех же данных несколько раз. (Мы уже обсуждали рекурсивные реализации для последовательности Фибоначчи в главе 9.) Ранее в этой книге мы попытались улучшить работу нашей программы с помощью C и Cython. Теперь мы попробуем достичь той же цели более простыми средствами кэширования. Но прежде вспомним код функции `fibonacci()`:

```
def fibonacci(n):
    """Возвращает n-й член последовательности Фибоначчи, вычисленный рекурсивно
    """
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Очевидно, `fibonacci()` — рекурсивная функция, которая вызывает себя дважды, если значение входа превышает 2. Это делает алгоритм крайне неэффективным. Сложность выполнения равна $O(2^n)$, и выполнение создает очень глубокое и широкое дерево вызова. При больших входных значениях функция будет выполняться очень долго, и существует высокая вероятность превысить максимальный предел рекурсии интерпретатора Python.

Внимательно посмотрев на следующую схему дерева вызова (рис. 14.3), вы увидите, что многие из промежуточных результатов вычисляются несколько раз. Можно было бы сэкономить время и ресурсы, используй мы некоторые из этих значений повторно.

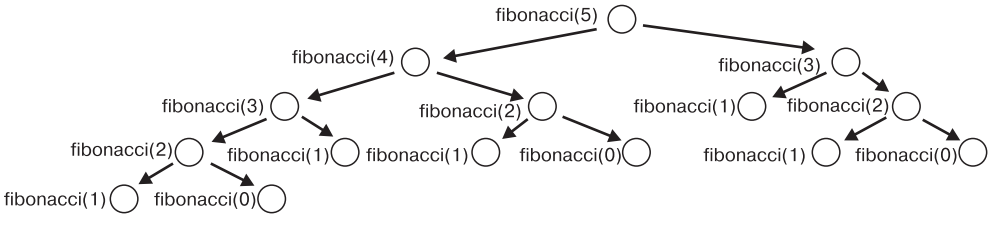


Рис. 14.3. Дерево вызова для fibonacci(5)

В качестве простого примера меморизации можно сохранить результаты предыдущих вызовов в словарь и при необходимости достать их. Оба рекурсивных вызова функции содержатся в одной строке кода:

```
return fibonacci(n - 1) + fibonacci(n - 2)
```

Мы знаем, что Python оценивает инструкции слева направо. Это значит, что в данном случае вызов функции с более высоким значением аргумента будет выполняться перед вызовом функции с более низким значением аргумента. Благодаря этому мы можем обеспечить меморизацию путем построения очень простого декоратора:

```
def memoize(function):
    """Мемоизация вызова функции одного аргумента
    """
    call_cache = {}

    def memoized(argument):
        try:
            return call_cache[argument]
        except KeyError:
            return call_cache.setdefault(
                argument, function(argument)
            )

    return memoized

@memoize
def fibonacci(n):
    """Возвращает n-й член последовательности Фибоначчи, вычисленный рекурсивно
    """
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Мы использовали словарь на замыкании декоратора `memoize()` для хранения данных от кэшированных значений. Сохранение и извлечение значений в такой структуре данных имеет среднюю сложность $O(1)$, вследствие чего значительно снижается общая сложность функции. Каждый уникальный вызов функции будет выполняться всего один раз. Дерево вызовов такой обновленной функции представлено на схеме ниже (рис. 14.4). Даже без математических доказательств визуально видно, что мы снизили сложность функции `fibonacci()` с $O(2^n)$ до $O(n)$.

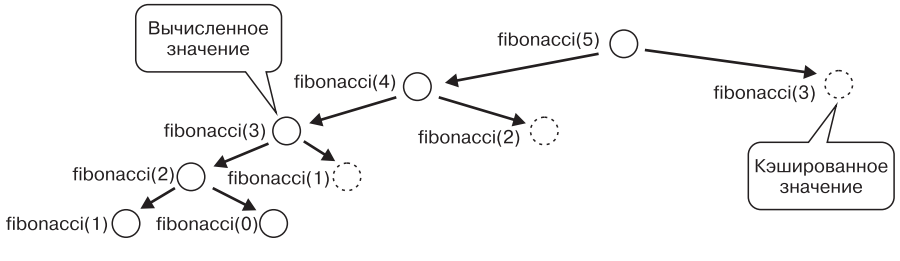


Рис. 14.4. Новое дерево вызова для fibonacci(5)

Реализация нашего декоратора `memoize()`, конечно, неидеальна. Она хорошо сработала в нашем примере, но универсальное решение тут не получится. Если вам нужно запоминать функцию с несколькими аргументами или вы хотите контролировать размер кэша, то требуется нечто более серьезное. К счастью, в стандартной библиотеке Python есть простая и многоразовая утилита, подходящая в большинстве случаев, когда требуется кэширование результатов детерминированных функций. Это утилита `lru_cache(maxsize, typed)` из модуля `functools`. Название происходит от алгоритма LRU, что расшифровывается как lact recently used — «последний использованный». Дополнительные параметры позволяют более точно настроить процесс меморизации:

- `maxsize` — устанавливает максимальный размер кэша. Значение `None` означает отсутствие ограничения;
- `typed` — этот параметр определяет, следует ли кэшировать как один и тот же результат значения различных типов, которые при сравнении дают равенство.

Использование `lru_cache` в нашем примере последовательности Фибоначчи будет выглядеть следующим образом:

```
@lru_cache(None)
def fibonacci(n):
    """Возвращает n-й член последовательности Фибоначчи, вычисленный рекурсивно
    """
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

В следующем подразделе мы рассмотрим недетерминированное кэширование.

Недетерминированное кэширование

Кэширование недетерминированных функций сложнее меморизации. Поскольку каждое выполнение такой функции дает разные результаты, как правило, невозможно использовать имеющиеся значения сколь угодно долго. Тогда нужно определиться, насколько долго кэшируемое значение может считаться актуальным. После определенного времени сохраненные результаты должны считаться устаревшими, и кэш следует обновить.

Недетерминированные функции, которые обычно кэшируются, часто зависят от какого-либо внешнего состояния, трудно отслеживаемого внутри кода приложения. Типичные примеры таких компонентов:

- реляционные базы данных или в целом любые типы движков хранения;
- сторонние сервисы, доступные через сеть (веб-API);
- файловые системы.

Другими словами, недетерминированное кэширование выполняется всегда, если результаты используются хотя бы временно. К ним обычно относятся состояния, согласуемые с другими состояниями.

Следует отметить, что такая реализация, очевидно, является компромиссом и, следовательно, связана с методами, которые мы рассмотрели выше, в разделе «Использование архитектурных компромиссов». Уйдя от исполняемой части вашего кода, когда это необходимо, и вместо этого используя имеющиеся результаты, вы рискуете, поскольку данные могут быть неактуальными и отражать старое состояние. В этом случае вы жертвуете точностью и/или полнотой ради скорости и производительности.

Конечно, такое кэширование будет эффективным в случае, если время, необходимое для взаимодействия с кэшем, меньше времени, которое кэшируемая функция тратит на выполнение. Если попросту пересчитать значение будет быстрее, то нужно делать именно это! Вот поэтому использование кэша разумно, только если дает сильную выгоду по времени.

Обычно кэшируются результаты взаимодействия с другими компонентами системы. Если вы хотите сэкономить время и ресурсы при обмене данными с базой данных, то «дорогие» запросы стоит кэшировать. Желая уменьшить количество операций ввода/вывода, можете кэшировать содержимое файлов, которые нужны программе чаще других.

Методы кэширования недетерминированных функций в целом очень похожи на те, которые используются в детерминированном кэшировании. Важное отличие заключается в том, что в этом случае требуется возможность удаления кэшированных значений, когда они теряют актуальность. Это значит, что декоратор `lru_cache()` из модуля `functools` уже плохо применим, но добавить в него нужный функционал нетрудно. Поскольку это очень распространенная проблема, которая уже неоднократно решалась многими разработчиками, вы без труда найдете несколько библиотек для недетерминированного кэширования.

В следующем подразделе мы рассмотрим сервисы кэширования.

Сервисы кэширования

Мы уже говорили, что недетерминированное кэширование можно реализовать с помощью локальной памяти процесса, однако на самом деле так делают редко. Дело в том, что локальная память не слишком полезна в качестве хранилища для приложений.

Если вы столкнулись с ситуацией, когда для решения проблем производительности нужно прибегнуть к недетерминированному кэшированию, то, возможно, есть и другое решение. Обычно недетерминированное кэширование незаменимо,

когда требуется выдавать данные или сервис нескольким пользователям одновременно. Рано или поздно вам придется сделать так, чтобы пользователи могли обслуживаться одновременно. Локальная память позволяет обмен данными между несколькими потоками, но это не самая удачная модель параллелизма. Она не очень хорошо масштабируется, и вам, возможно, потребуется запустить приложение как несколько процессов.

При удачном стечении обстоятельств ваше приложение будет работать на сотнях или тысячах машин. Если вы хотите в этом случае хранить кэшированные значения в локальной памяти, то придется дублировать кэш для каждого процесса, которому он нужен. Это не просто пустая трата ресурсов: если у каждого процесса есть собственный кэш, который уже является компромиссом между скоростью и последовательностью, то как вы можете гарантировать, что все кэши правильно согласуются друг с другом?

Сохранение порядка при множестве запросов — серьезная проблема, особенно для веб-приложений с распределенными движками. В сложных распределенных системах чрезвычайно трудно гарантировать, что пользователь будет всегда обслуживаться одним и тем же процессом. Это можно реализовать в какой-то степени, но после решения данной проблемы появятся десять других.

Если вы делаете приложение, которое должно обслуживать несколько пользователей одновременно, то лучше всего организовать недетерминистической кэш с помощью выделенного сервиса. Такие инструменты, как Redis или Memcached, позволяют объединить кэш ваших процессов. Это одновременно сокращает затраты драгоценных вычислительных ресурсов и избавляет вас от проблем, вызванных наличием слишком большого количества независимых и несовместимых кэшей.

Сервисы кэширования наподобие Memcached полезны для реализации кэшей с меморизацией для состояний, которые могут применяться в нескольких процессах и даже на нескольких серверах. Существует и другой способ кэширования, легко реализующийся на уровне архитектуры системы, и такой подход весьма распространен в приложениях, работающих по протоколу HTTP. Многие компоненты типичного стека HTTP предоставляют гибкие возможности кэширования, в которых часто используются механизмы, хорошо стандартизированные по протоколу HTTP. Такой вид кэширования может принимать разные формы:

- *кэширование обратного прокси (например, Nginx или Apache)* — если прокси-сервер кэширует отклики от нескольких веб-работников, функционирующих на одном хосте;
- *кэширование балансировки нагрузки (например, HAProxy)* — когда балансировщик нагрузки не только распределяет нагрузку по нескольким узлам, но и кэширует их отклики;

- *сеть распространения контента* — ресурсы с серверов кэшируются системой, которая также пытается держать их в непосредственной географической близости к пользователям, тем самым сокращая время обращения к сети.

В следующем пункте мы рассмотрим Memcached.

Memcached. Для серьезного кэширования вам понадобится Memcached — популярное и давно проверенное решение. Этот сервер кэширования используется многими приложениями, в том числе «Фейсбук» и «Википедия», позволяя удобно масштабировать сайты. Его возможности кластеризации позволяют в считаные мгновения настроить эффективную распределенную систему.

Memcached — мультиплатформенный сервис, и в нескольких языках программирования есть библиотеки для взаимодействия с ним. Есть много клиентов Python, которые немного отличаются друг от друга, но основное использование, как правило, одинаковое. Самое простое взаимодействие с Memcached почти всегда состоит из следующих трех методов:

- `set(key, value)` — сохраняет значение для данного ключа;
- `get(key)` — получает значение для данного ключа, если он существует;
- `delete(key)` — удаляет значение под данным ключом, если он существует.

Следующий фрагмент кода — пример интеграции с Memcached с помощью популярного пакета Python под названием `pymemcached`:

```
from pymemcache.client.base import Client

# Установка Memcached на порт 11211 на локальном хосте
client = Client(('localhost', 11211))

# Кэшируем некоторое значение под каким-либо ключом,
# и оно истекает через 10 секунд
client.set('some_key', 'some_value', expire=10)

# Извлечение значения для того же ключа
result = client.get('some_key')
```

Один из недостатков Memcached таков: он предназначен для хранения значений в виде строк или двоичных массивов, что не дает совместимости с некоторыми нативными типами Python. На самом деле совместимость есть только со строками. Это значит, что более сложные типы для хранения необходимо превращать в особый формат. Обычно для этого используется формат JSON. Пример того, как задействовать JSON с `pymemcached`, представлен ниже:

```
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 проблема, очень часто встречающаяся при работе с сервисом кэширования, который работает по принципу хранения в виде пар«ключ — значение», — это как выбирать имена ключей.

В случае простого кэширования вызовов функций с базовыми параметрами все будет просто. Вы можете преобразовать имя функции и ее аргументы в строки, а затем объединить их. Вам нужно беспокоиться лишь о том, чтобы не было конфликтов между ключами, которые были созданы для различных функций, если выполняете кэширование в разных местах в пределах приложения.

Более проблематичен случай, когда у кэшируемых функций есть сложные аргументы, которые состоят из словарей или пользовательских классов. В этом случае вам нужно будет найти способ преобразования сигнатур в ключи согласованным образом.

Последняя проблема заключается в том, что Memcached, как и многие другие сервисы кэширования, не любит слишком длинных ключевых строк. Чем короче, тем лучше. Длинные ключи либо могут привести к снижению производительности, либо просто не вписываются в установленные ограничения. Например, если вы кэшируете целые запросы SQL, то сами строки запроса, как правило, уникальные идентификаторы, которые можно использовать в качестве ключей. С другой стороны, сложные запросы обычно слишком длинные для сервисов кэширования наподобие Memcached. Обычной практикой является вычисление *MD5*, *SHA* или любой другой хеш-функции и применение ее в качестве ключа. В стандартной библиотеке Python есть модуль `hashlib`, который обеспечивает реализацию на несколько популярных алгоритмов хеширования. Стоит также обратить внимание на конфликты имен. Ни одна хеш-функция не гарантирует отсутствие конфликтов, поэтому всегда нужно быть в курсе потенциальных рисков.

Резюме

В этой главе мы узнали, как определить сложность кода, а также рассмотрели различные способы ее уменьшения. Далее изучили, как улучшить производительность с помощью архитектурных компромиссов. Наконец, исследовали, что такое кэширование и как его использовать для повышения производительности приложений.

Рассмотренные методы были посвящены оптимизации внутри одного процесса. Мы попытались уменьшить сложность кода, выбрать лучшие типы данных, а также использовать старые результаты выполнения функций. Если это не помогло, то старались достичь компромисса с помощью приближения, оставляя работу на потом. Кроме того, кратко обсудили тему очередей сообщений в качестве потенциального решения проблем с производительностью. Мы вернемся к этой теме в главе 16.

В следующей главе мы обсудим некоторые методы параллелизма и параллельной обработки в Python, которые тоже помогают повысить производительность приложений.

15 Многозадачность

Многозадачность и одно из ее проявлений, параллельная обработка, — один из самых сложных вопросов в области программной инженерии. Он столь обширен, что можно написать десятки книг, и даже этого не хватит, чтобы обсудить его в полной мере.

Поэтому мы не будем пытаться обмануть вас и с самого начала скажем, что мы лишь коснемся поверхности данной темы. Цель главы — показать, зачем многозадачность может потребоваться в вашем приложении, когда ее использовать и каковы популярные модели многозадачности, которые вы можете применять в Python:

- асинхронное программирование;
- многопоточность;
- мультипроцессорность.

Мы также обсудим некоторые особенности языка, встроенные модули и сторонние пакеты, которые позволяют реализовать эти модели в вашем коде. Однако обойдемся без лишних подробностей. Содержание главы будет отправной точкой для вашего дальнейшего исследования. Мы раскроем основные идеи и поможем принять решение о том, действительно ли вам нужен параллелизм, и если да, то какой подход наилучшим образом удовлетворит ваши потребности.

В этой главе:

- зачем нужна многозадачность;
- многопоточность;
- многопроцессорность;
- асинхронное программирование.

Технические требования

Пакет Python `aiohttp`, упомянутый в этой главе, можно скачать с PyPI с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по адресу github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter15.

Зачем нужна многозадачность

Прежде чем ответить на вопрос, зачем нужна *многозадачность*, неплохо бы выяснить, *что это вообще такое*?

Ответ на последний вопрос может оказаться удивительным для тех, кто привык думать, что это синоним параллельной обработки. Так вот, это не так. Более того, многозадачность не вопрос реализации приложения. Это свойство программы, алгоритм или задачи, где параллелизм — всего лишь один из возможных подходов к решению.

Лэмпорт в своей работе 1976 года под названием *Time, Clocks and the Ordering of Events in Distributed Systems* определяет понятие многозадачности следующим образом: *«Два события считаются параллельными, если ни одно из них не влияет на другое»*.

Экстраполируя данное утверждение на программы, алгоритмы или задачи, мы можем сказать: что-то выполняется одновременно, если его можно полностью или частично разложить на составные части (единицы), не зависящие друг от друга и от порядка. Такие блоки могут обрабатываться независимо друг от друга, и порядок обработки не влияет на конечный результат. Это значит, что они также могут обрабатываться одновременно или параллельно. Если мы обрабатываем информацию таким способом (то есть параллельно), то речь действительно идет о параллельной обработке. Но это не обязательно.

Выполнение работы распределенным способом, предпочтительно с использованием возможностей многоядерных процессоров или вычислительных кластеров, — естественное следствие многозадачности. Но это не единственный способ ее реализации. Есть много случаев, в которых параллельные задачи можно решить, не прибегая к параллельному выполнению.

Итак, теперь мы знаем, что такое многозадачность, и пора объяснить, зачем она нам. Когда несколько задач выполняется параллельно, это дает вам возможность работать с каждой из них собственным, предпочтительно более эффективным способом.

Мы привыкли к решению проблем классическим способом, то есть через выполнение последовательности шагов. Большинство из нас думает и обрабатывает информацию именно так, используя синхронные алгоритмы, которые позволяют делать по одному действию за раз. Но этот способ обработки информации не слишком хорош для решения масштабных задач или случаев, когда нам нужно удовлетворять следующие требования различных пользователей или программных агентов одновременно:

- когда время обработки задания ограничено производительностью одного блока обработки (одной машины, ядра процессора и т. д.);
- когда вы не в состоянии принимать и обрабатывать новые данные, пока ваша программа не завершит обработку предыдущих данных.

Таким образом, к многозадачному выполнению приходят в следующих случаях:

- когда масштаб проблемы настолько велик, что единственный способ решить ее за приемлемое время или в пределах имеющихся ресурсов — это распределить выполнение на несколько блоков, которые смогут обрабатывать задачу параллельно;
- ваше приложение должно принимать новые данные, даже если обработка старых еще не окончена.

Эти два класса задач охватывают большинство ситуаций, в которых параллельная обработка — разумный вариант. Первая группа проблем, безусловно, нуждается в параллельной обработке и обычно решается с помощью многопоточных и многопроцессорных моделей. Вторая группа не всегда требует параллелизма, и решение будет зависеть от тонкостей задачи. Эта группа проблем охватывает также случай, когда приложение должно обслуживать несколько клиентов (пользователей или программных агентов) независимо друг от друга, не нуждаясь в ожидании успешного обслуживания других.

Интересно, что эти группы проблем не исключают друг друга. Часто бывает нужно сохранить время отклика приложения и в то же время решать задачу обработки на одном процессоре. Поэтому иногда разные и, казалось бы, альтернативные или конфликтующие подходы к параллельности можно использовать одновременно. Это особенно распространено в разработке веб-серверов, где может быть необходимо задействовать асинхронные циклы событий или потоки в сочетании с несколькими процессами, чтобы применять все доступные ресурсы и обеспечивать низкие значения задержки при высокой нагрузке.

В следующем разделе поговорим о многопоточности.

Многопоточность

Разработчики часто считают это очень сложной темой. В общем-то, так и есть, но в Python есть высокоуровневые классы и функции, которые облегчают использование многопоточности. Реализация потоков в CPython, к сожалению, несколько неудобна, что делает ее менее полезной, чем в других языках. Хотя она подходит для некоторых проблем, но не столь хороша, как в C или Java.

В этом разделе мы обсудим ограничения многопоточности в CPython, а также общие параллельные задачи, для которых многопоточность Python будет жизнеспособным решением.

Что такое многопоточность

Под потоком имеется в виду поток выполнения. Программист может разделить свою работу на потоки, которые функционируют одновременно и используют один и тот же контекст памяти. Если ваш код не зависит от сторонних ресурсов, то многопоточность не позволит ускорить работу на одноядерном процессоре и даже добавит лишние затраты ресурсов на управление потоками. Многопоточность хорошо работает на многопроцессорных или многоядерных машинах, где каждый поток может выполняться на отдельном ядре процессора, в результате чего программа выполняется быстрее. Это общее правило, справедливое для большинства языков программирования. В Python увеличение производительности многопоточности на многоядерных процессорах несколько ограниченно, и мы обсудим это. Для простоты предположим, что данное утверждение верно и для Python.

Факт использования одного и того же контекста несколькими потоками означает, что данные нужно защитить от неконтролируемого одновременного доступа. Если два взаимосвязанных потока изменяют одни и те же данные, то это чревато возникновением ситуации, когда малейшее изменение в одном из потоков может неожиданным образом повлиять на конечный результат. Чтобы лучше понять эту проблему, представьте, будто есть два потока, которые увеличивают значение общей переменной:

```
counter_value = shared_counter
shared_counter = counter_value + 1
```

Теперь, предположим, что у переменной `shared_counter` есть начальное значение `0`. Теперь представим, что потоки обрабатывают один и тот же код параллельно, как показано ниже (табл. 15.1).

Таблица 15.1. Параллельная работа потоков

Поток 1		**Поток 2**	
counter_value = shared_counter	#	counter_value = shared_counter	#
counter_value = 0		counter_value = 0	
shared_counter = counter_value + 1	#	shared_counter = counter_value + 1	#
shared_counter = 0 + 1		shared_counter = 0 + 1	

В зависимости от моментов выполнения и доступности контекста может получиться результат 1 или 2. Такая ситуация называется *опасностью гонки* или *состоянием гонки* и часто является причиной серьезных проблем в работе программы.

Механизмы блокировки помогают защищать данные и программировать потоки, поскольку позволяют убедиться, что доступ к ресурсам дается безопасным образом. Но неосторожное использование блокировок тоже может создать проблемы. Самая

большая проблема возникает в случае, когда из-за неправильной организации кода два потока блокируют один ресурс и пытаются выполнить блокировку второго ресурса, уже заблокированного ранее. И все, программа «застынет». Такая ситуация называется *тупиком*, и выйти из нее очень трудно. *Отслеживание повторного обращения* частично решает проблему, поскольку ограничивает попытки заблокировать ресурс дважды.

Тем не менее, когда потоки используются для изолированных задач с помощью специально подготовленных инструментов, их применение позволяет увеличить скорость работы программы.

Многопоточность обычно поддерживается на уровне ядра системы. Когда у машины один процессор с одним ядром, система использует механизм *квантования времени*. В этом случае центральный процессор переключается с одного потока на другой так быстро, что кажется, будто потоки действуют одновременно. Это также делается и на уровне обработки. Параллелизм без нескольких блоков обработки, очевидно, будет лишь виртуальным и никакого роста производительности не даст. Во всяком случае, при написании кода иногда полезно применять потоки, даже если он должен выполняться на одном ядре, и позже мы рассмотрим возможный вариант применения такого выполнения.

Все меняется, когда у среды выполнения имеется несколько процессоров или его ядер. Даже если используется квантование времени, процессы и потоки распределяются между процессорами и программа выполняется быстрее.

Посмотрим на то, как Python работает с потоками.

Как Python работает с потоками

В отличие от ряда других языков в Python используется несколько потоков на уровне ядра, и каждый из них может запустить любой из потоков уровня интерпретатора. Однако в стандартной реализации CPython язык есть ограничение, которое делает потоки менее пригодными для применения в различных контекстах. Все потоки, обращающиеся к объектам Python, работают с одной глобальной блокировкой. Это делается потому, что большая часть внутренних структур интерпретатора, а также сторонний код C небезопасны для потоков и нуждаются в защите.

Этот механизм называется *глобальной блокировкой интерпретатора* (global interpreter lock, GIL), и мы уже обсуждали подробности его реализации на уровне Python/C API в главе 9. Тема об удалении GIL время от времени появляется на Python-Dev и многократно предлагалась разработчикам. К сожалению, на момент написания нашей книги никто не придумал разумного и простого решения, которое позволило бы избавиться от данного ограничения. Весьма маловероятно, что в ближайшее время мы увидим какой-либо прогресс в этой области. Разумнее будет предположить, что GIL останется в CPython, и поэтому нам придется жить с этим.

Итак, что такое многопоточность в Python?

Когда в потоках присутствует только чистый код Python, использовать потоки для ускорения программы не слишком разумно, поскольку GIL будет глобально сериализовать выполнение всех потоков. Однако GIL работает лишь с кодом Python. На практике глобальная блокировка интерпретатора снимается на многих системных вызовах блокировки и может быть снята в расширениях C, в которых не применяются функции Python/C API. Это значит, что несколько потоков могут делать операции ввода/вывода или выполнять код C сторонних расширений параллельно.

Многопоточность позволяет эффективно использовать время, когда программа ждет освобождения стороннего ресурса. Это связано с тем, что бездействующий поток, который явно выпустил GIL, может «проснуться» в момент возврата результатов. Наконец, всякий раз, когда нужна программа, имеющая адаптивный интерфейс, многопоточность будет полезна даже в одноядерных средах, где операционная система сама распределяет время между потоками. За счет многопоточности программа может легко взаимодействовать с пользователем при выполнении некоторых сложных вычислений в так называемом фоновом режиме.

Обратите внимание: GIL есть не в каждой реализации языка Python. Это ограничение присутствует только в CPython, Stackless Python и PyPy, но отсутствует в Jython и IronPython (см. главу 1). Версия PyPy без GIL разрабатывалась, однако на момент написания этой книги все еще находится в экспериментальной стадии и документации к ней нет. Реализация основана на *транзакционной памяти* и называется PyPy-STM. Трудно сказать, когда она будет (и будет ли вообще) выпущена официально. Пока складывается ощущение, что это произойдет нескоро.

В следующем подразделе мы обсудим, когда использовать многопоточность.

Когда использовать многопоточность

Несмотря на ограничения GIL, потоки могут быть очень полезны в таких случаях, как:

- создание адаптивных интерфейсов;
- делегирование работы;
- создание многопользовательских приложений.

Обсудим эти случаи ниже.

Создание адаптивных интерфейсов

Допустим, вы просите систему скопировать файлы из одной папки в другую с помощью какой-то программы с пользовательским интерфейсом. Задача, возможно,

отодвинется на задний план, а окно интерфейса станет постоянно обновляться программой. В этом случае вы получите живую обратную связь о ходе всего процесса, а также сможете при желании отменить операцию. Это раздражает меньше, чем сырые команды оболочки `cp` или `copy`, которые не дают какой-либо обратной связи, пока работа не будет закончена.

Хороший интерфейс также позволяет пользователю работать с несколькими задачами одновременно. Например, Gimp дает возможность обрабатывать изображение, пока что-то другое фильтруется и эти две задачи являются независимыми.

При попытке создать такой отзывчивый интерфейс полезно отодвигать длительные сервисы на задний план или по крайней мере создать постоянную обратную связь с пользователем. Самый простой способ добиться этого — задействовать потоки. В данном случае они будут предназначены уже не для повышения производительности, а для получения гарантий того, что пользователь по-прежнему может работать с интерфейсом, даже если для этого потребуется обработать какие-то данные в течение более длительного периода времени.

Если такие фоновые задачи выполняют множество операций ввода/вывода, то вы можете извлечь пользу из многоядерных процессоров. То есть ситуация получается *беспроигрышной*.

Делегирование работы

Если работа вашего приложения зависит от нескольких внешних ресурсов, то потоки могут реально помочь ускорить ее.

Рассмотрим функцию, которая индексирует файлы в папке и отправляет индексы в базу данных. В зависимости от типа файла функция вызывает какую-то внешнюю программу. Например, это может быть программа для работы с PDF-файлами или документами OpenOffice.

Вместо того чтобы обрабатывать все файлы последовательно, выполняя нужную программу, а затем сохраняя результаты в базу данных, ваша функция может создать отдельный поток для каждого конвертера и отправлять задачи по потокам. Общее время выполнения будет стремиться к времени обработки самого медленного конвертера, а не к сумме времени выполнения всех задач.

Такой подход — своего рода гибрид между многопоточностью и многопроцессорностью. Если вы делегируете работу во внешние процессы (например, с помощью функции `run()` из модуля `subprocess`), то на самом деле делаете работу в нескольких процессах. А в нашем случае мы в основном ждем обработки результатов в отдельных потоках, поэтому с точки зрения кода Python речь идет о многопоточности.

Другой общий случай использования потоков — выполнение нескольких запросов HTTP к внешнему сервису. Например, если вы хотите получить несколько

результатов из удаленного веб-API, то синхронное выполнение может занять много времени, особенно если удаленный сервер расположен далеко. Всегда ожидая предыдущего ответа, прежде чем принимать новые запросы, вы потратите много времени, просто ожидая ответа от внешнего сервиса, и дополнительные временные задержки прибавятся к каждому такому запросу. Если вы общаетесь с каким-нибудь мощным сервисом (Google Maps API, например), то весьма вероятно, что он сможет обслуживать ваши запросы одновременно, не влияя на время отклика отдельных запросов. Тогда будет целесообразно выполнить несколько запросов в отдельных потоках. Помните, что при выполнении запроса HTTP максимальное время тратится на чтение из сокета TCP. Данная операция блокирует ввод/вывод, вследствие чего CPython снимает GIL при выполнении функции C `recv()`. Это позволяет значительно улучшить производительность вашего приложения.

Многопользовательские приложения

Многопоточность также применяется для реализации параллельной обработки в многопользовательских приложениях. Например, веб-сервер отправляет запрос пользователя в новый поток, а затем ожидает новых запросов. Отдельный поток на каждого пользователя упрощает дело, но разработчик будет вынужден подумать о блокировке общих ресурсов. Однако это не будет проблемой, если все общие данные отправляются в реляционную БД, которая берет на себя вопросы многозадачности. Таким образом, потоки в многопользовательском приложении работают почти как отдельные независимые процессы. Они находятся в одном и том же процессе лишь для того, чтобы облегчить управление на уровне приложений.

Например, веб-сервер может поместить все запросы в очередь и ждать, пока освободится поток для отправки. Кроме того, допускается совместное использование памяти, что тоже позволяет ускорить работу и уменьшить нагрузку на память. Два очень популярных Python WSGI-совместимых веб-сервера *Gunicorn* (gunicorn.org) и *uWSGI* (uwsgi-docs.readthedocs.org) позволяют обслуживать HTTP-запросы с несколькими потоками в соответствии с этим принципом.

Применение многопоточности в многопользовательских приложениях влечет меньше затрат, чем мультипроцессорность. Отдельные процессы стоят дороже, поскольку для каждого из них загружается новый интерпретатор. С другой стороны, слишком большое количество потоков — тоже дорого. Мы знаем, что GIL не создает больших проблем для ввода/вывода, но все равно вам рано или поздно нужно будет выполнить код Python. Вы не можете распараллелить все части приложения на отдельные потоки, вследствие чего не получится использовать все ресурсы на машинах с многоядерными процессорами и одним процессом Python.

Именно поэтому оптимальное решение — сочетание многопроцессорной и многопоточной идеи. К счастью, многие из WSGI-совместимых веб-серверов позволяют организовывать все таким образом.

Далее мы рассмотрим пример многопоточного приложения.

Пример многопоточного приложения

Чтобы увидеть, как многопоточность в Python работает на практике, сделаем несколько примеров приложений, которые за счет многопоточности будут функционировать лучше. Мы обсудим простую задачу, с которой вы еще не раз столкнетесь в процессе создания параллельных запросов HTTP. Эта задача уже упоминалась в качестве распространенного варианта использования многопоточности.

Допустим, нам нужно извлечь данные из ряда веб-сервисов с помощью нескольких запросов, которые нельзя объединить в один большой HTTP-запрос. В качестве реального примера будем использовать курсы обмена валют с *Foreign exchange rates API* по адресу exchangeratesapi.io. Причины такого выбора:

- это открытый сервис, не требующий каких-либо ключей аутентификации;
- API этого сервиса очень прост, и к нему легко обращаться с помощью популярной библиотеки `requests`;
- код для этого API открытый и написан на Python. Это значит, что если официальный сервис «упадет», то вы сможете скачать исходный код из официального репозитория на GitHub по ссылке github.com/exchangeratesapi/exchangeratesapi.

В наших примерах мы попытаемся получить курсы выбранных валют, используя несколько валют в качестве опорных точек. Результаты оформим в виде матрицы обменного курса, как показано ниже:

```
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
```

В выбранном API есть несколько способов формирования запроса нескольких точек данных в пределах одного запроса, но, к сожалению, нет способа делать запрос данных с помощью нескольких базовых валют одновременно. Получить курс для одного основания проще простого:

```
>>> import requests
>>> response =
requests.get("https://api.exchangeratesapi.io/latest?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'}
```

Поскольку наша цель — показать, какую по времени выгоду многопоточное решение параллельных задач дает по сравнению со стандартным подходом, мы начнем с реализации вообще без многопоточности. Ниже приведен код программы, которая перебирает список базовых валют, делает запрос к API и отображает результаты в стандартный вывод в виде 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.exchangeratesapi.io/latest?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]
    # Примечание: валюта обменивается сама на себя с коэффициентом 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("time elapsed: {:.2f}s".format(elapsed))
```

Вокруг выполнения основной функции `main()` мы добавили несколько операторов, которые измеряют, сколько времени потребовалось на выполнение работы. На нашем компьютере на эту задачу требуется даже больше одной секунды:

```
$ python3 synchronous.py
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
time elapsed: 1.13s
```

Каждый запуск нашего скрипта всегда будет занимать разное время, поскольку оно в основном зависит от удаленного сервиса, доступного через сеть. Поэтому на конечный результат влияет сразу много недетерминированных факторов. Лучше всего сделать много тестов, а затем вычислить среднее значение на основе измерений. Но для простоты мы не будем так делать. Чуть позже вы увидите, что этого упрощенного подхода достаточно для демонстрации.

В следующем подпункте поговорим об использовании отдельного потока для каждого элемента.

Использование отдельного потока для каждого элемента. Настало время для усовершенствований. Мы не делаем в Python никаких крупных вычислений, и длительное время выполнения вызвано исключительно связью с внешним сервисом. Мы посылаем HTTP-запрос на удаленный сервер, он вычисляет ответ, а затем ждем, пока ответ придет обратно. В данном процессе участвует много операций ввода/вывода, поэтому многопоточность кажется жизнеспособным вариантом. Мы можем сразу делать все запросы в отдельных потоках, а затем просто ждать, пока получим результаты их всех. Если сервис, к которому мы обращаемся, может обрабатывать наши запросы одновременно, то мы определенно увидим улучшение производительности.

Начнем с самого простого подхода. В Python есть чистая и простая в использовании абстракция системы потоков в модуле `threading`. Ядро этой стандартной библиотеки составляет класс `Thread`, который представляет собой один экземпляр потока. Ниже описана модифицированная версия функции `main()`, которая создает и запускает новый поток для каждого места, а затем ждет, пока все потоки не закончат выполнение:

```
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 быстрое и грубое решение, и задача решается спустя рукава. Написать надежную программу, которая будет обслуживать тысячи или миллионы пользователей, таким образом не получится. В данном подходе есть серьезные проблемы, с которыми можно разобраться позже. Но он все же работает, как мы можем видеть из следующего кода:

```
$ python3 threads_one_per_item.py
1 CZK =   0.044 USD,   0.039 EUR,   0.167 PLN,   0.375 NOK,     1.0 CZK
1 NOK =   0.117 USD,   0.104 EUR,   0.446 PLN,     1.0 NOK,    2.66 CZK
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
time elapsed: 0.13s
```

Причем работает значительно быстрее.

Таким образом, мы знаем, что потоки полезны для нашего приложения, и пришло время начать применять их более разумно. Для начала необходимо определить следующие проблемы в показанном выше коде.

- Мы заводим новый поток для каждого параметра. Инициализация потока тоже занимает некоторое время, но расходы эти незначительны, а проблема не единственная. Вдобавок потоки потребляют другие ресурсы, такие как память или дескрипторы файлов. В нашем примере строго определенное количество элементов, но если бы их было сколько угодно? Вы точно не хотите запускать непонятно сколько потоков в зависимости от произвольного размера входных данных.
- Функция `fetch_rates()`, которая выполняется в вызовах потоков, вызывает встроенную функцию `print()`, и на практике очень маловероятно, что вы захотите делать это за пределами основного потока приложения. Дело в основном в том, как стандартный вывод в Python буферизуется. Вывод может оказаться некорректным, когда несколько вызовов данной функции будут «перескакивать» между потоками. Кроме того, функция `print()` считается медленной. Если ее безрассудно использовать в нескольких потоках, то это может привести к сериализации, что сведет на нет все преимущества многопоточности.
- И последнее, но не менее важное: делегируя каждый вызов функции в отдельный поток, мы усложняем задачу по контролированию скорости, с которой обрабатывается наш ввод. Да, мы хотим выполнить работу максимально быстро, но очень часто внешние сервисы накладывают жесткие ограничения на частоту

запросов от одного клиента. Иногда целесообразно разработать программу так, чтобы можно было регулировать скорость обработки во избежание попадания вашего приложения в черный список внешних интерфейсов API.

А теперь рассмотрим, как использовать пул потоков.

Использование пула потоков. Первая задача, которую мы постараемся решить, — неограниченность количества потоков, создающихся нашей программой. Хорошо бы создать пул потоковых работников строго определенного размера, который будет обрабатывать всю параллельную работу и общаться с работниками через некие потокобезопасные структуры данных. Подход *«пул потоков»* позволит решить две проблемы, которые мы упоминали выше.

Итак, общая идея заключается в том, чтобы запустить некое предопределенное количество потоков, которые станут разбирать задачи из очереди, пока она не опустеет. Когда задачи кончатся, потоки будут возвращаться и мы сможем выйти из программы. Хорошим кандидатом для создания такой структуры будет класс `Queue` из встроенного модуля `queue`. Это реализация очереди *first in first out (FIFO)*, очень похожая на `deque`-коллекцию из модуля `collections` и специально разработанная для обработки межпоточной связи. Ниже приведена модифицированная версия функции `main()`, которая запускает ограниченное количество потоков с помощью функции `worker()` в качестве цели и взаимодействует с ними, используя потокобезопасную очередь:

```
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.exchangeratesapi.io/latest?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]

    # Примечание: валюта обменивается сама на себя с коэффициентом 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(block=False)
        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("time elapsed: {:.2f}s".format(elapsed))
```

Результат запуска модифицированной версии нашей программы аналогичен предыдущему:

```
$ python3 threads_thread_pool.py
1 EUR =    1.13 USD,     1.0 EUR,    4.29 PLN,    9.62 NOK,    25.6 CZK
1 NOK =   0.117 USD,   0.104 EUR,   0.446 PLN,     1.0 NOK,    2.66 CZK
1 USD =     1.0 USD,   0.887 EUR,     3.8 PLN,    8.53 NOK,    22.7 CZK
1 PLN =   0.263 USD,   0.233 EUR,     1.0 PLN,    2.24 NOK,    5.98 CZK
1 CZK =   0.044 USD,   0.039 EUR,   0.167 PLN,   0.375 NOK,     1.0 CZK

time elapsed: 0.17s
```

Общее время выполнения может оказаться больше, чем в предыдущем случае, но зато теперь невозможно исчерпать все вычислительные ресурсы из-за про-

извольной длины входных данных. Кроме того, мы можем настроить параметр `THREAD_POOL_SIZE`, чтобы найти компромисс между затратами ресурсов и временем.

А теперь рассмотрим, как использовать двусторонние очереди.

Использование двусторонних очередей. Теперь можно решить проблему вывода данных при использовании нескольких потоков. Желательно переложить ответственность за вывод на основной поток, запускавший рабочие потоки. Для этого мы можем создать другую очередь, которая будет отвечать за сбор результатов от наших работников. Ниже приведен код, в котором все собрано воедино, а 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.exchangeratesapi.io/latest?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]

    # Примечание: валюта обменивается сама на себя с коэффициентом 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(block=False)
        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("time elapsed: {:.2f}s".format(elapsed))
```

Это исключает риск появления некорректных входных данных, который мог бы возникнуть, если бы функция `present_result()` выполняла больше операторов `print()` или некие дополнительные вычисления. Мы не ожидаем каких-либо улучшений производительности от такого подхода при небольших объемах входных данных, но зато снизили риск сериализации потока из-за медленного выполнения `print()`. Ниже приведен окончательный результат:

```
$ python3 threads_two_way_queues.py
1 USD =     1.0 USD,   0.887 EUR,     3.8 PLN,    8.53 NOK,    22.7 CZK
1 PLN =   0.263 USD,   0.233 EUR,     1.0 PLN,    2.24 NOK,    5.98 CZK
1 EUR =    1.13 USD,     1.0 EUR,    4.29 PLN,    9.62 NOK,    25.6 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

time elapsed: 0.17s
```

Теперь посмотрим, как работать с ошибками и ограничениями скорости.

Работа с ошибками и ограничениями скорости. Наконец, у вас могут возникнуть проблемы при работе с ограничениями скорости, установленными сторон-

ними поставщиками услуг. В случае API валютных курсов сервис не сообщал нам о каких-либо ограничениях скорости или механизмах дросселирования. Но многие сервисы (даже платные) часто накладывают ограничения скорости. И уж тем более несправедливо ругать за это сервис, который предоставляется пользователям полностью бесплатно.

При использовании нескольких потоков очень легко исчерпать любое ограничение скорости, если сервис не ограничивает сами запросы, — для этого нужно нагрузить сервис до уровня, когда он уже не сможет отвечать на запросы. Проблема становится еще более серьезной из-за того, что мы пока не рассматривали сценарии отказа, а работа с исключениями в многопоточном коде Python немного сложнее, чем обычно.

Функция `request.raise_for_status()` выбросит исключение и получит код статуса или тип ошибки (например, ограничение скорости), что для нас хорошо. Это исключение выбрасывается в отдельном потоке и не приводит к падению всей программы. Рабочий поток, конечно же, тотчас завершится, но основной поток будет ждать выполнения всех задач из `work_queue` до конца (с вызовом `work_queue.join()`). Теперь мы можем оказаться в ситуации, когда некоторые рабочие потоки будут выходить из строя и программа так и не дождется завершения. Это значит, наши потоки должны корректно обрабатывать возможные исключения и проверять, все ли элементы из очереди обрабатываются.

Внесем в код незначительные изменения, позволяющие подготовиться к любым потенциальным проблемам. В случае исключений в рабочем потоке мы можем поместить экземпляр ошибки в очередь `results_queue` и пометить текущую задачу выполненной, как если бы ошибок не было. Мы будем уверены, что основной поток не станет до бесконечности ожидать `work_queue.join()`. Затем основной поток проверяет результаты и вновь выбрасывает исключения, найденные в очереди результатов. Ниже приведены улучшенные версии функций `worker()` и `main()`, которые будут безопасно обрабатывать исключения (изменения выделены):

```
def worker(work_queue, results_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get(block=False)
        except Empty:
            break
        else:
            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)
```

Когда мы научились нормально обрабатывать исключения, настало время немного повредить наш код. Мы не хотим злоупотреблять нашим бесплатным API и вызывать отказ в обслуживании. Вместо того чтобы чересчур нагружать API, мы смоделируем типичную ситуацию, которая является результатом многих механизмов дросселирования. Многие API возвращают ошибку `429 Too Many Requests`, если клиент превышает допустимый предел скорости. Итак, мы обновим функцию `fetch_rates()` с целью переопределить код состояния каждые несколько ответов таким образом, чтобы выбросить исключение. Ниже показана обновленная версия функции, которая имитирует HTTP-ошибки через каждые несколько запросов:

```
def fetch_rates(base):
    response = requests.get(
        f"https://api.exchangeratesapi.io/latest?base={base}"
    )
    if random.randint(0, 5) < 1:
        # Имитируем ошибку, переопределяя код состояния
        response.status_code = 429

    response.raise_for_status()
    rates = response.json()["rates"]
    # Примечание: валюта обменивается сама на себя с коэффициентом 1:1
    rates[base] = 1.
    return base, rates
```

Если вы используете эту функцию в вашем коде, то получите примерно такую ошибку:

```
$ python3 threads_exceptions_and_throttling.py
1 PLN =   0.263 USD,   0.233 EUR,     1.0 PLN,    2.24 NOK,    5.98 CZK
1 EUR =    1.13 USD,     1.0 EUR,    4.29 PLN,    9.62 NOK,    25.6 CZK
1 USD =     1.0 USD,   0.887 EUR,     3.8 PLN,    8.53 NOK,    22.7 CZK
Traceback (most recent call last):
  File "threads_exceptions_and_throttling.py", line 136, in <module>
    main()
  File "threads_exceptions_and_throttling.py", line 129, in main
    raise result
  File "threads_exceptions_and_throttling.py", line 96, in worker
    result = fetch_rates(item)
  File "threads_exceptions_and_throttling.py", line 70, in fetch_rates
    response.raise_for_status()
  File "/usr/local/lib/python3.7/site-packages/requests/models.py", line 940, in
raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 429 Client Error: OK for url:
https://api.exchangeratesapi.io/latest?base=NOK
```

Забудем о воображаемой ошибке и сделаем вид, что это исключение не является результатом ошибочного кода. В такой ситуации наша программа будет работать слишком быстро для бесплатного сервиса. Она делает слишком много одновременных запросов, и для правильной работы нам нужна возможность ограничить скорость программы.

Ограничение скорости работы часто называют дросселированием. На PyPI есть несколько пакетов, которые позволяют ограничить скорость выполнения, и пакеты эти очень просты в использовании. Однако сейчас мы не станем задействовать сторонний код. Дросселирование — отличная возможность хорошо ввести некоторые механизмы блокировки для потоков, поэтому попробуем создать решение дросселирования с нуля.

Алгоритм, который мы будем использовать, иногда называют корзиной с маркерами (токенами) (Token Bucket). Работает он очень просто и включает следующие этапы.

1. Есть корзина с заранее определенным количеством маркеров.
2. Каждый маркер соответствует одному разрешению на обработку одного элемента задач.
3. Каждый раз, когда работник запрашивает один или нескольких маркеров (разрешений), мы делаем следующее:
   - измеряем, сколько времени было потрачено с последнего раза, когда мы заполняли корзину;
   - если позволяет разница во времени, то снова наполняем корзину маркерами в количестве, соответствующем этой разнице во времени;

- если количество маркеров больше или равно запрошенной сумме, то уменьшаем количество хранимых маркеров и возвращаем это значение;
- если количество хранимых маркеров меньше, чем требуется, то возвращаемся к нулю.

Важно всегда инициализировать корзину с нулевым количеством маркеров и не позволять ей переполняться. Если мы не будем соблюдать эти меры предосторожности, то появятся лишние маркеры, превышающие лимит скорости. В нашей ситуации данный лимит выражается в запросах в секунду, и нам не придется иметь дело с произвольными квантами времени. Мы предполагаем, что измерение базируется на одной секунде, поэтому не будем хранить больше маркеров, чем количество запросов, разрешенных для этого периода времени. Ниже представлен пример реализации класса, который выполняет дросселирование с помощью алгоритма Token Bucket:

```
From threading import Lock

class Throttle:
    def __init__(self, rate):
        self._consume_lock = Lock()
        self.rate = rate
        self.tokens = 0
        self.last = 0

    def consume(self, amount=1):
        with self._consume_lock:
            now = time.time()
            # Сначала инициализируется измерение времени.
            # Запрос маркера, чтобы избежать внутренних проблем.
            if self.last == 0:
                self.last = now

            elapsed = now - self.last

            # Убедимся, что прошло достаточно много
            # времени для выдачи нового маркера
            if int(elapsed * self.rate):
                self.tokens += int(elapsed * self.rate)
                self.last = now

            # Запрет переполнения
            self.tokens = (
                self.rate
                if self.tokens > self.rate
                else self.tokens
            )

            # Если маркеры в наличии, то они выдаются
            if self.tokens >= amount:
                self.tokens -= amount
```

```
            else:
                amount = 0

            return amount
```

Использовать этот класс очень просто. Предположим, мы создали только один экземпляр `Throttle` (например, `Throttle(10)`) в основном потоке и передали каждый рабочий поток в качестве позиционного аргумента. Задействовать ту же структуру данных в различных потоках безопасно, поскольку мы защитили изменения внутреннего состояния экземпляром класса `Lock` из модуля `threading`. Теперь мы можем обновить реализацию функции `worker()`, чтобы она ожидала все элементы, пока `Throttle` не выпустит новый маркер, как показано ниже:

```
def worker(work_queue, results_queue, throttle):
    while True:
        try:
            item = work_queue.get(block=False)
        except Empty:
            break
        else:
            while not throttle.consume():
                pass

            try:
                result = fetch_rates(item)
            except Exception as err:
                results_queue.put(err)
            else:
                results_queue.put(result)
            finally:
                work_queue.task_done()
```

В следующем разделе посмотрим на другую модель многозадачности.

Многопроцессорная обработка

Будем честны — многопоточность сложновата в реализации, как мы уже видели в предыдущем разделе. Но самый простой подход к проблеме должен требовать минимум усилий, а нормальная реализация потоков требует огромного количества кода.

Нам пришлось организовать пул потоков, очереди связи, корректно обрабатывать исключения из потоков, а также подумать о безопасности потоков при попытке превысить ограничение скорости. Параллельное выполнение всего одной функции из некой внешней библиотеки потребовало десятков строк кода! И мы лишь предполагаем, что все сделали правильно, поскольку разработчик внешнего пакета заявил о потокобезопасности библиотеки. Похоже, цена высока за решение, которое практически применимо только для выполнения задач ввода/вывода.

Альтернативный подход, позволяющий реализовать параллельность, — многопроцессорность. Отдельные процессы Python, которые не ограничивают друг друга через GIL, дают возможность более эффективно использовать ресурсы. Это особенно важно для приложений, работающих на многоядерных процессорах, выполняющих очень ресурсоемкие задачи. На данный момент это единственное встроенное решение для разработчиков на Python (с интерпретатором CPython), которое позволяет извлечь пользу из нескольких ядер процессора в любой ситуации.

Другое преимущество применения нескольких процессов заключается в том, что они не разделяют контекст памяти. За счет этого становится труднее испортить данные и ввести в приложение тупиковые ситуации. Общий контекст памяти означает, что требуется самостоятельно разделять данные между отдельными процессами, но, к счастью, существует много готовых способов реализации надежного межпроцессного взаимодействия. В Python есть готовые примитивы, которые позволяют наладить связь между процессами почти так же легко, как и между потоками.

Самый простой способ запускать новые процессы на любом языке программирования — в какой-то момент прибегнуть к *разветвлению* программы. В системах POSIX (UNIX, macOS и Linux) такое разветвление, или вилка, представляет собой системный вызов, который передается в Python с помощью функции `os.fork()` и создает новый дочерний процесс. Эти два процесса затем работают сами по себе. Ниже приведен пример скрипта, в котором разветвление вызывается 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: hey, I am the child process")
        print("CHLD: all the pids i know %s" % pid_list)

    else:
        pid_list.append(os.getpid())
        print()
        print("PRNT: hey, I am the parent")
        print("PRNT: the child is pid %d" % child_pid)
        print("PRNT: all the pids i know %s" % pid_list)

if __name__ == "__main__":
    main()
```

Результат в консоли:

```
$ python3 forks.py
PRNT: hey, I am the parent
PRNT: the child is pid 21916
PRNT: all the pids i know [21915, 21915]
CHLD: hey, I am the child process
CHLD: all the pids i know [21915, 21916]
```

Обратите внимание: оба процесса имеют одинаковое исходное состояние своих данных перед вызовом `os.fork()`. У обоих один и тот же номер PID (идентификатор процесса) в качестве первого значения коллекции `pid_list`. Затем эти состояния расходятся, и мы видим, что дочерний процесс добавляет значение `21916`, а родительский дублирует свой PID `21915`. Это связано с тем, что контексты памяти этих двух процессов не являются общими. Они имеют одинаковые начальные условия, но не могут влиять друг на друга после вызова `os.fork()`.

После того как контекст копируется в дочерний процесс, каждый процесс работает с собственным адресным пространством. Чтобы реализовать связь, процессы должны работать с общесистемными ресурсами или средствами, например с *сигналами*.

К сожалению, `os.fork` недоступен для Windows, где для разветвления нужно запускать новый интерпретатор. Таким образом, реализация будет зависеть от платформы. Кроме того, в модуле `os` есть функции, которые позволяют порождать новые процессы под Windows, но их вы будете использовать редко. То же самое верно и для `os.fork()`. В Python есть большой модуль многопроцессорной обработки, который создает интерфейс высокого уровня.

Огромное преимущество данного модуля состоит в том, что в нем есть абстракции, которые мы написали с нуля в примере многопоточного приложения. Это позволяет ограничить количество шаблонного кода, улучшает удобство сопровождения приложения и уменьшает его сложность. Удивительно, но, несмотря на свое название, модуль `multiprocessing` предоставляет интерфейс, используемый для потоков, поэтому вы сможете применять один и тот же интерфейс для обоих подходов.

В следующем подразделе рассмотрим встроенный модуль `multiprocessing`.

Встроенный модуль multiprocessing

Модуль `multiprocessing` — портативный инструмент для работы с процессами, как если бы они были потоками.

В модуле есть класс `Process`, который очень похож на класс `Thread` и может быть использован на любой платформе следующим образом:

```
from multiprocessing import Process
import os
```

```
def work(identifier):
    print(
        'hey, i am a process {}, pid: {}'
        ''.format(identifier, 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()
```

Этот скрипт, будучи выполненным, дает следующий результат:

```
$ python3 processing.py
hey, i am a process 1, pid: 9196
hey, i am a process 0, pid: 8356
hey, i am a process 3, pid: 9524
hey, i am a process 2, pid: 3456
hey, i am a process 4, pid: 6576
```

Когда создаются процессы, память раздваивается (в системах POSIX). Наиболее эффективный способ применять процессы — позволить им работать после того, как они были созданы, чтобы избежать лишних затрат, а также проверить их состояния со стороны родительского процесса. Помимо копирования контекста памяти, класс `Process` также добавляет дополнительный аргумент `args` в конструктор, и данные тоже можно передавать.

Налаживание связи между модулями процесса требует дополнительных трудозатрат, поскольку их локальная память по умолчанию не общая. Чтобы облегчить работу, модуль предоставляет следующие несколько способов общения между процессами:

- с помощью класса `multiprocessing.Queue` — функциональный клон `queue.Queue`, который мы ранее использовали для связи между потоками;
- с помощью модуля `multiprocessing.Pipe`, который представляет собой двусторонний канал связи;
- с помощью модуля `multiprocessing.sharedctypes`, позволяющего создавать произвольные типы C (из модуля `ctypes`) в выделенном пуле памяти, которая распределяется между процессами.

Классы `multiprocessing.Queue` и `queue.Queue` имеют один и тот же интерфейс. Единственное их различие состоит в том, что первый предназначен для

применения в разных средах процесса, а не с несколькими потоками, вследствие чего в нем используются различные внутренние транспортные линии и блокирующие примитивы. Мы уже видели, как задействовать очереди с многопоточностью, в пункте «Пример многопоточного приложения» предыдущего раздела, поэтому не будем делать то же самое для многопроцессорного варианта. Использование остается неизменным, следовательно, такой пример не принесет ничего нового.

Более интересная картина возникает при работе с классом `Pipe`. Это дуплексный (двусторонний) канал, похожий по своей концепции на процессы UNIX. Интерфейс `Pipe` очень похож на простой сокет из встроенного модуля `socket`. Разница заключается в том, что он позволяет передавать любой pickable-объект (с помощью модуля `pickle`) вместо сырых байтов.

Это облегчает организацию связи между процессами, поскольку вы можете отправить практически любой базовый тип Python, как показано ниже:

```
from multiprocessing import Process, Pipe

class CustomClass:
    pass

def work(connection):
    while True:
        instance = connection.recv()

        if instance:
            print("CHLD: {}".format(instance))

        else:
            return

def main():
    parent_conn, child_conn = Pipe()

    child = Process(target=work, 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()
```

Взглянув на результат вывода этого скрипта, вы увидите, что можете легко передавать экземпляры пользовательского класса 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>
```

Другой способ реализовать общее состояние между процессами заключается в использовании сырых типов в общем пуле памяти с классами из `multiprocessing.sharedctypes`. Основные из них — `Value` и `Array`. Ниже приведено несколько примеров 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[:])
```

Результатом будет следующий вывод:

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

При работе с `multiprocessing.sharedctypes` следует помнить: вы работаете с общей памятью, поэтому во избежание риска повреждения данных необходимо использовать блокирующие примитивы. В модуле есть классы, аналогичные тем, которые применяются в модуле `threading`, такие как `Lock`, `Rlock` и `Semaphore`. Недостаток классов из `sharedctypes` заключается в том, что они позволяют разделять только основные типы C из модуля `ctypes`. Если вам нужно передать более

сложные структуры или экземпляры классов, то следует задействовать `Queue`, `Pipe` или другие каналы связи между процессами. В большинстве случаев разумно избегать использования типов из `sharedctypes`, поскольку они увеличивают сложность кода и влекут за собой все опасности и проблемы многопоточности.

Далее посмотрим, как применять пул процессов.

Использование пула процессов

Применение нескольких процессов вместо потоков значительно увеличивает вычислительные затраты. Главным образом это касается объема памяти, поскольку каждый процесс имеет собственный и независимый контекст памяти. То есть создать неопределенное количество дочерних процессов будет еще более проблематично, чем в многопоточных приложениях.

Наилучший шаблон управления применением ресурсов в многопроцессорных приложениях — это создание пула процессов, как мы уже делали в подпункте «Использование пула потоков» на с. 473.

А самое лучшее в модуле `multiprocessing` — наличие готового к использованию класса `Pool`, который берет на себя все сложности управления несколькими работниками. Эта реализация пула значительно уменьшает количество шаблонного кода, а также количество вопросов, связанных с двусторонней связью. Вам даже не придется использовать метод `join()` вручную, поскольку пул можно применить в качестве менеджера контекста (оператор `with`). Ниже приведен один из наших предыдущих примеров, переписанных для использования 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.exchangeratesapi.io/latest?base={base}"
    )

    response.raise_for_status()
    rates = response.json()["rates"]
    # Примечание: валюта обменивается сама на себя с коэффициентом 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("time elapsed: {:.2f}s".format(elapsed))
```

Как видите, код значительно сократился. Это значит, что его будет легче сопровождать и отлаживать в случае проблем. В данный момент в коде всего две строки кода отвечают за многопроцессорную обработку. Это большой шаг вперед по сравнению с ситуацией, когда приходилось создавать пул с нуля. Теперь нам даже не понадобится думать о каналах связи, поскольку они создаются неявно внутри реализации класса `Pool`.

Посмотрим, как использовать `multiprocessing.dummy` в качестве интерфейса многопоточности.

Использование multiprocessing.dummy в качестве интерфейса многопоточности

Абстракции высокого уровня из модуля `multiprocessing`, такие как класс `Pool`, дают большое преимущество по сравнению с простыми инструментами из модуля `threading`. Но это не значит, что многопроцессорность всегда лучше, чем многопоточность. Существует большое количество случаев использования, в которых потоки могут оказаться лучше, чем процессы. Это особенно верно для ситуаций, когда требуется добиться малой задержки и/или высокой эффективности применения ресурсов.

Тем не менее это не значит, что придется выбросить все полезные абстракции модуля `multiprocessing`, если нужно задействовать потоки вместо процессов. Существует модуль `multiprocessing.dummy`, который повторяет API `multiprocessing`, но применяет несколько потоков вместо разветвления/порождения новых процессов.

Это позволяет уменьшить количество шаблонного кода, а также делает код более портативным. Еще раз посмотрим на нашу функцию `main()` из предыдущих примеров. Мы могли бы дать пользователю возможность выбрать вариант применения (процессы или потоки), для этого нужно лишь заменить класс конструктора следующим образом:

```
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)
```

В следующем разделе поговорим об асинхронном программировании.

Асинхронное программирование

Асинхронное программирование за последние несколько лет стало очень популярным. В Python 3.5 наконец появились некоторые особенности синтаксиса, укрепившие концепцию асинхронного выполнения. Но это вовсе не означает, что асинхронное программирование стало возможно только начиная с версии Python 3.5. Еще до него существовало много библиотек и фреймворков, и большинство из них уходят корнями в старые версии Python 2. Существует даже целая альтернативная реализация Python под названием Stackless (см. главу 1), сосредоточенная именно на данном подходе к программированию. Некоторые из этих решений, такие как Twisted, Tornado и Eventlet, по-прежнему пользуются популярностью и стоят изучения. Во всяком случае, начиная с Python 3.5, асинхронное программирование стало проще, чем когда-либо прежде. Можно ожидать, что его встроенные асинхронные функции заменят большую часть старых инструментов или внешних проектов и постепенно превратятся в фреймворк высокого уровня, основанный на встроенных модулях Python.

Если попробовать объяснить, что такое асинхронное программирование, то проще всего представить его в виде потока без системы планирования. Это значит, что асинхронная программа может одновременно обрабатывать задачи, но его контекст переключается внутри, а не системным планировщиком.

Разумеется, в асинхронной программе мы не будем использовать потоки, чтобы обрабатывать задачи одновременно. В большинстве решений применяются свои концепции и, в зависимости от реализации, называются по-разному. Ниже приведены некоторые примеры имен, служащих для описания таких параллельных программных элементов:

- зеленые потоки, или гринлеты (greenlet, gevent или eventlet);
- сопрограммы (асинхронное программирование Python 3.5);
- тасклеты (Stackless Python).

Как правило, это одна и та же концепция, которая реализуется в несколько различных видах.

По понятным причинам в данном разделе мы сосредоточимся только на сопрограммах, которые сразу поддерживаются в Python, начиная с версии 3.5.

Кооперативная многозадачность и асинхронный ввод/вывод

Кооперативная многозадачность лежит в основе асинхронного программирования. В этом стиле многозадачности операционная система не занимается инициализацией переключения контекста (на другой процесс или поток). Вместо этого каждый процесс добровольно отдает управление, когда находится в режиме ожидания, чтобы обеспечить одновременное выполнение нескольких программ. В этом смысл слова *«кооперативный»*. Все процессы должны сотрудничать в целях многозадачности.

Данная модель многозадачности когда-то использовалась в операционных системах, но сегодня уже редко применяется на системном уровне. Это связано с имеющейся опасностью того, что один плохо спроектированный сервис может легко поставить под угрозу стабильность системы в целом. В современном виде планирование процессов и потоков вместе с контекстом переключается непосредственно операционной системой, и так реализуется параллелизм на уровне системы. Однако кооперативная многозадачность — отличный инструмент параллелизма на уровне приложений.

При реализации кооперативной многозадачности на уровне приложений мы не работаем с потоками или процессами, которые должны отпускать управление, поскольку все выполнение содержится в одном процессе или потоке. Вместо этого у нас есть несколько задач (сопрограммы, тасклеты или гринлеты), отдающих управление одной функции, которая обрабатывает согласование задач. Как правило, она представляет собой своего рода цикл событий.

Чтобы избежать путаницы в будущем (из-за терминологии Python), с этого момента мы станем называть такие параллельные задачи *сопрограммами*. Наиболее важный вопрос кооперативной многозадачности состоит в том, когда отпускать

управление. В большинстве асинхронных приложений управление передается планировщику или циклу событий в операциях ввода/вывода. Не имеет значения, считывает программа данные из файловой системы или обменивается через порт, поскольку такие операции ввода/вывода всегда создают некое время ожидания при простое процесса. Время ожидания зависит от внешнего источника и дает хорошую возможность освободить управление так, чтобы другие сопрограммы могли выполнять свою работу, если не находятся в ожидании.

Это делает такой подход похожим на то, как в Python реализована многопоточность. Мы знаем, что GIL сериализует потоки Python, но отпускается при каждой операции ввода/вывода. Основное отличие таково: потоки в Python реализованы в виде потоков системного уровня, поэтому операционная система может выгрузить работающий в данный момент поток и передать управление другому потоку в любой момент времени. В асинхронном программировании задачи никогда не вытесняются основным циклом обработки событий. Именно поэтому данный стиль многозадачности также называется *невытесняющей многозадачностью*.

Конечно, каждое приложение Python работает на некой операционной системе, в которой существуют и другие процессы, конкурирующие за ресурсы. Это значит, что операционная система всегда имеет право выгрузить весь процесс и передать управление другому. Но, возвращаясь к работе, наше асинхронное приложение продолжает с того же места, на котором остановилось, когда вмешался системный планировщик. Поэтому сопрограммы по-прежнему считаются невытесняющими.

В следующем подразделе мы рассмотрим ключевые слова `async` и `await`.

Ключевые слова async и await

Ключевые слова `async` и `await` — основные строительные блоки асинхронного программирования Python.

Ключевое слово `async`, будучи использованным перед оператором `def`, определяет новую подпрограмму. Выполнение сопрограммы может быть приостановлено и возобновлено в строго определенных обстоятельствах. Ее синтаксис и поведение очень похожи на генераторы (см. главу 3). Фактически в старых версиях Python нужно применять именно генераторы, когда вы хотите реализовать сопрограммы. Вот пример объявления функции, которая использует ключевое слово `async`:

```
async def async_hello():
    print("hello, world!")
```

Функции, определенные с ключевым словом `async`, особенные. После вызова они не выполняют код внутри, но вместо этого возвращают объект сопрограммы, например:

```
>>> async def async_hello():
...     print("hello, world!")
...
```

```
>>> async_hello()
<coroutine object async_hello at 0x1014129e8>
```

Объект сопрограммы ничего не делает, пока его выполнение не планируется в цикле обработки событий. Модуль `asyncio` дает базовую реализацию цикла событий, а также включает много других асинхронных утилит, как показано ниже:

```
>>> import asyncio
>>> async def async_hello():
...     print("hello, world!")
...
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(async_hello())
hello, world!
>>> loop.close()
```

Очевидно, поскольку мы создали всего одну простую сопрограмму, параллелизма в нашей программе нет. Чтобы увидеть реальную одновременную обработку, нужно создать несколько задач, которые будут выполняться в цикле обработки событий.

Новые задачи добавляются в цикл с помощью вызова метода `loop.create_task()` или путем предоставления другого объекта с помощью функции `asyncio.wait()`. Мы будем использовать последний подход и попытаемся асинхронно напечатать последовательность чисел, которая была сгенерирована с помощью функции `range()`, следующим образом:

```
import asyncio

async def print_number(number):
    print(number)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()

    loop.run_until_complete(
        asyncio.wait([
            print_number(number)
            for number in range(10)
        ])
    )
    loop.close()
```

Функция `asyncio.wait()` принимает список объектов сопрограмм и сразу же возвращается. Результатом является генератор, выдающий объекты с будущими результатами (так называемые *фьючерсы*). Как следует из названия, он ждет, пока не выполнятся все отправленные ему сопрограммы. Возврат генератора вместо объекта сопрограммы делается ради обратной совместимости с предыдущими версиями Python, о которой мы поговорим чуть позже. Результат выполнения этого скрипта может выглядеть так:

```
$ python asyncprint.py
0
7
8
3
9
4
1
5
2
6
```

Как видно, цифры печатаются не в том порядке, в каком мы их создали для наших сопрограмм. Но именно этого мы и хотели добиться.

Второе важное ключевое слово, которое было добавлено в Python 3.5, — это `await`. Оно используется для ожидания результата выполнения сопрограммы или фьючерса (описано ниже) и передачи управления циклу обработки событий. Чтобы лучше понять, как это работает, мы должны рассмотреть более сложный пример кода.

Допустим, мы хотим создать две такие сопрограммы, которые будут выполнять в цикле простые задачи:

- ждать случайное количество секунд;
- выводить некий текст, предоставленный в качестве аргумента, а также количество времени, проведенное в режиме ожидания.

Начнем со следующей простой реализации, имеющей проблемы параллелизма, которые мы позже попытаемся устранить с помощью `await`:

```
import time
import random
import asyncio

async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        time.sleep(time_to_sleep)
        print(
            "{} waited {} seconds"
            "".format(name, time_to_sleep)
        )

async def main():
    await asyncio.wait([waiter("first"), waiter("second")])

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    loop.close()
```

При выполнении в консоли (с помощью команды `time` для измерения времени) мы получим следующий вывод:

```
$ time python corowait.py
second waited 0.25 seconds
second waited 0.25 seconds
second waited 0.5 seconds
second waited 0.5 seconds
first waited 0.75 seconds
first waited 0.75 seconds
first waited 0.25 seconds
first waited 0.25 seconds
real  0m3.734s
user  0m0.153s
sys   0m0.028s
```

Как видите, обе сопрограммы завершили выполнение, но не в асинхронном режиме. Причина в том, что в обеих из них используется функция `time.sleep()`, которая блокирует, но не выпускает управление в цикл обработки событий. Такой метод будет работать лучше в многопоточной системе, но мы пока не хотим применять потоки. Итак, как можно это исправить?

Мы можем задействовать функцию `asyncio.sleep()`, которая является асинхронной версией `time.sleep()`, и ждать ее результата с помощью ключевого слова `await`. Мы уже использовали данный оператор в первой версии функции `main()`, но только для улучшения ясности кода. Это явно не делает нашу реализацию более параллельной. Посмотрим на следующую улучшенную версию сопрограммы `waiter()`, в которой применяется `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(
            "{} waited {} seconds"
            "".format(name, time_to_sleep)
        )
```

Запустив обновленный скрипт, мы увидим, что вывод двух функций чередуется:

```
$ time python corowait_improved.py
second waited 0.25 seconds
first waited 0.25 seconds
second waited 0.25 seconds
first waited 0.5 seconds
first waited 0.25 seconds
second waited 0.75 seconds
first waited 0.25 seconds
second waited 0.5 seconds
```

```
real   0m1.953s
user   0m0.149s
sys    0m0.026s
```

Дополнительное преимущество этого простого улучшения заключается в том, что код стал работать быстрее. Общее время выполнения оказалось меньше, чем сумма времен выполнения, поскольку сопрограммы поочередно передавали управление.

В следующем подразделе мы рассмотрим `asyncio` в более старых версиях Python.

Модуль asyncio в старых версиях Python

Модуль `asyncio` появился в Python 3.4, поэтому только в данной версии Python есть серьезная поддержка асинхронного программирования, даже в версиях ранее Python 3.5. К сожалению, кажется, что этих двух версий достаточно для устранения проблем совместимости.

Так или иначе, заготовки для асинхронного программирования в Python появились раньше, чем синтаксические элементы, поддерживающие этот шаблон. Конечно, лучше поздно, чем никогда, но это создало ситуацию, в которой появилось два разных синтаксиса для работы с сопрограммами.

Начиная с Python 3.5, можно использовать `async` и `await` следующим образом:

```
async def main():
    await asyncio.sleep(0)
```

В Python 3.4 нужно использовать декоратор `asyncio.coroutine` и оператор `yield from` так:

```
@asyncio.couroutine
def main():
    yield from asyncio.sleep(0)
```

Другой полезный факт заключается в том, что оператор был введен в Python 3.3 и в PyPI есть инструмент портирования `asyncio`. Это значит, вы можете использовать данную реализацию кооперативной многозадачности в Python 3.3.

В следующем подразделе рассмотрим практический пример асинхронного программирования.

Практический пример асинхронного программирования

Как мы уже несколько раз упоминали в данной главе, асинхронное программирование — отличный инструмент для обработки операций ввода/вывода. Поэтому пришло время создать нечто более практичное, чем простой вывод чисел или асинхронное ожидание.

Чтобы соблюсти последовательность, мы попробуем решить ту же задачу, которую решили ранее с помощью многопоточности и многопроцессорности. То есть попробуем асинхронно получать некоторые данные о текущих обменных курсах валют от внешнего источника через Интернет. Было бы здорово, если бы мы могли использовать библиотеку `requests`, как и в предыдущих разделах. Но мы не можем этого сделать, а точнее, можем, но неэффективно.

К сожалению, данная библиотека не поддерживает асинхронный ввод/вывод с ключевыми словами `async` и `await`. Существуют другие проекты, которые направлены на добавление параллелизма в `requests`, но они работают либо через GEvent (`grequests`) (github.com/kennethreitz/grequests), либо через пул потоков/процессов (`requests-futures`) (github.com/ross/requests-futures). Ни один из этих способов не решает нашу проблему.

Зная об ограничениях библиотеки, которая с такой легкостью была использована в предыдущих примерах, мы должны чем-то заполнить данный пробел. API валютных курсов очень легок в применении, поэтому нам просто нужно будет использовать некую родную библиотеку. В стандартной библиотеке Python в версии 3.7 по-прежнему нет никаких библиотек, которые позволяют делать асинхронные HTTP наподобие `urllib.urlopen()`. Мы определенно не хотим создавать поддержку протокола с нуля, поэтому воспользуемся помощью пакета `aiohttp`, доступного на PyPI. Это многообещающая библиотека, в которой добавляются клиентские и серверные реализации для асинхронного HTTP. Ниже приведен небольшой модуль, построенный на основе `aiohttp`, создающий одну вспомогательную функцию `get_rates()`, которая делает запросы валютных курсов:

```
import aiohttp

async def get_rates(session: aiohttp.ClientSession, base: str):
    async with session.get(
        f"https://api.exchangeratesapi.io/latest?base={base}"
    ) as response:
        rates = (await response.json())['rates']
        rates[base] = 1.

        return base, rates
```

Предположим, что этот код хранится в модуле с именем `asyncrates`, который мы будем использовать позже. Теперь мы можем переписать пример, позволивший нам исследовать многопоточность и многопроцессорность. Ранее мы делили всю работу на два отдельных этапа.

1. Выполнение всех запросов к внешнему сервису параллельно с использованием функции `fetch_place()`.
2. Отображение всех результатов в цикле с помощью функции `present_result()`.

Поскольку кооперативная многозадачность совершенно отличается от применения нескольких процессов или потоков, мы можем слегка изменить наш подход. Большинство вопросов, рассмотренных нами выше в подпункте «Использование отдельного потока для каждого элемента», уже отпало. Мы легко можем вывести результаты сразу после ожидания HTTP-запроса. Это упростит код и сделает его более понятным:

```
import asyncio
import time

import aiohttp

from asyncrates import get_rates

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

async def fetch_rates(session, place):
    return await get_rates(session, place)

async def present_result(result):
    base, rates = (await result)

    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:
        await asyncio.wait([
            present_result(fetch_rates(session, base))
            for base in BASES
        ])

if __name__ == "__main__":
    started = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    elapsed = time.time() - started

    print()
    print("time elapsed: {:.2f}s".format(elapsed))
```

Это довольно легко для простого API. Но иногда вам нужна специализированная клиентская библиотека, которая не является асинхронной и не портируется мановением руки. Мы рассмотрим такую ситуацию в следующем подразделе.

Интеграция синхронного кода с помощью фьючерсов async

Асинхронное программирование восхитительно, особенно для разработчиков масштабируемых серверных приложений. На практике это один из самых важных инструментов для построения высокопараллельных серверов.

Но, как известно, жизнь вносит коррективы. Многие популярные пакеты, работающие с вводом/выводом, не предназначены для решения задач в сочетании с асинхронным кодом. Основные причины этого таковы:

- низкий уровень использования продвинутых функций Python 3 (особенно асинхронное программирование);
- низкий уровень понимания различных концепций параллелизма среди начинающих программистов Python.

Это значит, что зачастую миграция существующих синхронных многопоточных приложений и пакетов либо совершенно невозможна (из-за архитектурных ограничений), либо слишком дорога. Многие проекты могли бы стать гораздо более внушительными после внедрения асинхронного стиля многозадачности, но лишь немногие из них в конечном итоге делают это.

Это говорит о том, что прямо в настоящий момент вы испытываете много трудностей при попытке построить асинхронные приложения с нуля. Эти проблемы будут в том или ином смысле похожи на проблемы, упомянутые нами в предыдущем подразделе «Практический пример асинхронного программирования», — несовместимые интерфейсы и синхронная блокировка операций ввода/вывода.

Конечно, иногда можно и отказаться от использования ключевого слова `await`, когда получается такая несовместимость, и просто получать необходимые ресурсы синхронно. Но это не дает всей остальной сопрограмме выполнять свой код, пока вы ждете результатов. Технически схема работает, но все преимущества асинхронного программирования в данном случае сводятся на нет. Короче говоря, совмещение асинхронного ввода/вывода с синхронным вводом/выводом — плохой вариант. Это своего рода игра «все или ничего».

Еще одна проблема — долгие операции с ЦП. При выполнении операции ввода/вывода отпустить управление из сопрограммы не составит труда. При записи/чтении из файловой системы или порта в какой-то момент настанет время ожидания, поэтому использование ключевого слова `await` — лучшее, что вы можете сделать. Но как поступить, когда вам нужно что-то посчитать и вы знаете, что это займет некоторое время? Вы можете разделить задачу на части и отпускать управление всякий раз, когда работа слегка продвигается. Но вскоре обнаружите, что это не очень хорошо. Код станет беспорядочным, и при этом не будет гарантировать хороших результатов. Нарезка задачи по времени должна лежать на интерпретаторе или операционной системе.

Итак, как поступить, если у вас есть некий код, который выполняет длинные синхронные операции ввода/вывода, и вы не можете или не хотите его переписывать? И стоит ли вообще это делать, имея тяжелые операции на ЦП в приложении, созданном под асинхронный ввод/вывод? Вероятно, стоит использовать обходной путь. И под таковым мы имеем в виду многопоточность или многопроцессорность.

Это звучит не слишком приятно, но иногда лучшее решение — то, которого мы пытались избежать. Параллельная обработка тяжелых для ЦП задач в Python всегда лучше выполняется через многопроцессорность. Многопоточность тоже может хорошо работать с операциями ввода/вывода (быстро и без больших затрат ресурсов) с помощью ключевых слов `async` и `await`, если вы все настроите правильно и будете работать осторожно.

Поэтому, когда что-то попросту не подходит к вашему асинхронному приложению, используйте код, перекладывающий данную задачу на отдельный поток или процесс. Можно делать вид, что это была сопрограмма, и передавать управление в цикл обработки событий с помощью `await`. В конечном итоге вы все равно обработаете результаты, когда они будут готовы. К счастью для нас, в стандартной библиотеке Python есть модуль `concurrent.futures`, который также интегрирован с `asyncio`. Оба модуля совместно позволяют планировать функции блокировки для выполнения в потоках или дополнительных процессах, как если бы были асинхронными неблокирующими сопрограммами.

В следующем пункте поговорим об исполнителях и фьючерсах.

Исполнители и фьючерсы

Прежде чем мы рассмотрим, как вводить потоки или процессы в асинхронный цикл событий, более подробно поговорим о модуле `concurrent.futures`, который позже станет основным компонентом нашего так называемого «обходного пути».

Наиболее важные классы модуля `concurrent.futures` — это `Executor` и `Future`.

Класс `Executor` представляет собой пул ресурсов, которые могут обрабатывать рабочие элементы параллельно. Он может показаться очень похожим в своей цели на классы из модуля `multiprocessing` — `Pool` и `dummy.Pool`, но имеет совершенно другой интерфейс и семантику. Это базовый класс, не предназначенный для конкретизации; он содержит две реализации:

- `ThreadPoolExecutor` — представляет собой пул потоков;
- `ProcessPoolExecutor` — представляет собой пул процессов.

У каждого исполнителя есть три следующих метода:

- `submit(func, *args, **kwargs)` — планирует функцию `func` для выполнения в пуле ресурсов и возвращает объект `Future`, представляющий выполнение вызываемого;

- `map(func, *iterables, timeout=None, chunksize=1)` — выполняет функцию `func` над `Iterable` аналогично методу `multiprocessing.Pool.map()`;
- `shutdown(wait = True)` — выключает исполнителя и освобождает все его ресурсы.

Самый интересный метод из перечисленных — это `submit()`, поскольку он возвращает объект `Future`. Он представляет собой асинхронное выполнение вызываемого объекта и лишь косвенно представляет свой результат. Чтобы получить фактическое возвращаемое значение, вам нужно вызвать метод `Future.result()`. И если вызываемый объект уже выполнился, то метод `result()` не будет блокироваться и просто вернет результат функции. В противном случае он станет блокироваться до момента готовности результата. Это своего рода обещание результата (на самом деле это та же концепция, что и промис в JavaScript). Вам не нужно распаковать его сразу после получения (с помощью метода `result()`), но если вы попытаетесь сделать это, то он гарантированно в конечном итоге вернет нечто подобное представленному ниже:

```
>>> def loudly_return():
...     print("processing")
...     return 42
...
>>> from concurrent.futures import ThreadPoolExecutor
>>> with ThreadPoolExecutor(1) as executor:
...     future = executor.submit(loudly_return)
...
processing
>>> future
<Future at 0x33cbf98 state=finished returned int>
>>> future.result()
42
```

Если вы хотите использовать метод `Executor.map()`, то по своему применению он ничем не отличается от `Pool.map()` из модуля `multiprocessong`:

```
def main():
    with ThreadPoolExecutor(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)
```

В следующем подпункте мы рассмотрим, как использовать исполнителей в цикле обработки событий.

Использование исполнителей в цикле обработки событий

Экземпляры класса `Future`, возвращаемые методом `Executor.submit()`, в своей концепции очень близки к сопрограммам, используемым в асинхронном программировании. Именно поэтому мы можем применять исполнителей, чтобы создать

своего рода гибрид кооперативной многозадачности и многопроцессорности или многопоточности.

Ядро этого обходного пути — метод `BaseEventLoop.run_in_executor(executor, func, *args)` класса цикла событий. Он позволяет планировать выполнение функции `func` в пуле процессов или потоков, представленном аргументом `executor`. Самое важное в этом методе — то, что он возвращает новый *ожидаемый* объект (который можно поставить *в режим ожидания* оператором `await`). Таким образом, вы можете выполнить блокирующую функцию, не являющуюся сопрограммой, как если бы это была сопрограмма, и блокировки не случится, независимо от того, сколько времени требуется на выполнение. Заблокироваться может только функция, которая ожидает результатов от такого вызова, но весь цикл событий будет работать.

И полезный факт: вам не нужно даже создавать экземпляр исполнителя. Если вы передаете `None` в качестве аргумента исполнителя, то класс `ThreadPoolExecutor` будет использоваться с количеством потоков по умолчанию (для Python 3.7 это количество процессоров, умноженное на 5).

Итак, предположим, что мы не хотим переписывать проблемную часть работающего с API кода. Мы легко можем перенести вызов блокировки в отдельный поток с помощью `loop.run_in_executor()`, в то же время превращая функцию `fetch_rates()` в ожидаемую сопрограмму:

```
async def fetch_rates(base):
    loop = asyncio.get_event_loop()
    response = await loop.run_in_executor(
        None, requests.get,
        f"https://api.exchangeratesapi.io/latest?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]
    # Примечание: валюта обменивается сама на себя с коэффициентом 1:1
    rates[base] = 1.
    return base, rates
```

Резюме

Это было сложно, но мы успешно разобрали большинство основных подходов к параллельному программированию для Python-программистов.

Поняв, что такое параллелизм, мы перешли к действию и разобрали на части одну из типичных параллельных задач с помощью многопоточности. Выявив основные недостатки нашего кода и исправив их, мы перешли к многопроцессорности, чтобы посмотреть, как все будет работать в нашем случае.

Мы обнаружили, что несколько процессов с модулем `multiprocessing` оказалось проще использовать, чем `threading`. Но сразу после этого поняли, что можем

применять тот же API для потоков благодаря модулю `multiprocessing.dummy`. Таким образом, решение между многопроцессорностью и многопоточностью будет зависеть от задачи, а не от удобства интерфейса.

Говоря о задачах, мы наконец попробовали асинхронное программирование, которое должно быть лучшим решением для приложений с большим количеством ввода/вывода, и в результате поняли, что не можем полностью отказаться от потоков и процессов. Таким образом, мы вернулись туда, откуда начали.

И это приводит нас к окончательному выводу — панацеи нет. Есть подходы, которые могут быть более предпочтительными, и, чтобы выбрать верный, нужно знать их все. Вы можете сделать правильный выбор самостоятельно, используя весь арсенал инструментов параллельности в одном приложении, и это не редкость.

Предыдущий вывод — прекрасное введение в тему следующей главы 17, посвященной паттернам проектирования. Понятно, что нет одного паттерна, который решит все ваши проблемы. Но вы должны знать их как можно больше, поскольку в конечном итоге будете использовать постоянно.

В следующей главе мы рассмотрим тему, в некоторой степени связанную с параллелизмом: событийно-ориентированное и сигнальное программирование. Мы сосредоточимся на различных коммуникационных моделях, которые являются основой распределенных асинхронных и высокопараллельных систем.

Часть V
Техническая архитектура

В этой части мы рассмотрим различные архитектурные модели и парадигмы, призванные сделать архитектуру программного обеспечения простой и устойчивой. Мы изучим, как можно разъединить даже большие и сложные системы. Познакомимся с самыми распространенными паттернами проектирования, которые используют разработчики на Python.

16 Событийно-ориентированное и сигнальное программирование

В главе 15 мы рассмотрели различные модели реализации параллельной обработки в Python. Чтобы лучше объяснить это понятие, мы использовали следующее определение: *два события происходят одновременно, и ни одно из них не влияет на другое*.

Мы часто представляем события как некие упорядоченные моменты времени, которые происходят один за другим и часто имеют причинно-следственную связь. Но в программировании под событием понимается нечто иное. Событие — не что-то, что *происходит*. События в программировании — независимые единицы информации, которые может обработать программа. Именно это понятие события — реальный краеугольный камень параллелизма.

Параллельное программирование — парадигма программирования для обработки одновременных событий. Существует обобщение этой парадигмы, которое имеет дело с голой концепцией событий, независимо от того, являются ли те одновременными. Подход к программированию, в котором программы обрабатываются как поток событий, называется *событийно-ориентированным программированием*.

Это важная парадигма, поскольку позволяет легко разделить на части даже большие и сложные системы. Это помогает расставить четкие границы между независимыми компонентами и улучшить изоляцию между ними.

В этой главе:

- что такое событийно-ориентированное программирование;
- различные стили событийно-ориентированного программирования;
- событийно-ориентированные архитектуры.

Прочитав главу, вы узнаете о самых распространенных методах событийно-ориентированного программирования в Python, а также о том, как экстраполировать их на событийно-ориентированные архитектуры. Научитесь легко определять проблемы, которые моделируются с помощью событийно-ориентированного программирования.

Технические требования

Ниже приведены пакеты Python, упомянутые в этой главе, которые можно скачать с PyPI:

- `flask`;
- `blinker`.

Для запуска примеров `kinter` вам потребуется библиотека `Tk` для Python. Обычно она доступна по умолчанию в большинстве дистрибутивов Python, однако в некоторых операционных системах требуется дополнительная установка пакета. Он обычно называется `python3-tk`.

Установить эти пакеты можно с помощью следующей команды:

```
python3 -m pip install <имя-пакета>
```

Файлы кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter16.

Что такое событийно-ориентированное программирование

Событийно-ориентированное, или событийное, программирование концентрируется на событиях (сообщениях) и их потоках между компонентами программного обеспечения. Вы наверняка замечали, что понятие «события» встречается во многих типах программного обеспечения. Исторически сложилось, что программирование, основанное на событиях, — наиболее распространенная парадигма для программного обеспечения, ориентированного на непосредственное взаимодействие с человеком. То есть это самая подходящая парадигма для графических интерфейсов. Почти любая программа ожидает со стороны человека некоего ввода, который моделируется в виде событий или сообщений. В такой постановке событийно-ориентированная программа — просто набор событий или сообщений обработчиков, реагирующих на взаимодействие с человеком.

События также не должны быть прямым результатом взаимодействия с пользователем. Архитектура любого веб-приложения тоже управляется событиями. Браузеры отправляют на веб-сервер запросы от имени пользователя, и они часто обрабатываются как отдельные события. Сами запросы, конечно, часто являются результатом непосредственного ввода пользователя (он-то просто нажимает кнопки и ссылки), но это не всегда так. Многие современные приложения могут синхронизировать информацию с веб-сервером без участия пользователя, и такое общение происходит в автоматическом режиме, а пользователь об этом даже не знает.

Таким образом, событийно-ориентированное программирование — общий способ сочетания программных компонентов различных размеров, которое происходит на разных уровнях архитектуры программного обеспечения. В зависимости от масштаба и типа архитектуры ПО подобное программирование может принимать различные формы. Это может быть:

- модель параллелизма, непосредственно поддерживаемая семантикой данного языка программирования (например, `async/await` в Python);
- способ структурирования кода приложения диспетчерами/обработчиками событий, сигналами и т. д.;
- архитектура межпроцессной или межсервисной связи, позволяющая объединять независимые компоненты программного обеспечения в более крупную систему.

В следующем подразделе обсудим, чем событийно-ориентированное программирование отличается для асинхронных систем.

Событийно-ориентированный != асинхронный

Событийно-ориентированное программирование — парадигма, чрезвычайно распространенная в асинхронных системах, но это не значит, что каждое событийно-ориентированное приложение должно быть асинхронным. Это также не означает, что событийно-ориентированное программирование подходит только для параллельных и асинхронных приложений. На самом деле данный подход чрезвычайно полезен даже для разделения задач, которые до этого были строго синхронными, но не параллельными.

Рассмотрим, например, триггеры базы данных, имеющиеся практически в любой реляционной БД системы. Триггер БД — процедура, выполняемая в ответ на определенное событие, происходящее в базе. Это основной строительный блок систем баз данных, помимо прочего, позволяющий БД сохранять целостность данных в сценариях, которые не получается моделировать из-за ограничений, присущих базам. Так, в базе данных PostgreSQL выделяется три типа событий на уровне строк, которые могут возникнуть в любой таблице или на панели просмотра:

- `INSERT`;
- `UPDATE`;
- `DELETE`.

В случае со строками таблицы можно определить триггеры для выполнения до (`BEFORE`) или после (`AFTER`) определенного события. Таким образом, с точки зрения связи событий и процедур мы можем рассматривать каждый вариант `AFTER/BEFORE` как отдельное событие. Чтобы лучше это понять, рассмотрим следующий пример триггеров базы 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();
```

Здесь есть два триггера, которые выполняются, когда обновляется строка в таблице `users`. Первый из них выполняется перед обновлением, а второй — после. Это значит, что оба события зависят от событий и не могут обрабатываться одновременно. С другой стороны, подобные наборы событий, происходящие в разных строках из различных сессий, могут быть одновременными. Будут ли триггеры, поступающие из разных сессий, независимыми и можно ли обработать их асинхронно, зависит от множества факторов (является ли транзакцией, уровень изоляции и др.) и в целом от системы баз данных. Но это вовсе не значит, что систему в целом нельзя смоделировать, как если бы события были действительно независимыми.

В следующем подразделе мы рассмотрим событийно-ориентированное программирование в графических пользовательских интерфейсах (GUI).

Событийно-ориентированное программирование в GUI

Графические пользовательские интерфейсы (graphical user interface, GUI) приходят на ум первыми, когда речь заходит о событийно-ориентированном программировании. Данный вид программирования — элегантный способ сочетания пользовательского ввода с GUI, поскольку именно он показывает, как пользователь взаимодействует с графическим интерфейсом. Такие интерфейсы часто включают множество компонентов, с которыми пользователь может выстроить взаимодействие, и оно почти всегда нелинейно. В сложной модели интерфейсов взаимодействие происходит через совокупность событий, создаваемых пользователем из различных компонентов интерфейса.

Понятие события — общее для большинства библиотек и фреймворков пользовательских интерфейсов, но в разных библиотеках применяются разные паттерны проектирования для достижения событийно-ориентированной связи. Некоторые библиотеки даже задействуют другие понятия, описывающие их архитектуру (например, сигналы в библиотеке Qt). Тем не менее общая картина всегда одинакова: каждый компонент интерфейса (часто называемый *виджетом*) может при взаимодействии генерировать события и они могут передаваться другим компонентам или непосредственно прикрепляться к обработчикам событий. В зависимости от библиотеки GUI событиями могут быть простые именованные сигналы, которые

говорят, что произошло некое действие (например, щелчок на виджете *A*), или более сложные сообщения, содержащие дополнительную информацию о контексте взаимодействия (скажем, нажатие клавиши или положение указателя мыши).

Мы обсудим различия паттернов проектирования в разделе «Различные стили событийно-ориентированного программирования» позже. А сейчас посмотрим на пример приложения Python с интерфейсом, которое можно создать с помощью встроенного модуля `tkinter`:

```
import this
from tkinter import *
from tkinter import messagebox

rot13 = str.maketrans(
    "ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz",
    "NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm"
)

def main_window(root):
    frame = Frame(root, width=100, height=100)
    zen_button = Button(root, text="Python Zen", command=show_zen)
    zen_button.pack()

def show_zen():
    messagebox.showinfo(
        "Zen of Python",
        this.s.translate(rot13)
    )

if __name__ == "__main__":
    root = Tk()
    main_window(root)
    root.mainloop()
```

Библиотека Tk, содержащая модуль tkinter, обычно идет в комплекте с дистрибутивом Python. Если ее почему-то нет в вашей операционной системе, то вы сможете установить ее с помощью менеджера пакетов системы. Например, на Debian можно легко установить ее на Python под именем python3-tk, используя следующую команду:

```
sudo apt-get install python3-tk
```

Это приложение с графическим интерфейсом выводит одну кнопку *Python Zen*. Когда она будет нажата, приложение откроет новое окно, содержащее текст `Zen of Python`, который был импортирован из модуля `this`. Модуль `tkinter` позволяет создавать более конкретные события (нажатия клавиш или щелчки кнопкой мыши), которые будут связаны с обратными вызовами с помощью метода `bind()`, однако это не всегда полезно. Вместо метода `bind()` мы задействуем аргумент `command`

виджета `Button`. Это переведет исходные события ввода (нажатие и отпускание кнопки мыши) в обратный вызов функции при сохранении удобства использования самого интерфейса (например, запуск действия, только когда кнопка мыши будет отпущена над кнопкой). Большинство фреймворков GUI работает аналогичным образом — вы редко работаете с сырыми событиями клавиатуры и мыши, но связываете ваши команды/обратные вызовы с событиями более высокого уровня, такими как:

- изменение состояния флажка;
- щелчок на кнопке;
- выбор опции;
- закрытие окна.

В следующем подразделе мы рассмотрим событийно-ориентированную связь.

Событийно-ориентированная связь

Событийно-ориентированное программирование — очень распространенный метод построения распределенных сетевых приложений, особенно с появлением сервисно-ориентированной и микросервисной архитектур. Событийно-ориентированное программирование позволяет легко делить сложные системы на отдельные компоненты, которые имеют ограниченный набор функций. В сервисно-ориентированных или микросервисных архитектурах поток событий происходит не между классами или функциями внутри одного процесса, а между сетевыми сервисами. В крупных распределенных архитектурах поток событий между сервисами часто координируется с помощью специальных протоколов (например, AMQP и ZeroMQ) и/или специализированных сервисов. Мы обсудим некоторые из этих решений позже, в разделе «Событийно-ориентированные архитектуры».

Тем не менее вам не нужно иметь формализованный способ координации событий, равно как и сервис обработки событий, чтобы считать ваше сетевое приложение событийно-ориентированным. На самом деле, если посмотреть на любое веб-приложение на Python, можно заметить, что большинство веб-фреймворков Python имеют много общего с приложениями GUI. Так, рассмотрим простое веб-приложение, которое было написано с помощью 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()
```

Сравнив этот листинг с примером приложения `tkInter` из предыдущего подраздела, вы заметите, что они структурно очень похожи. Конкретные маршруты (пути) HTTP-запросов переводятся на выделенные обработчики. Если мы считаем наше приложение событийно-ориентированным, то путь запроса может рассматриваться как связь между конкретным типом событий (скажем, нажатие ссылки) и обработчиком. Подобно событиям в GUI-приложениях, HTTP-запросы могут содержать дополнительные данные о контексте взаимодействия. Эта информация, конечно, намного более структурирована, поскольку протокол HTTP определяет несколько типов запросов (например, `POST`, `GET`, `PUT` и `DELETE`) и несколько способов передачи дополнительных данных (строка запроса, тело запроса и заголовки).

Конечно, в случае применения Flask пользователь не взаимодействует с базой непосредственно, а задействует в качестве интерфейса браузер. Но так ли уж велика эта разница? Многие кросс-платформенные библиотеки пользовательских интерфейсов (например, Tcl/Tk, Qt и GTK +) представляют собой просто прокси между приложением и API системы. Таким образом, в обоих случаях речь идет о связи и событиях, проходящих через различные слои приложения. При этом в веб-приложениях слои более очевидны и связь всегда явная.

В следующем разделе мы рассмотрим различные стили событийно-ориентированного программирования.

Различные стили событийно-ориентированного программирования

Как мы уже говорили, событийно-ориентированное программирование может быть реализовано на различных уровнях архитектуры программы с помощью разных паттернов проектирования. Оно также часто применяется к специфическим отраслям программной инженерии в таких областях, как создание сетей, системное программирование и программирование графических пользовательских интерфейсов. Таким образом, событийно-ориентированное программирование — не одиночный подход к программированию, а коллекция разнообразных моделей, инструментов и алгоритмов, формирующих общую парадигму, которая концентрируется на программировании вокруг потока событий.

В связи с этим событийно-ориентированное программирование существует во множестве различных вариантов и стилей. Его фактические реализации могут быть основаны на различных паттернах и методах проектирования. В ряде из этих методов и инструментов даже не используется сам термин «событие». Однако, несмотря на такое разнообразие, мы без труда можем выделить несколько основных стилей событийно-ориентированного программирования, которые являются основой более конкретных моделей.

В следующих подразделах мы приведем краткий обзор трех основных стилей событийно-ориентированного программирования, с которыми вы можете столкнуться при написании программ на Python.

Стиль на основе обратных вызовов

Программирование на основе обратных вызовов — один из наиболее распространенных стилей событийно-ориентированного программирования. В этом стиле объекты, которые генерируют события, отвечают за определение их обработчиков событий. Получаются связи «один к одному» или (по большей части) «многие к одному» между генераторами событий и их обработчиками.

Этот стиль событийно-ориентированного программирования доминирует среди фреймворков и библиотек GUI. Причина проста — такой подход соответствует тому, как пользователи и программисты думают о пользовательских интерфейсах. Каждое действие, которое мы делаем: манипуляции с переключателем, нажатие кнопки или установка флажка — имеет под собой конкретную цель.

Мы уже видели пример событийно-ориентированного программирования на основе обратного вызова в примере графического приложения, написанного с помощью библиотеки `tkinter` (см. подраздел «Событийно-ориентированное программирование в GUI» предыдущего раздела). Вспомним одну строку из того приложения:

```
zen_button = Button(root, text="Python Zen", command=show_zen)
```

Данный экземпляр класса `Button` определяет, что функция `show_zen()` должна вызываться каждый раз, когда нажимается кнопка. Наше событие неявное, и обратный вызов `show_zen()` (в `tkinter` обратные вызовы называются *командами*) не получает никакого объекта, инкапсулирующего событие, которое порождает его вызов. В этом есть смысл, поскольку ответственность за прикрепление обработчиков событий должна лежать скорее на генераторе событий (здесь это кнопка), а сам факт возникновения события обработчику малоинтересен.

В некоторых реализациях событийно-ориентированного программирования на основе обратного вызова фактическая связь между генераторами событий и их обработчиками представляет собой отдельный шаг, который может быть выполнен после инициализации генератора. Этот стиль связывания доступен и в `tkinter`, но только для необработанных взаимодействий с пользователем. Ниже приведен обновленный фрагмент из предыдущего приложения на `tkinter`, применяющего этот стиль привязки событий:

```
def main_window(root):
    frame = Frame(root, width=100, height=100)

    zen_button = Button(root, text="Python Zen")
    zen_button.bind("<ButtonRelease-1>", show_zen)
    zen_button.pack()

def show_zen(event):
    messagebox.showinfo(
        "Zen of Python",
        this.s.translate(rot13)
    )
```

В предыдущем примере событие больше не является неявным, поэтому обратный вызов `show_zen()` должен уметь принимать объект события. Он содержит основную информацию о действии пользователя, например о положении указателя мыши, время события, и связанный с ним виджет. Важно помнить: данный тип привязки событий по-прежнему одноадресный. Это значит, что одно событие (здесь `<ButtonRelease-1>`) от одного объекта (здесь `zen_button`) может быть связано только с одним обратным вызовом (здесь `show_zen()`). Можно прикрепить один обработчик для нескольких событий и/или объектов, но одно событие, которое происходит из одного источника, может получить лишь один обратный вызов. Любая попытка присоединить новую функцию обратного вызова с помощью метода `bind()` перекроет старую связь.

Одноадресный характер событийного программирования на основе обратных вызовов имеет очевидные ограничения, поскольку влечет за собой тесную связь компонентов приложения. Невозможность подключения нескольких обработчиков для одиночных событий часто означает, что каждый обработчик

работает только с одним генератором и не может быть привязан к объектам другого типа.

Теперь рассмотрим стиль на основе субъекта.

Стиль на основе субъекта

Стиль событийного программирования *на основе субъекта* — естественное продолжение предыдущего стиля. Здесь генераторы событий (субъекты) позволяют другим объектам подписываться/регистрироваться для получения уведомлений о своих событиях. Этот стиль похож на предыдущий, поскольку генераторы обычно хранят список функций или методов, которые можно вызывать, когда происходит какое-то новое событие.

При программировании на основе субъекта фокус перемещается от события к субъекту (генератору события). Наиболее распространенный пример этого стиля — паттерн проектирования `Observer`. Мы подробно обсудим его в главе 17, однако он настолько важен, что мы не можем рассматривать событийное программирование на основе субъекта, не приведя хотя бы краткий обзор. Таким образом, мы взглянем на него одним глазком, просто чтобы посмотреть, чем этот стиль отличается от программирования на основе обратного вызова, а подробности этого паттерна обсудим уже в следующей главе.

Говоря вкратце, паттерн проектирования `Observer` состоит из двух классов объектов — наблюдателей и субъектов (иногда наблюдаемых). `Subject` — объект, содержащий список паттернов `Observer`, которым интересно, что происходит с `Subject`. Таким образом, `Subject` — генератор события, а `Observer` — обработчик.

Простая реализация паттерна `Observer` может выглядеть примерно так:

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

    def register(self, observer):
        self._observers.append(observer)

    def _notify_observers(self, event):
        for observer in self._observers:
            observer.notify(self, event)

class Observer:
    def notify(self, subject, event):
        print(f"Received event {event} from {subject}")
```

Это, конечно, всего лишь основа. Метод `_notify_observers()` должен вызываться внутри класса `Subject` всякий раз, когда происходит нечто потенциально интересное

зарегистрированным наблюдателям. Это может быть любое событие, но, как правило, субъекты информируют наблюдателей о важных изменениях своего состояния.

Чтобы это продемонстрировать, предположим, что субъекты уведомляют всех своих наблюдателей о появлении новых наблюдателей. Ниже представлены обновленные классы `Observer` и `Subject`, которые показывают процесс обработки событий:

```
import itertools

class Subject:
    _new_id = itertools.count(1)

    def __init__(self):
        self._id = next(self._new_id)
        self._observers = []

    def register(self, observer):
        self._notify_observers(f"register({observer})")
        self._observers.append(observer)

    def _notify_observers(self, event):
        for observer in self._observers:
            observer.notify(self, event)

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"

class Observer:
    _new_id = itertools.count(1)

    def __init__(self):
        self._id = next(self._new_id)

    def notify(self, subject, event):
        print(f"{self}: received event '{event}' from {subject}")

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"
```

Если вы попытаетесь создать экземпляр и связать эти классы в интерактивной сессии интерпретатора, то увидите следующий результат:

```
>>> from subject_based_events import Subject
>>> subject = Subject()
>>> observer1 = Observer()
>>> observer2 = Observer()
>>> observer3 = Observer()
>>> subject.register(observer1)
>>> subject.register(observer2)
<Observer: 1>: received event 'register(<Observer: 2>)' from <Subject: 1>
```

```
>>> subject.register(observer3)
<Observer: 1>: received event 'register(<Observer: 3>)' from <Subject: 1>
<Observer: 2>: received event 'register(<Observer: 3>)' from <Subject: 1>
```

Событийное программирование на основе субъекта позволяет группировать обработчики событий. Этот тип обработки дает возможность использовать более гибкие обработчики событий, что является сильным преимуществом для обеспечения модульности. К сожалению, смещение фокуса с событий на субъекты может стать обременительным. В нашем примере наблюдатели будут уведомлены о каждом событии, генерируемом классом `Subject`. Они не имеют возможности зарегистрироваться лишь на определенные типы событий. При большом количестве субъектов и адресатов это может стать проблемой. Либо наблюдателю придется фильтровать все входящие события, либо объекту надо будет позволять наблюдателям регистрировать только определенные события. Первый подход представляется неэффективным в случае достаточно большого количества проходящих через фильтр событий. Второй подход может слишком усложнить регистрацию наблюдателя и отправку события.

Несмотря на преимущества, подход на основе субъекта к событийному программированию редко делает компоненты приложения более слабосвязанными, чем подход на основе обратного вызова. Поэтому он не подходит для архитектуры больших приложений, а вот для мелких частных задач — вполне. В основном это связано с тем, что подход ориентирован на субъекты, у которых обработчики должны помнить много предположений о наблюдаемых субъектах. Кроме того, в реализации этого стиля (паттерн `Observer`) и наблюдатель, и субъект должны в какой-то момент встретиться в одном и том же контексте. Другими словами, наблюдатели не могут зарегистрироваться на события, если нет субъекта, который их генерирует.

К счастью, существует стиль событийно-ориентированного программирования, позволяющий выполнять обработку многоадресных событий таким образом, чтобы поддерживать слабосвязанность в больших приложениях. Это тематический стиль — прямое продолжение событийного программирования на основе субъекта.

В следующем подразделе мы рассмотрим тематический стиль.

Тематический стиль

Тематическое событийное программирование ставит во главу угла типы событий, которые передаются между программными компонентами, не склоняясь в какую-либо сторону взаимоотношений «генератор — обработчик». Тематическое программирование — обобщение предыдущих стилей. Событийно-ориентированные приложения, написанные в этом стиле, позволяют компонентам (например, классам, объектам и функциям) генерировать события и/или регистрироваться на типы событий, полностью игнорируя другую сторону.

Другими словами, обработчики могут быть зарегистрированы на типы событий, даже при отсутствии генератора, который будет создавать их, а генераторы — выдавать события, даже если никто не подписан на их получение. В этом стиле событийно-ориентированного программирования события выступают сущностями первого класса, которые часто определяются отдельно от генераторов и обработчиков. Такие события нередко относятся к специальному классу или просто являются глобальными одноэлементными экземплярами одного общего класса `Event`. Поэтому обработчики могут подписываться на события, даже если нет объекта, который бы их генерировал.

В зависимости от выбранного фреймворка или библиотеки выбора абстракция, которая используется для инкапсуляции таких наблюдаемых типов событий/классов, может называться по-разному. Популярные термины — «канал», «тема» и «сигналы». Термин *«сигнал»* особенно популярен, и потому этот стиль программирования часто называют *сигнальным программированием*. Сигналы используются в таких популярных библиотеках и фреймворках, как Django (веб-фреймворк), Flask (веб-микрофреймворк), SQLAlchemy ORM (база данных) и Scrapy (веб-фреймворк).

Удивительно, но успешные проекты Python не делают собственные сигнальные фреймворки с нуля, а задействуют уже существующую специализированную библиотеку. Самая популярная библиотека сигналов в Python — `blinker`. Она характеризуется чрезвычайно широкой совместимостью с Python (Python 2.4 или более поздняя версия, Python 3.0 или более поздняя версия, Jython 2.5 или выше либо PyPy 1.6 или более поздняя версия) и имеет чрезвычайно простой и краткий API, что позволяет использовать ее практически в любом проекте.

Библиотека `blinker` построена на концепции именованных сигналов. Новое определение сигнала создается с помощью конструктора `signal(name)`. Два отдельных вызова конструктора `signal(name)` с одинаковым значением имени будут возвращать один и тот же объект сигнала. Это позволяет легко ссылаться на сигналы в любой момент. Ниже приведен пример класса `SelfWatch`, который использует именованные сигналы, чтобы его экземпляры уведомлялись всякий раз при создании их собрата:

```
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}: received event from {sender}")

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"
```

Следующая интерактивная сессия показывает, как новые экземпляры класса `SelfWatch` уведомляют уже существующие экземпляры класса об инициализации:

```
>>> from topic_based_events import SelfWatch
>>> selfwatch1 = SelfWatch()
>>> selfwatch2 = SelfWatch()
<SelfWatch: 1>: received event from <SelfWatch: 2>
>>> selfwatch3 = SelfWatch()
<SelfWatch: 2>: received event from <SelfWatch: 3>
<SelfWatch: 1>: received event from <SelfWatch: 3>
>>> selfwatch4 = SelfWatch()
<SelfWatch: 2>: received event from <SelfWatch: 4>
<SelfWatch: 3>: received event from <SelfWatch: 4>
<SelfWatch: 1>: received event from <SelfWatch: 4>
```

Библиотека `blinker` имеет и другие интересные особенности.

- *Анонимные сигналы* — пустые вызовы `signal()` всегда создают совершенно новый анонимный сигнал. При хранении сигнала в виде модуля или атрибута класса вы вводите строковые литералы или случайные сигналы конфликтов имен.
- *Субъектная подписка* — метод `signal.connect()` позволяет выбрать определенного отправителя; это дает возможность использовать субъектную диспетчеризацию поверх тематической.
- *Декораторы сигналов* — метод `signal.connect()` можно использовать в качестве декоратора; код становится короче, и обработка становится более очевидной.
- *Данные в сигналах* — метод `signal.send()` принимает произвольные именованные аргументы, которые будут переданы подключенному обработчику; это позволяет использовать сигналы в качестве механизма передачи сообщений.

Еще одна интересная вещь о тематическом стиле событийно-ориентированного программирования заключается в том, что он не создает субъекто-зависимые отношения между компонентами. Исходя из ситуации, обе стороны могут быть генераторами событий и обработчиками. Данный способ обработки событий становится только механизмом связи. Это делает тематическое событийное программирование хорошим выбором для общей архитектуры. Слабосвязанность программных компонентов позволяет вносить небольшие постепенные изменения. Кроме того, процесс приложения, слабо связанный внутренне через систему событий, можно легко разделить на несколько сервисов, которые общаются между собой через

очередь сообщений. Это позволяет трансформировать управляемые событиями приложения в распределенные архитектуры.

В следующем разделе рассмотрим событийно-ориентированные архитектуры.

Событийно-ориентированные архитектуры

От событийно-ориентированных приложений остается сделать всего шаг до событийно-ориентированной архитектуры. Событийно-ориентированное программирование позволяет разделить приложение на отдельные компоненты, взаимодействующие друг с другом путем передачи событий или сигналов. Если вы уже сделали это, то должны также иметь возможность разделить приложение на отдельные сервисы, которые делают то же самое, но передают события друг другу с помощью какого-либо механизма IPC или по сети.

Событийно-ориентированные архитектуры переводят концепцию событийного программирования на уровень межсервисной связи. Архитектуры все-таки заслуживают внимания по многим причинам.

- *Масштабируемость и использование ресурсов.* Если ваша рабочая нагрузка может быть разделена на множество не зависящих от порядка событий, то событийно-ориентированные архитектуры позволяют легко распределять работу по множеству вычислительных узлов (хостов). Объем вычислительной мощности также может динамически подстраиваться под количество событий, обрабатываемых в системе в настоящее время.
- *Слабая связь.* Системы, состоящие из множества маленьких (предпочтительно) сервисов, которые обмениваются данными через очередь, как правило, более свободны по сравнению с монолитным программным обеспечением. Слабосвязанность позволяет легче вносить постепенные изменения и дорабатывать архитектуру системы.
- *Отказоустойчивость.* Событийно-ориентированные системы с собственным механизмом передачи события (распределенные очередями сообщений), как правило, более устойчивы к переходным проблемам. Современные очереди сообщений, такие как Kafka или RabbitMQ, предлагают несколько способов обеспечения того, что сообщение всегда будет доставлено по меньшей мере одному получателю и повторно доставлено в случае непредвиденных ошибок.

Событийно-ориентированные архитектуры лучше подходят для задач, которые могут решаться в асинхронном режиме, таких как обработка файлов или передача файлов/электронной почты, или для систем, работающих с регулярными и/или запланированными событиями. В Python это будет способ преодоления ограничений производительности интерпретатора CPython (наподобие GIL, о котором шла речь в главе 15).

И что не менее важно: событийно-ориентированные архитектуры в некотором роде сродни бессерверным вычислениям. В этой модели облачных вычислений не нужно думать об инфраструктуре и тратиться на вычислительные мощности. Все проблемы масштабирования и управления инфраструктурой ложатся на вашего провайдера облачных сервисов, а вы передаете ему лишь код. Часто цены таких сервисов зависят только от ресурсов, которые использует ваш код. Наиболее важная категория бессерверных вычислительных сервисов — модель *Function as a Service (FAAS)*, которая выполняет небольшие блоки кода (функции) в ответ на события.

В следующем подразделе мы обсудим очереди событий и сообщений.

Очереди событий и сообщений

Большинство однопроцессных реализаций событийно-ориентированного программирования обрабатываются в последовательном режиме. Будь то основанное на обратных вызовах приложение GUI или полноценное сигнальное программирование с `blinker` — событийное приложение обычно поддерживает какой-либо вид отображения между событиями и ведет списки обработчиков выполнения всякий раз, когда происходит одно из этих событий.

Этот стиль передачи информации в распределенных приложениях, как правило, осуществляется через связь «запрос — ответ» — двунаправленный и, очевидно, синхронный способ связи между сервисами. Он определенно может быть основой для простой обработки событий, но имеет много недостатков, которые делают метод неэффективным в крупномасштабных или сложных системах. Самая большая проблема связи «запрос — ответ» заключается в том, что появляется высокая степень связанности между компонентами.

- Каждый компонент должен иметь возможность находить зависимые сервисы. Другими словами, генераторы событий должны знать сетевые адреса сетевых обработчиков.
- Подписка происходит непосредственно в сервисе, который генерирует событие. Это значит, что в целях создания нового соединения событий нужно изменить больше одного сервиса.
- Обе стороны связи должны договориться об используемом протоколе связи и формате сообщений. Это усложняет возможные изменения.
- Сервис, генерирующий события, должен обрабатывать возможные ошибки, которые возвращаются в ответах от зависимых сервисов.
- Связь «запрос — ответ» часто плохо обрабатывается в асинхронном режиме. Это значит, что событийная архитектура, построенная на такой связи, редко выигрывает от параллельной обработки потоков.

Ввиду этих причин событийно-ориентированные архитектуры, как правило, реализуются с использованием понятия очередей сообщений, а не циклов «запрос — ответ». Очередь сообщений представляет собой механизм связи в виде отдельного сервиса или библиотеки, которая заинтересована только в сообщениях и предполагаемом механизме доставки. Мы уже упоминали практический пример применения очередей сообщений в подразделе «Использование очереди задач и отложенная обработка» раздела «Использование архитектурных компромиссов» главы 14.

Очереди сообщений позволяют обеспечить слабосвязанность сервисов, поскольку изолируют генераторы событий и обработчики. Генераторы публикуют сообщения непосредственно в очередь и не думают о том, отслеживает ли кто-то эти события. Аналогично обработчики берут события непосредственно из очереди и не думают о том, кто произвел события (иногда информация о генераторе имеет значение, но в таких случаях содержится в сообщении или принимает участие в механизме маршрутизации сообщений). В подобном потоке связи никогда не возникает прямая синхронная связь между генераторами событий и обработчиками и вся информация проходит через очередь.

В ряде случаев подобное разделение может быть доведено до такой крайности, что один сервис может сам взаимодействовать с внешним механизмом очереди. В этом нет ничего удивительного, поскольку очереди сообщений — уже отличный способ межпоточной связи, который позволяет избежать блокировки (см. главу 15).

Помимо слабосвязанности, очереди сообщений (особенно в виде выделенных сервисов) имеют множество дополнительных возможностей.

- Большинство очередей сообщений способны обеспечить удержание сообщений. Это значит, что, даже если сервис очереди сообщений «упадет», сообщения не потеряются.
- Многие очереди сообщений поддерживают доставку сообщений/обработку подтверждений и позволяют определить механизм повтора для сообщений, которые не получается доставить. Это в совокупности с удержанием сообщений гарантирует, что если сообщение было успешно отправлено, то рано или поздно обработается даже в случае сбоя сети или сервиса.
- Очереди сообщений по природе своей параллельны. Благодаря различной семантике распределения сообщений это отличная основа для хорошо масштабируемой и распределенной архитектуры.

Когда дело доходит до фактической реализации очереди сообщений, можно выделить две основные архитектуры.

- **Очереди с брокером.** В этой архитектуре существует сервис (или кластер сервисов), который отвечает за принятие и распространение события. Наиболее часто встречающийся пример системы очереди сообщений с открытым

исходным кодом — *RabbitMQ* и *Apache Kafka*. Популярный облачный сервис — *Amazon SQS*. Эти типы систем наиболее эффективны с точки зрения удержания сообщения и встроенной семантики доставки сообщений.

- **Очереди без брокера.** Реализованы исключительно в качестве библиотек программирования. Ведущая и наиболее популярная библиотека сообщений без брокера — *ZeroMQ* (часто пишется *ØMQ*). Самое большое преимущество сообщений без брокера — эластичность. В них простота эксплуатации (без дополнительного централизованного обслуживания или кластера серисов) пожертвована в угоду полноте функций (такие вещи, как удержание и сложная доставка сообщений, осуществляются внутри сервисов).

Оба подхода к обмену сообщениями имеют преимущества и недостатки. В очереди сообщений с брокером всегда есть дополнительный сервис (в случае очередей с открытым исходным кодом, работающих в собственной инфраструктуре) или дополнительный пункт в счет-фактуре (в случае облачных сервисов). Такие системы обмена сообщениями быстро стали важной частью вашей архитектуры. Если сервис перестает работать, то все ваши системы останавливаются из-за межсервисной связи. В результате вы получите систему, где все доступно прямо «из коробки» и остается лишь сделать правильную конфигурацию и несколько вызовов API.

В случае очередей без брокера связь часто бывает более распределенной. То, что в коде кажется простой публикацией события в какой-то абстрактный канал, часто оказывается абстракцией на уровне кода для одноранговой связи, которая скрыто происходит в библиотеке обмена сообщениями без брокера. Это значит, что архитектура системы не зависит от единого сервиса обмена сообщениями или кластера. Даже если некоторые сервисы ломаются, остальные части системы все еще могут взаимодействовать друг с другом. Недостаток этого подхода заключается в том, что вы остаетесь в одиночестве, когда речь идет о таких вещах, как удержание и доставка сообщений/обработка подтверждений или попытки повторной доставки. Если вам нужно все перечисленное, то придется реализовывать такие возможности непосредственно в вашем сервисе или создать собственный брокер обмена сообщениями с помощью библиотеки без брокера.

Резюме

В текущей главе мы рассмотрели элементы событийно-ориентированного программирования. Мы привели наиболее распространенные примеры и приложения данного подхода к программированию, чтобы лучше представить себе эту парадигму. Затем мы описали три основных стиля данного программирования: на основе обратных вызовов, субъекта и темы. Существует много событийно-ориентированных паттернов проектирования и методов программирования, но все они так или иначе

попадают в одну из этих трех категорий. В последней части главы мы сосредоточились на событийно-ориентированных архитектурах программирования.

Поскольку мы приближаемся к концу книги, вы, вероятно, заметили, что чем дальше, тем меньше мы говорим о Python как таковом. В этой главе мы обсудили элементы событийно-ориентированного и сигнального программирования, но мало говорили о самом языке. Мы затронули некоторые примеры кода на Python, но лишь для того, чтобы представить понятие парадигмы событийно-ориентированного программирования.

Дело в том, что Python — как и любой другой язык программирования — просто инструмент. Чтобы быть хорошим программистом, нужно хорошо знать свой инструмент, но простого знания языка и его библиотек здесь недостаточно. Именно поэтому мы начали с прикладных тем, таких как современные особенности синтаксиса, рекомендации по упаковке и стратегиям развертывания, а затем постепенно перешли на более абстрактные темы, зависящие от конкретного языка.

В следующей главе мы сохраним данную тенденцию и обсудим некоторые полезные паттерны проектирования. На сей раз мы представим намного больше кода, но это не будет иметь значения, поскольку шаблоны проектирования по определению не зависят от конкретного языка. Именно *независимость от языка* делает паттерны одной из самых распространенных тем в книгах о программировании. Тем не менее мы надеемся, что вам не будет скучно, так как мы постараемся сосредоточиться только на паттернах, актуальных для разработчиков на Python.

17 Полезные паттерны проектирования

Паттерн проектирования — многоразовое, местами зависящее от языка программирования, решение некоторой проблемы, часто встречающейся при разработке ПО. Самая популярная книга по этой теме — *Design Patterns: Elements of Reusable Object-Oriented Software*, *Addison-Wesley Professional* Эрика Гаммы, Джона Влиссидеса, Ральфа Джонсона и Ричарда Хелма, также известных как *«Банда четырех»* (Gang of Four, GoF). Данная книга считается одной из самых значимых в этой области и содержит описание 23 паттернов проектирования с примерами на Smalltalk и C++.

Во время написания кода приложения эти паттерны помогают решать общие проблемы. Они знакомы всем разработчикам, поскольку описывают проверенные парадигмы разработки. Однако изучать паттерны следует с учетом используемого языка, ввиду того что некоторые из них в тех или иных языках либо не применяются, либо уже встроены.

В этой главе описываются наиболее полезные паттерны в Python и интересные для обсуждения паттерны, дополненные примерами реализации. Ниже приведены три раздела, соответствующие категориям паттернов, определенным GoF:

- порождающие паттерны — используются для создания объектов с конкретным поведением;
- структурные паттерны — помогают структурировать код для конкретных случаев применения;
- поведенческие паттерны — помогают распределить обязанности и инкапсулировать поведение.

В этой главе вы узнаете, как выглядят наиболее распространенные конструкции и как реализовать их на Python. Вы также научитесь распознавать проблемы, которые можно успешно решить с помощью этих паттернов, чтобы улучшить вашу архитектуру приложений и общее удобство сопровождения ПО.

Технические требования

Файлы кода для этой главы можно найти по ссылке github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter17.

Порождающие паттерны

Порождающие паттерны имеют дело с механизмом инстанцирования объекта. Такой паттерн определяет способ создания экземпляров объекта или даже построения классов.

Эти паттерны очень важны в компилируемых языках, таких как C или C++, поскольку в них сложнее генерировать типы по требованию во время выполнения.

Но в Python создать новые типы во время выполнения довольно просто. Встроенная функция `type` позволяет определить новый тип объекта:

```
>>> MyType = type('MyType', (object,), {'a': 1})
>>> ob = MyType()
>>> type(ob)
<class '__main__.MyType'>
>>> ob.a
1
>>> isinstance(ob, object)
True
```

Классы и типы встроены. Мы уже работали с созданием новых классов, и вы можете взаимодействовать с классом и генерацией объекта, используя *метаклассы*. Эти функции лежат в основе реализации паттерна *«Фабрика»*, но мы не будем описывать его в данном разделе, поскольку широко освещали тему создания классов и объектов в главе 3.

Помимо «Фабрики», единственный интересный порождающий паттерн от GoF с точки зрения Python — *«Синглтон»*.

Синглтон

Синглтон ограничивает создание экземпляров класса только одним экземпляром.

Паттерн гарантирует, что у данного класса всегда существует лишь один экземпляр. Это можно использовать, например, когда вы хотите ограничить доступ ресурса только к одному контексту памяти в данном процессе. Так, класс соединителя базы данных может быть синглтоном, работающим с синхронизацией и управляющим данными в памяти. Предполагается, что никакой другой экземпляр не взаимодействует в этот момент с БД.

Этот паттерн может значительно упростить способ обработки параллелизма в приложении. Утилиты, которые дают функции, общие для всего приложения,

часто объявляются именно синглтонами. Например, в веб-приложениях класс, отвечающий за резервирование уникального идентификатора документа, делается таким, поскольку должна быть только одна утилита, которая выполняет эту работу.

Существует популярная полуидиома создания синглтонов в Python с помощью переопределения метода класса `__new__()`:

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls, *args, **kwargs)

        return cls._instance
```

Попытавшись создать несколько экземпляров этого класса и сравнить их идентификаторы, вы увидите, что все они представляют собой один и тот же объект:

```
>>> instance_a = Singleton()
>>> instance_b = Singleton()
>>> id(instance_a) == id(instance_b)
True
>>> instance_a == instance_b
True
```

Мы называем это полуидиомой, поскольку она в некотором смысле опасна. Проблемы начинаются, когда вы пытаетесь сделать подкласс синглтона и создать экземпляр этого нового подкласса, если уже создали экземпляр базового класса:

```
>>> class ConcreteClass(Singleton): pass
...
>>> Singleton()
<Singleton object at 0x000000000306B470>
>>> ConcreteClass()
<Singleton object at 0x000000000306B470>
```

Проблема усугубится, когда вы заметите, что такое поведение связано с порядком создания экземпляра. В зависимости от вашего порядка использования класса вы можете получить или нет один и тот же результат. Посмотрим, каковы будут результаты, если вы сначала создадите экземпляр подкласса, а затем — базового класса:

```
>>> class ConcreteClass(Singleton): pass
...
>>> ConcreteClass()
<ConcreteClass object at 0x00000000030615F8>
>>> Singleton()
<Singleton object at 0x000000000304BCF8>
```

Видно, что поведение получилось совершенно другим и труднопредсказуемым. В больших приложениях это может привести к проблемам и трудностям с отладкой. В зависимости от контекста выполнения вы можете задействовать или нет желаемые классы. Поскольку такое поведение очень трудно прогнозировать и контролировать, приложение может выйти из строя из-за измененного порядка импорта или даже пользовательского ввода. Но если у вашего синглтона не будет подклассов, то подобная реализация будет относительно безопасной. Но это бомба замедленного действия. Все может пойти крахом, если некто забудет о данном риске в будущем и решит создать подкласс из вашего синглтона. Следовательно, лучше не применять эту конкретную реализацию и попробовать другую.

Намного более безопасным будет задействовать более продвинутую технику — *метаклассы*. Переопределяя метод `__call__()` метакласса, вы можете повлиять на создание пользовательских классов. Это позволяет получить многоразовый синглтон:

```
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]
```

Применяя `Singleton` как метакласс для пользовательских классов, вы можете получить однокомпонентный класс, безопасный для подклассов и не зависящий от порядка создания экземпляров:

```
>>> ConcreteClass() == ConcreteClass()
True
>>> ConcreteSubclass() == ConcreteSubclass()
True
>>> ConcreteClass()
<ConcreteClass object at 0x000000000307AF98>
>>> ConcreteSubclass()
<ConcreteSubclass object at 0x000000000307A3C8>
```

Другой способ решения проблемы тривиальной реализации синглтона — использовать метод, предложенный Алексом Мартелли. Он придумал нечто похожее в поведении на синглтон, но совершенно другое по структуре. Это не классический паттерн из книги «Банды четырех», однако он распространен среди разработчиков на Python. Он называется `Borg` или `Monostate`.

Идея довольно проста. В паттерне «Синглтон» важно не количество живых экземпляров, а тот факт, что все они должны иметь одинаковое состояние в любой момент времени. Так, Алекс Мартелли придумал класс, который делает все экземпляры класса одинаковыми:

```
class Borg(object):
    _state = {}

    def __new__(cls, *args, **kwargs):
        ob = super().__new__(cls, *args, **kwargs)
        ob.__dict__ = cls._state
        return ob
```

Это устраняет вопрос подклассов, но все еще зависит от того, как работает код подкласса. Например, если переопределяется метод `__getattr__`, то паттерн будет нарушен.

Тем не менее синглтоны не должны иметь несколько уровней наследования. Класс, отмеченный как синглтон, уже определен.

Тем не менее данный паттерн рассматривается многими разработчиками как способ борьбы с уникальностью в приложении. Если нужен синглтон, то почему бы вместо него не использовать модуль с функциями, учитывая, что модуль Python уже является синглтоном? Наиболее распространенная схема — определение переменной уровня модуля как экземпляра класса, который должен быть синглтоном. Благодаря этому вы также не ограничиваете разработчиков в изначальном проекте.

Однокомпонентные классы — неявное средство борьбы с уникальностью в приложении. Без них можно обойтись. Если вы не работаете в похожем на Java фреймворке, в котором нужен такой паттерн, то стоит использовать модуль вместо класса.

В следующем разделе посмотрим на структурные паттерны.

Структурные паттерны

Структурные паттерны очень важны в больших приложениях. Они решают, как организован код, и дают разработчику готовый способ взаимодействия с каждой частью приложения.

В течение долгого времени наиболее известная реализация многих структурных паттернов в мире Python делалась в проекте Zope с *Zope Component Architecture (ZCA)*. В нем реализуется большинство паттернов, описанных в этом разделе, и предоставлен богатый набор инструментов для работы с ними. ZCA предназначен для работы не только в рамках Zope, но и других структур, например Twisted. В нем возможна реализация интерфейсов и адаптеров между разными элементами. К сожалению (или нет), проект Zope уже покинул Олимп и сейчас не пользуется прежней популярностью. Но его ZCA все еще можно использовать как справочник по реализации структурных паттернов в Python. Байджу Мутукадан

(Baiju Muthukadan) написал книгу по работе с Zope под названием *Comprehensive Guide to Zope Component Architecture*. Она доступна и в печатном виде, и бесплатно в Сети (http://muthukadan.net/docs/zca.html). Книга была написана в 2009 году, поэтому неактуальна для последних версий Python, но все равно достаточно полезна, поскольку содержит обоснования для некоторых упомянутых паттернов.

В синтаксисе Python уже есть некоторые из популярных структурных паттернов. Например, декораторы класса и функций можно рассматривать как варианты *паттерна «Декоратор»*. Кроме того, поддержка создания и импорта модулей реализована в *паттерне «Модуль»*.

Перечень общих структурных паттернов довольно длинный. В оригинальной книге *Design Patterns* перечислено семь из них, а в другой литературе можно найти и другие. Мы не будем обсуждать все из них, но остановимся лишь на трех наиболее популярных и признанных из них:

- «Адаптер»;
- «Заместитель»;
- «Фасад».

Рассмотрим эти структурные паттерны в следующих подразделах.

Адаптер

Паттерн *«Адаптер»* позволяет создавать интерфейс, через который существующий класс может использоваться из другого интерфейса. Другими словами, адаптер оборачивает класс или объект `A` так, что он работает в контексте, предназначенном для класса или объекта `B`.

Создавать адаптеры в Python очень просто, и это связано с принципами типизации в данном языке. Философию типизации в Python обычно называют утиной типизацией: *«Если что-то ходит как утка и говорит как утка — значит, это утка!»*

Согласно этому правилу, если принимается значение функции или метода, то решение должно быть основано не на его типе, а на интерфейсе. Таким образом, до тех пор, пока объект ведет себя ожидаемым образом, его тип считается совместимым. Эта сентенция полностью отличается от многих статически типизированных языков, где такое встречается редко.

На практике если некий код предназначен для работы с данным классом, то можно подавать ему объекты из другого класса при наличии соответствующих методов и атрибутов. Конечно, это предполагает, что код не проверяет принадлежность `instance` к определенному классу напрямую.

Адаптер основан на данной философии и определяет механизм оборачивания, где класс или объект оборачивается так, чтобы работать в контексте, который

не был для него предназначен. Типичный пример — `StringIO`, поскольку адаптирует тип `str` таким образом, что его можно использовать как тип `file`:

```
>>> from io import StringIO
>>> my_file = StringIO('some content')
>>> my_file.read()
'some content'
>>> my_file.seek(0)
>>> my_file.read(1)
's'
```

Возьмем другой пример. Класс `DublinCoreInfos` знает, как отобразить сводку подмножества информации Dublin Core (dublincore.org) для данного документа, представленного как `dict`. Он считывает несколько полей, таких как имя автора или название, и затем выводит их. Чтобы иметь возможность отображать Dublin Core для файла, его необходимо адаптировать так же, как это делает `StringIO`. На рис. 17.1 показана UML-подобная схема для реализации такого рода адаптера.

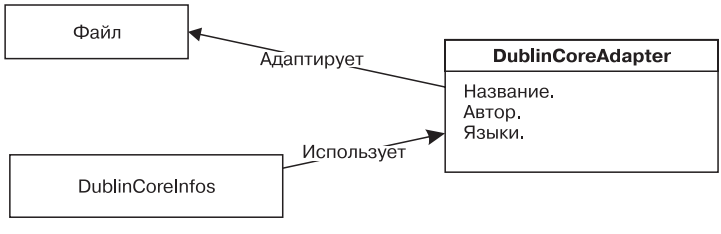


Рис. 17.1. UML-схема для простого примера адаптера

`DublinCoreAdapter` оборачивает экземпляр `file` и предоставляет доступ к метаданным через него:

```
from os.path import split, splitext class DublinCoreAdapter: def
__init__(self, filename): self._filename = filename @property def
title(self): return splitext(split(self._filename)[-1])[0] @property def
languages(self): return ('en',) def __getitem__(self, item): return
getattr(self, item, 'Unknown') class DublinCoreInfo(object): def
summary(self, dc_dict): print('Title: %s' % dc_dict['title'])
print('Creator: %s' % dc_dict['creator']) print('Languages: %s' % ',
'.join(dc_dict['languages']))
```

А вот пример использования:

```
>>> adapted = DublinCoreAdapter('example.txt')
>>> infos = DublinCoreInfo()
>>> infos.summary(adapted)
Title: example
Creator: Unknown
Languages: en
```

Помимо того что такой паттерн позволяет выполнять замену, адаптер может в целом изменить способ действий разработчика. Адаптировать объект к работе в определенном контексте — значит предполагать, что реальный класс объекта вообще не имеет значения. Важно то, что данный класс реализует поведение, ожидаемое `DublinCoreInfo`, и оно фиксируется или выполняется адаптером. Таким образом, код может просто сообщать, совместим ли с объектами, которые реализуют конкретное поведение. Это делается с помощью интерфейсов, и мы поговорим о них далее.

Интерфейсы

Интерфейс — это определение API. Он описывает список методов и атрибутов, которые класс должен иметь, чтобы реализовать определенное поведение. Это описание не реализует какой-либо код, а лишь определяет своего рода требования для любого класса, который хочет реализовать интерфейс. Любой класс может реализовать один или несколько интерфейсов.

В Python утиная типизация преобладает над явными определениями интерфейса, но иногда бывает лучше использовать именно второй вариант. Например, явное определение интерфейса позволяет фреймворку определить функциональные возможности через интерфейсы.

Преимущество состоит в том, что классы слабо связаны, а это считается хорошей практикой. Например, чтобы выполнить данный процесс, класс A зависит не от класса B, а от интерфейса I. Класс B реализует I, но сам может быть любым классом.

Поддержка подобного метода встроена во многих статически типизированных языках, таких как Java или Go. Интерфейсы позволяют функциям или методам ограничивать диапазон допустимых объектов параметров, реализующих данный интерфейс, независимо от того, из какого класса он исходит. Это обеспечивает бо́льшую гибкость, чем ограничение аргументов конкретными типами или подклассами. Получается явная версия утиной типизации: Java выполняет проверку интерфейсов на предмет безопасности типов во время компиляции, а не с помощью утиной типизации, чтобы связать все воедино во время выполнения.

В Python совершенно иная философия типизации, поэтому в нем нет встроенной поддержки интерфейсов. Во всяком случае, если хотите получить более явный контроль над интерфейсами приложений, то вашему выбору предлагается два решения:

- использовать сторонний фреймворк, который вводит понятие интерфейса;
- применить некоторые расширенные возможности языка для построения методики обработки интерфейсов.

Посмотрим на некоторые из решений.

Использование zope.interface. Существует несколько структур, которые позволяют создавать в Python явные интерфейсы. Наиболее известная из них является частью проекта Zope — пакет `zope.interface`. Сегодня этот проект не так популярен, как раньше, но данный пакет все еще лежит в основе фреймворка Twisted.

Ключевой класс пакета `zope.interface` — это класс `Interface`. Он позволяет явно определить новый интерфейс через подкласс. Предположим, что мы хотим определить обязательный интерфейс для каждой реализации прямоугольника:

```
from zope.interface import Interface, Attribute

class IRectangle(Interface):
    width = Attribute("The width of rectangle")
    height = Attribute("The height of rectangle")

    def area():
        """Возвращаем площадь прямоугольника
        """

    def perimeter():
        """Возвращаем периметр прямоугольника
        """
```

При определении интерфейсов с помощью `zope.interface` важно помнить несколько моментов:

- общее соглашение об именах для интерфейсов — использование буквы `I` в качестве префикса имени;
- методы интерфейса не должны принимать параметр `self`;
- поскольку интерфейс не дает конкретную реализацию, он должен состоять только из пустых методов. Вы можете использовать оператор `pass`, поднимать ошибку `NotImplementedError` или добавлять `docstring` (предпочтительно именно так);
- интерфейс может также задавать необходимые атрибуты с помощью класса `Attribute`.

Если у вас определено такое соглашение, то вы можете определить новые конкретные классы, которые обеспечивают реализацию для нашего интерфейса `IRectangle`. Для этого вам нужно использовать декоратор класса `implementor()` и реализовать все определенные методы и атрибуты:

```
@implementer(IRectangle)
class Square:
    """Конкретная реализация квадрата с интерфейсом прямоугольника
    """
```

```
    def __init__(self, size):
        self.size = size

    @property
    def width(self):
        return self.size

    @property
    def height(self):
        return self.size

    def area(self):
        return self.size ** 2

    def perimeter(self):
        return 4 * self.size

@implementer(IRectangle)
class Rectangle:
    """Конкретная реализация прямоугольника
    """
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return self.width * 2 + self.height * 2
```

Обычно говорят, что интерфейс определяет соглашение о том, что должна выполнить конкретная реализация. Основное преимущество этого паттерна заключается в возможности проверить согласованность между соглашением и реализацией, прежде чем используется объект. Обычный подход утиной типизации позволяет лишь найти несоответствие, когда во время выполнения отсутствует нужный атрибут или метод. А `zope.interface` дает возможность просмотреть фактическую реализацию с помощью двух методов модуля из `zope.interface.verify`, чтобы найти несогласованность на ранней стадии:

- `verifyClass(interface, class_object)` — проверяет объект класса на предмет существования методов и правильности их сигнатур, не глядя при этом на атрибуты;
- `verifyObject(interface, instance)` — проверяет методы, их сигнатуры, а также атрибуты фактического экземпляра объекта.

Поскольку мы определили наш интерфейс и две конкретных реализации, проверим соглашения в интерактивном режиме:

```
>>> from zope.interface.verify import verifyClass, verifyObject
>>> verifyObject(IRectangle, Square(2))
True
>>> verifyClass(IRectangle, Square)
True
>>> verifyObject(IRectangle, Rectangle(2, 2))
True
>>> verifyClass(IRectangle, Rectangle)
True
```

Ничего впечатляющего. Классы `Rectangle` и `Square` тщательно выполняют соглашение, и мы не увидим ничего интересного, кроме успешной проверки. Но что происходит, если мы делаем ошибку? Рассмотрим пример двух классов, которые не смогут сделать полную реализацию интерфейса `IRectangle`:

```
@implementer(IRectangle)
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
@implementer(IRectangle)
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

У класса `Point` нет методов или атрибутов интерфейса `IRectangle`, поэтому его проверка покажет несоответствия на уровне класса:

```
>>> verifyClass(IRectangle, Point)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "zope/interface/verify.py", line 102, in verifyClass
    return _verify(iface, candidate, tentative, vtype='c')
  File "zope/interface/verify.py", line 62, in _verify
    raise BrokenImplementation(iface, name)
    zope.interface.exceptions.BrokenImplementation: An object has failed to
    implement interface <InterfaceClass __main__.IRectangle>
         The perimeter attribute was not provided.
```

С классом `Circle` проблем будет больше. У него определены все методы интерфейса, но соглашение нарушается на уровне атрибутов экземпляра. Именно по этой причине в большинстве случаев вам нужно использовать функцию `verifyObject()`, чтобы полностью проверить реализацию интерфейса:

```
>>> verifyObject(IRectangle, Circle(2))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "zope/interface/verify.py", line 105, in verifyObject
    return _verify(iface, candidate, tentative, vtype='o')
  File "zope/interface/verify.py", line 62, in _verify
    raise BrokenImplementation(iface, name)
    zope.interface.exceptions.BrokenImplementation: An object has failed to
    implement interface <InterfaceClass __main__.IRectangle>
         The width attribute was not provided.
```

Кроме того, модуль `zope.inteface` позволяет разъединять приложения. Он дает возможность применять соответствующие интерфейсы объектов, не слишком усложняя множественное наследование, а также позволяет заранее найти несоответствия. Но самый большой недостаток этого подхода — необходимость явно определить для проверки, что данный класс следует правилам некоего интерфейса. Это особенно трудно, если вам нужно проверить экземпляры, поступающие из внешних классов встроенных библиотек. В `zope.interface` есть решения для данной проблемы, и вы можете, конечно, решать такие проблемы по своему усмотрению, используя паттерн «Адаптер» или даже обезьяний патч. Во всяком случае, простота подобных решений является по крайней мере спорной.

Использование аннотаций функций и абстрактных базовых классов. Паттерны проектирования призваны облегчать решение проблем, а не создавать лишние сложности. Модуль `zope.interface` — весьма эффективный и может отлично дополнить некоторые проекты, но в то же время не является панацеей. Используя его, вы обнаружите, что много времени уходит на решение проблем несовместимости интерфейсов сторонних классов и нескончаемых слоев адаптеров, а писать саму фактическую реализацию будет некогда. Если вы чувствуете, что началось нечто подобное, то, значит, что-то пошло не так. К счастью, Python поддерживает создание облегченной альтернативы интерфейсов. Это не полноценное решение, как `zope.interface` или его альтернативы, но в целом обеспечивает более гибкие приложения. Вероятно, вам придется написать немного больше кода, но в конечном итоге вы получите нечто более расширяемое, лучше обрабатывающее внешние типы и, возможно, более перспективное.

Обратите внимание: в Python, по сути, нет четкого понятия об интерфейсах и, вероятно, никогда не будет, но кое-какие функции в данном языке отдаленно их напоминают. В частности, это:

- *абстрактные базовые классы* (abstract base class, ABC);
- *аннотации функций*;
- *аннотации типов*.

Суть нашего решения заключается в использовании абстрактных базовых классов, поэтому начнем разговор с них.

Как вы, наверное, знаете, прямое сравнение типов считается вредным и *не питоническим*. Вы всегда должны избегать сравнений подобного рода:

```
assert type(instance) == list
```

Подобное сравнение типов в функциях или методах полностью уничтожает возможность передавать подтип класса в качестве аргумента функции. Более качественный подход заключается в использовании функции `isinstance()`, которая будет учитывать наследование:

```
assert isinstance(instance, list)
```

Дополнительное преимущество `isinstance()` заключается в том, что вы можете использовать более широкий диапазон типов, чтобы проверить их совместимость. Например, если ваша функция ожидает получить в качестве аргумента какую-либо последовательность, то вы можете сравнить ее со списком основных типов:

```
assert isinstance(instance, (list, tuple, range))
```

Такой способ проверки совместимости типов в некоторых ситуациях эффективен, но все еще неидеален. Он будет работать с любым подклассом списка, кортежа или диапазона, но не подействует, если пользователь передаст что-то ведущее себя так же, как один из этих типов последовательностей, но не наследуется от любого из них. Смягчим требования и скажем, что мы хотим принять в качестве аргумента любой итерируемый объект. Что бы вы сделали? Перечень итерируемых типов довольно длинный: `list`, `tuple`, `range`, `str`, `bytes`, `dict`, `set`, `generators` и др. Перечень применимых встроенных типов велик, и даже если вы перечислите их все, это все равно не поможет работать с пользовательским классом, в котором есть метод `__iter__()`, но наследование идет непосредственно от `object`.

Это как раз та ситуация, где нужны абстрактные базовые классы. ABC — это класс, который не должен давать конкретную реализацию, но вместо этого определяет план класса, пригодного для проверки на совместимость типа. Эта концепция очень похожа на концепцию абстрактных классов и виртуальных методов, известных в языке C++.

Абстрактные базовые классы используются для двух целей:

- проверки полноты реализации;
- проверки на совместимость интерфейса.

Итак, предположим, что хотим определить интерфейс, который гарантирует, что у класса есть метод `push()`. Нам нужно создать новый абстрактный базовый класс с помощью специального метакласса `ABCMeta` и декоратора `abstractmethod()` из стандартного модуля `abc`:

```
from abc import ABCMeta, abstractmethod

class Pushable(metaclass=ABCMeta):

    @abstractmethod
    def push(self, x):
        """ Аргумент отправляется во что бы то ни стало
        """
```

Модуль `abc` тоже предоставляет базовый класс ABC, который можно использовать вместо метакласса:

```
from abc import ABCMeta, abstractmethod

class Pushable(metaclass=ABCMeta):
    @abstractmethod
    def push(self, x):
        """ Аргумент отправляется во что бы то ни стало
        """
```

После этого мы сможем использовать этот класс `Pushable` в качестве базового для конкретной реализации, и он станет защищать нас от конкретизации объектов, которые будут иметь неполную реализацию. Определим класс `DummyPushable`, реализующий все методы интерфейса, и класс `IncompletePushable`, нарушающий соглашение:

```
class DummyPushable(Pushable):
    def push(self, x):
        return


class IncompletePushable(Pushable):
    pass
```

Если вы хотите получить экземпляр `DummyPushable`, то проблем не будет, поскольку он реализует метод `push()`:

```
>>> DummyPushable()
<__main__.DummyPushable object at 0x10142bef0>
```

Но если вы попытаетесь создать экземпляр класса `IncompletePushable`, то получите `TypeError` из-за отсутствия реализации метода `interface()`:

```
>>> IncompletePushable()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class IncompletePushable with
abstract methods push
```

Данный подход — отличный способ обеспечить полноту реализации базовых классов, сохраняя при этом явность, как у модуля `zope.interface`. Экземпляры `DummyPushable`, конечно, являются экземплярами `Pushable`, поскольку `Dummy` — подкласс `Pushable`. Но как насчет других классов, имеющих такие же методы, но не являющихся потомками Pushable? Создадим и проверим:

```
>>> class SomethingWithPush:
...     def push(self, x):
...         pass
...
>>> isinstance(SomethingWithPush(), Pushable)
False
```

Чего-то не хватает. У класса `SomethingWithPush`, конечно, есть совместимый интерфейс, однако он не считается экземпляром `Pushable`. Чего же не хватает? Ответ: метода `__subclasshook__(subclass)`, позволяющего вводить собственную логику в процедуру, которая определяет, является ли объект экземпляром данного класса. К сожалению, эту логику придется придумывать самостоятельно, поскольку создатели ABC не хотели ограничивать разработчиков в перезаписи механизма `isinstance()`. Мы можем делать с ним что угодно, но для этого придется писать шаблонный код.

Ну да, мы можем делать что угодно, но, как правило, разумнее всего следовать общей схеме метода `__subclasshook__()`. Стандартная процедура для проверки набора определенных методов где-то в MRO данного класса приведена ниже:

```
from abc import ABCMeta, abstractmethod

class Pushable(metaclass=ABCMeta):

    @abstractmethod
    def push(self, x):
        """ Аргумент отправляется во что бы то ни стало
        """
    @classmethod
    def __subclasshook__(cls, C):
        if cls is Pushable:
            if any("push" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

Если метод `__subclasshook__()` определен таким образом, то вы можете подтвердить, что экземпляры, которые неявно реализуют интерфейс, также считаются экземплярами интерфейса:

```
>>> class SomethingWithPush:
...     def push(self, x):
...         pass
...
>>> isinstance(SomethingWithPush(), Pushable)
True
```

К сожалению, подобный подход к проверке совместимости типов и полноты реализации не учитывает сигнатуры методов класса. Таким образом, если количество ожидаемых аргументов в реализации отличается, то класс по-прежнему будет совместимым. В большинстве случаев это не проблема, но при надобности иметь такой точный контроль над интерфейсами снова нужен пакет `zope.interface`. Как уже было сказано, метод `__subclasshook__()` не ограничивает вас в добавлении сложности логики функции `isinstance()` и тем самым позволяет достичь именно такого контроля.

Еще две функции, которые дополняют абстрактные базовые классы, связаны с аннотациями и подсказками по типам. Об аннотациях функций мы кратко говорили в главе 2. Они позволяют аннотировать функции и их аргументы произвольными выражениями. Как мы упомянули в главе 2, аннотация — лишь заглушка, не несущая никакого синтаксического смысла. В стандартной библиотеке есть утилита, в которой эта функция создает определенное поведение. Во всяком случае, вы можете использовать ее как удобный и легкий способ сообщить разработчику, какой ожидается интерфейс. Например, рассмотрим интерфейс `IRectangle`, переписанный из `zope.interface`:

```
from abc import (
    ABCMeta,
    abstractmethod,
    abstractproperty
)

class IRectangle(metaclass=ABCMeta):

    @abstractproperty
    def width(self):
        return

    @abstractproperty
    def height(self):
        return

    @abstractmethod
    def area(self):
        """ Возвращаем площадь прямоугольника
        """

    @abstractmethod
    def perimeter(self):
        """ Возвращаем периметр прямоугольника
        """

    @classmethod
    def __subclasshook__(cls, C):
        if cls is IRectangle:
            if all([
                any("area" in B.__dict__ for B in C.__mro__),
```

```
                any("perimeter" in B.__dict__ for B in C.__mro__),
                any("width" in B.__dict__ for B in C.__mro__),
                any("height" in B.__dict__ for B in C.__mro__),
            ]):
                return True
        return NotImplemented
```

Если у вас есть функция, которая работает только с прямоугольниками, скажем `draw_rectangle()`, то вы можете аннотировать ожидаемые аргументы следующим образом:

```
def draw_rectangle(rectangle: IRectange):
    ...
```

Это просто информация для разработчика об ожидаемой информации, и ничего больше. И даже это делается через неформальное соглашение, поскольку, как мы знаем, голые аннотации не содержат никакого синтаксического значения. Но зато они доступны во время выполнения, поэтому мы можем сделать кое-что другое. Ниже приведен пример реализации общего декоратора, который способен проверить интерфейс из аннотации функции, если она предоставляется с помощью абстрактных базовых классов:

```
def ensure_interface(function):
    signature = inspect.signature(function)
    parameters = signature.parameters

    @wraps(function)
    def wrapped(*args, **kwargs):
        bound = signature.bind(*args, **kwargs)
        for name, value in bound.arguments.items():
            annotation = parameters[name].annotation

            if not isinstance(annotation, ABCMeta):
                continue
            if not isinstance(value, annotation):
                raise TypeError(
                    "{} does not implement {} interface"
                    "".format(value, annotation)
                )

        function(*args, **kwargs)

    return wrapped
```

Далее мы сможем создать некий конкретный класс, который неявно реализует интерфейс `IRectangle` (без наследования от `IRectangle`) и обновляет реализацию функции `draw_rectangle()`, чтобы увидеть, как работает решение целиком:

```
class ImplicitRectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height
```

```
    @property
    def width(self):
        return self._width

    @property
    def height(self):
        return self._height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return self.width * 2 + self.height * 2

@ensure_interface
def draw_rectangle(rectangle: IRectangle):
    print(
        "{} x {} rectangle drawing"
        "".format(rectangle.width, rectangle.height)
    )
```

Если мы подадим на вход функции `draw_rectangle()` несовместимый объект, то поднимется ошибка `TypeError` с пояснением:

```
>>> draw_rectangle('foo')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 101, in wrapped
TypeError: foo does not implement <class 'IRectangle'> interface
```

Но если мы будем использовать `ImplicitRectangle` или что-либо еще, напоминающее `IRectangle`, то функция будет работать нормально:

```
>>> draw_rectangle(ImplicitRectangle(2, 10))
2 x 10 rectangle drawing
```

Это наш пример реализации `ensure_interface()` на основе декоратора `type-checked()` из проекта `typeannotations`, призванный обеспечить проверку возможностей во время выполнения (github.com/ceronman/typeannotations). Его исходный код содержит интересные идеи о том, как обрабатывать аннотации типов для проверки интерфейса во время выполнения.

Последняя функция, которая может использоваться в дополнение к этому паттерну, — подсказки по типам. Они подробно описаны в PEP 484 и появились в языке совсем недавно. Подсказки по типам есть в новом модуле `typing` и доступны начиная с Python 3.5. Они построены на аннотациях функций, и в них используется немного забытый синтаксис Python 3. Предназначены для указания типов и проверки различных будущих типов Python. Модуль `typing` и документ

PEP 484 направлен на создание стандартной иерархии типов и классов, которые следует применять для описания аннотаций типов.

Тем не менее подсказки по типам не кажутся чем-то революционным, поскольку эта функция не идет в комплекте с другими механизмами проверки, встроенными в стандартную библиотеку. Если вы хотите использовать проверку типов или обеспечить строгую совместимость интерфейса в коде, то нужно интегрировать сторонние библиотеки. Именно поэтому мы не будем копаться в деталях PEP 484. Во всяком случае, подсказки по типам и связанные с ними документы стоят упоминания, поскольку в случае появления какого-то необычного решения в сфере проверки типов в Python, весьма вероятно, в его основе будет лежать PEP 484.

Использование collections.abc. *Абстрактные базовые классы (ABC)* — своего рода небольшие строительные блоки для создания абстракции более высокого уровня. Они позволяют реализовать пригодные для использования интерфейсы, но имеют очень общий характер и предназначены для более крупных задач, чем отдельно взятый паттерн проектирования. Вы можете полностью раскрыть свой творческий потенциал и творить магию, но для создания таких крупных и в то же время действительно полезных вещей требуется много усилий. И не факт, что они вообще окупятся.

Из-за этого пользовательские абстрактные базовые классы применяются редко. И все же в модуле `collections.abc` есть много предопределенных абстрактных классов, которые обеспечивают совместимость типов с общими интерфейсами Python. С помощью базовых классов, имеющихся в этом модуле, вы можете проверить, например, является ли данный объект вызываемым либо отображением или поддерживает итерацию. Использовать эти классы в сочетании с функцией `isinstance()` намного лучше, чем сравнивать с базовыми типами Python. Вам стоит знать, как задействовать эти базовые классы, даже если вы не хотите определять свои собственные интерфейсы с помощью `ABCMeta`.

Наиболее распространенные абстрактные базовые классы из модуля `collections.abc`, которые вы, вероятно, будете использовать, представлены ниже:

- `Container` — интерфейс означает, что объект поддерживает оператор `in` и реализует метод `__contains__()`;
- `Iterable` — интерфейс означает, что объект поддерживает итерации и реализует метод `__iter__()`;
- `Callable` — интерфейс означает, что его можно вызывать как функцию и он реализует метод `__call__()`;
- `Hashable` — интерфейс означает, что объект является `hashable` (то есть может содержаться в множествах и в качестве ключа в словарях) и реализует метод `__hash__()`;

- `Sized` — интерфейс означает, что объект имеет размер (то есть к нему применяется функция `len()` и реализует метод `__len__()`).

Полный список абстрактных базовых классов из модуля `collections.abc` можно посмотреть в официальной документации Python (docs.python.org/3/library/collections.abc.html).

Заместитель

Паттерн *«Заместитель»* позволяет получить опосредованный доступ к «дорогому» или отдаленному ресурсу. Заместитель находится между клиентом и субъектом, как показано на следующей схеме (рис. 17.2).

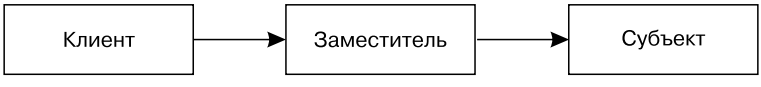


Рис. 17.2. Положение заместителя

Заместители предназначены для оптимизации доступа к субъекту, если он связан с большими затратами ресурсов. Так, например, декораторы `memoize()` и `lru_cache()`, описанные в главе 12, можно считать заместителями.

Заместители также могут применяться для обеспечения интеллектуального доступа к субъекту. Например, большие видеофайлы можно заворачивать в заместитель, чтобы не выгружать их в память, если пользователь запрашивает всего лишь имя файла.

В качестве примера приведем модуль `urllib.request`. `urlopen` — заместитель для контента, расположенного по удаленному URL. После создания заголовки можно запрашивать независимо от самого контента, не читая остальную часть ответа:

```
>>> class Url(object):
...     def __init__(self, location):
...         self._url = urlopen(location)
...     def headers(self):
...         return dict(self._url.headers.items())
...     def get(self):
...         return self._url.read()
...
>>> python_org = Url('http://python.org')
>>> python_org.headers().keys()
dict_keys(['Accept-Ranges', 'Via', 'Age', 'Public-Key-Pins', 'X-Clacks-
Overhead', 'X-Cache-Hits', 'X-Cache', 'Content-Type', 'Content-Length',
'Vary', 'X-Served-By', 'Strict-Transport-Security', 'Server', 'Date',
'Connection', 'X-Frame-Options'])
```

Таким образом, можно определить, была ли изменена страница, не запрашивая при этом саму страницу, а только изменение заголовка. Возьмем пример с большим файлом:

```
>>> ubuntu_iso =
Url('http://ubuntu.mirrors.proxad.net/hardy/ubuntu-8.04-desktop-i386.iso')
>>> ubuntu_iso.headers()['Last-Modified']
'Wed, 23 Apr 2008 01:03:34 GMT'
```

Другой вариант использования заместителя — *уникальность данных*.

Например, рассмотрим сайт, который раздает один и тот же документ в несколько мест. В документ добавляются дополнительные поля для каждого такого расположения плюс счетчик запросов и несколько настроек разрешений. Заместитель в этом случае может работать именно с этими данными, а также указывать на оригинальный документ, не копируя его. Таким образом, данный документ может иметь множество заместителей, и если его контент изменится, то об этом узнают все, независимо от особенностей синхронизации версий.

В целом паттерн «Заместитель» полезен для реализации описания того, что некий объект может находиться где-то в другом месте. Популярные причины такого подхода:

- ускоряется процесс;
- устраняется внешний доступ к ресурсам;
- снижается нагрузка на память;
- гарантируется уникальность данных.

Фасад

Фасад обеспечивает высокоуровневый и простой доступ к подсистеме. Фасад — просто ярлык для определенных функциональных возможностей приложения, работающий в обход остальной подсистемы. Это реализуется, например, путем предоставления функций высокого уровня на уровне пакета.

Фасад, как правило, реализуется в уже существующих системах, где частое использование некоего пакета сосредоточено в функциях высокого уровня. Как правило, такому паттерну не нужны классы и достаточно лишь простых функций в модуле `__init__.py`.

Хороший проект, в котором добавлен фасад для более сложных интерфейсов, — пакет `requests` (docs.python-requests.org). Он действительно упрощает сложную работу с HTTP-запросами и ответами в Python и представляет собой чистый и легкочитаемый API. Это своего рода *«HTTP для людей»*. Такая простота всегда имеет цену, однако возможные компромиссы и дополнительные расходы не пугают разработчиков, которые вовсю используют данный пакет как HTTP-инструмент.

В конце концов, он позволяет ускорить создание проектов, а время разработчика обычно стоит дороже оборудования.

Фасад упрощает применение пакетов. Фасады, как правило, добавляют после нескольких итераций обратной связи от пользователей.

В следующем разделе рассмотрим поведенческие паттерны.

Поведенческие паттерны

Поведенческие паттерны предназначены для упрощения взаимодействия между классами путем структурирования процессов их взаимодействия.

В этом разделе приведены три примера популярных поведенческих паттернов, которые вы можете рассмотреть при написании кода на Python:

- «Наблюдатель»;
- «Посетитель»;
- «Шаблонный метод».

Рассмотрим эти три примера в следующих подразделах.

Наблюдатель

«Наблюдатель» используется для уведомления некоторых объектов об изменении состояния наблюдаемого объекта. Мы уже обсуждали этот паттерн в предыдущей главе, но здесь рассмотрим примеры ситуаций, когда его можно было бы применить.

Например, представим, что у нас есть приложение, которое хранит рекламные материалы (резюме, презентации, видео и листовки) и юридические документы в цифровой форме для нужд отдела продаж крупной компании. У нее много торговых представителей и есть документы, которые надо обновлять. А система выполняет задачи обработки этих цифровых документов и сообщает представителям об обновлениях документов, облегчая им жизнь:

- видеоматериалы преобразуются в файлы различных размеров и конвертируются с помощью портативных аудио- и видеокодеков;
- у PDF-документов генерируются страницы предпросмотра, которые используются в виде значков в CMS-системе компании;
- все новые документы собираются в еженедельный сборник и рассылаются всем сотрудникам отдела продаж;

- новые конфиденциальные материалы шифруются для обеспечения дополнительной безопасности данных;
- пользователи, которые скачали определенные материалы, уведомляются об обновлениях.

Следовательно, почти каждому компоненту системы важно знать о событиях, связанных с жизнью каждого документа. Мы могли бы спроектировать наше приложение таким образом, что каждый компонент получает информацию о внесении в документы изменений. Паттерн «Наблюдатель» в этой ситуации особенно хорош, поскольку именно наблюдатель решает, какие типы событий ему интересны. Затем каждый компонент будет получать уведомления о каждом событии, на которые он подписался. Конечно, для этого весь код, работающий с фактическим состоянием наблюдаемого объекта (например, создание, изменение или удаление документов), должен генерировать такие события. Но это намного легче, чем поддерживать вручную длинный список хуков, которые нужно вызывать каждый раз, когда что-то происходит с наблюдаемым объектом. Популярный веб-фреймворк, поддерживающий этот паттерн программирования, — *Django* с его механизмом сигналов.

Для регистрации наблюдателей в Python на уровне класса может быть реализован класс `Event`:

```
class Event:
    _observers = []

    def __init__(self, subject):
        self.subject = subject

    @classmethod
    def register(cls, observer):
        if observer not in cls._observers:
            cls._observers.append(observer)

    @classmethod
    def unregister(cls, observer):
        if observer in cls._observers:
            cls._observers.remove(observer)

    @classmethod
    def notify(cls, subject):
        event = cls(subject)
        for observer in cls._observers:
            observer(event)
```

Идея в том, что наблюдатели регистрируют себя с помощью метода класса `Event` и получают уведомления от экземпляров `Event`, которые содержат субъект.

Ниже приведен пример конкретного подкласса `Event` с подписанными на него наблюдателями:

```
class WriteEvent(Event):
    def __repr__(self):
        return 'WriteEvent'

def log(event):
    print(
        '{!r} was fired with subject "{}"'
        ''.format(event, event.subject)
    )

class AnotherObserver(object):
    def __call__(self, event):
        print(
            "{!r} trigerred {}'s action"
            "".format(event, self.__class__.__name__)
        )

WriteEvent.register(log)
WriteEvent.register(AnotherObserver())
```

А вот пример результат запуска события с помощью метода `WriteEvent.notify()`:

```
>>> WriteEvent.notify("something happened")
WriteEvent was fired with subject "something happened" WriteEvent trigerred
AnotherObserver's action
```

Эта реализация проста и служит лишь для примера. Чтобы сделать ее полностью работоспособной, нужно добавить следующее:

- разрешить разработчику изменять порядок или события;
- объект `event` должен содержать больше информации.

Разделять свой код — это круто, и паттерн «Наблюдатель» здесь весьма кстати. Он разделяет приложение на модули и делает его более расширяемым. Если вы хотите использовать существующий инструмент, то попробуйте *Blinker*. Он предоставляет быстрый механизм сигналов и трансляции для объектов в Python.

Посетитель

Посетитель полезен для отдельных алгоритмов из структур данных, а цель у него та же, что и у наблюдателя. Он позволяет расширить функциональные возможности данного класса, не меняя его код. Однако посетитель идет немного дальше, определяя класс, который отвечает за хранение данных, и передает в другие классы, называемые *посетителями*. Каждый из них специализируется в одном алгоритме и применяет его к данным. Такое поведение очень похоже на парадигму MVC, где документы

представляют собой пассивные контейнеры, отправляемые на просмотр через контроллеры, или модели содержат данные, измененные с помощью контроллера.

Паттерн `visitor` осуществляется через точку входа в классе данных, в которую можно заходить всем видам посетителей. Класс, предоставляющий свои данные через этот паттерн, будет называться `visitable` (посещаемый), а класс, получающий доступ к данным, — `visitor` (посетитель).

Класс `visitable` решает, как вызывает класс `visitor`, например путем определения вызываемого метода. Так, `visitor`, отвечающий за печать встроенного контента, реализуется через метод `visit_TYPENAME()` метода, и каждый из этих типов может вызвать данный метод в своем методе `accept()`:

```
class VisitableList(list):
    def accept(self, visitor):
        visitor.visit_list(self)

class VisitableDict(dict):
    def accept(self, visitor):
        visitor.visit_dict(self)

class Printer(object):
    def visit_list(self, instance):
        print('list content: {}'.format(instance))

    def visit_dict(self, instance):
        print('dict keys: {}'.format(
            ', '.join(instance.keys()))
        )
```

Это делается следующим образом:

```
>>> visitable_list = VisitableList([1, 2, 5])
>>> visitable_list.accept(Printer())
list content: [1, 2, 5]
>>> visitable_dict = VisitableDict({'one': 1, 'two': 2, 'three': 3})
>>> visitable_dict.accept(Printer())
dict keys: two, one, three
```

Использование такого паттерна означает, что каждый посещаемый класс должен иметь метод `accept`, что не совсем удобно.

Поскольку Python позволяет выполнять самоанализ кода, лучше всего будет автоматически связать посетителей и посещенные классы:

```
>>> def visit(visited, visitor):
...     cls = visited.__class__.__name__
...     method_name = 'visit_%s' % cls
...     method = getattr(visitor, method_name, None)
...     if isinstance(method, Callable):
...         method(visited)
...     else:
...         raise AttributeError(
```

```
...             "No suitable '{}' method in visitor"
...             "".format(method_name)
...         )
...
>>> visit([1,2,3], Printer())
list content: [1, 2, 3]
>>> visit({'one': 1, 'two': 2, 'three': 3}, Printer())
dict keys: two, one, three
>>> visit((1, 2, 3), Printer())
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 10, in visit
AttributeError: No suitable 'visit_tuple' method in visitor
```

Такой паттерн используется в модуле `ast`, например, в классе `NodeVisitor`, который вызывает посетитель после компиляции каждого узла. Все дело в том, что в Python нет оператора сравнения наподобие Haskell.

Еще один пример — механизм просмотра каталога, который вызывает посетители в зависимости от расширения файла:

```
>>> def visit(directory, visitor):
...     for root, dirs, files in os.walk(directory):
...         for file in files:
...             # foo.txt → .txt
...             ext = os.path.splitext(file)[-1][1:]
...             if hasattr(visitor, ext):
...                 getattr(visitor, ext)(file)
...
>>> class FileReader(object):
...     def pdf(self, filename):
...         print('processing: {}'.format(filename))
...
>>> walker = visit('/Users/tarek/Desktop', FileReader())
processing slides.pdf
processing sholl23.pdf
```

Если ваше приложение имеет структуры данных, которые посещаются многими алгоритмами, то посетитель сможет решить часть проблем. Контейнеру данных лучше всего фокусироваться только на предоставлении доступа к данным и их удержании, а обо всем прочем не стоит думать.

Шаблонный метод

Шаблонный метод помогает разработать общий алгоритм, определяющий абстрактные шаги, которые реализуются в подклассах. Он использует *принцип замены Лисков* (Liskov Substitution Principle, LSP), определенный в «Википедии» следующим образом: *«Если S — подтип T, то объекты типа T в программе можно заменить на объекты типа S, не меняя свойств этой программы»*.

Другими словами, абстрактный класс определяет, как работает алгоритм, через шаги, которые реализуются в конкретных классах. Абстрактный класс может также включать базовую или частичную реализацию алгоритма и позволяет разработчикам переопределять его части. Например, некоторые методы класса `Queue` в модуле `queue` можно будет переопределить, чтобы изменить его поведение.

Реализуем пример шаблона для класса, который работает с индексацией текста. `Indexer` — класс `indexer`, обрабатывающий текст в пять шагов. Этот процесс является общим для каждого конкретного метода индексации.

1. Нормализация текста.
2. Разделение текста.
3. Удаление стоп-слов.
4. Стемминг — замена каждого из слов его основой.
5. Подсчет частоты встречаемости слов в тексте.

`Indexer` обеспечивает частичную реализацию алгоритма процесса, но требует реализации методов `_remove_stop_words` и `_stem_words` в подклассе. `BasicIndexer` реализует строгий минимум, в то время как `LocalIndex` использует файл стоп-слов и базу данных однокоренных слов. `FastIndexer` реализует все шаги и может быть основан на быстром индексаторе, таком как *Xapian* или *Lucene*.

Пример простенькой реализации:

```
from collections import Counter

class Indexer:
    def process(self, text):
        text = self._normalize_text(text)
        words = self._split_text(text)
        words = self._remove_stop_words(words)
        stemmed_words = self._stem_words(words)

        return self._frequency(stemmed_words)

    def _normalize_text(self, text):
        return text.lower().strip()

    def _split_text(self, text):
        return text.split()

    def _remove_stop_words(self, words):
        raise NotImplementedError

    def _stem_words(self, words):
        raise NotImplementedError

    def _frequency(self, words):
        return Counter(words)
```

Реализация `BasicIndexer` могла бы выглядеть следующим образом:

```
class BasicIndexer(Indexer):
    _stop_words = {'he', 'she', 'is', 'and', 'or', 'the'}

    def _remove_stop_words(self, words):
        return (
            word for word in words
            if word not in self._stop_words
        )

    def _stem_words(self, words):
        return (
            (
                len(word) > 3 and
                word.rstrip('aeiouy') or
                word
            )
            for word in words
        )
```

И как всегда, пример использования:

```
>>> indexer = BasicIndexer()
>>> indexer.process("Just like Johnny Flynn said\nThe breath I've taken and
the one I must to go on")
Counter({"i'v": 1, 'johnn': 1, 'breath': 1, 'to': 1, 'said': 1, 'go': 1,
'flynn': 1, 'taken': 1, 'on': 1, 'must': 1, 'just': 1, 'one': 1, 'i': 1,
'lik': 1})
```

Шаблоны полезны для алгоритмов, которые можно изменять и делить на изолированные части. Это, вероятно, наиболее часто применяемый паттерн в Python, который не всегда нужно реализовывать с помощью подклассов. Например, множество встроенных функций Python, работающих с алгоритмическими проблемами, принимают аргументы, которые позволяют делегировать часть реализации в стороннюю функцию. Например, функция `sorted()` может иметь необязательный именованный аргумент `key`, который впоследствии используется алгоритмом сортировки. То же самое можно сказать и о функциях `min()` и `max()`.

Резюме

Паттерны проектирования — многоразовые, в некоторой степени зависящие от языка решения общих проблем в создании программного обеспечения. Они являются частью культуры всех разработчиков, независимо от того, на каком языке используются.

Таким образом, полезно иметь под рукой примеры реализации наиболее часто используемых паттернов для данного языка. Во многих источниках (веб-статьи

и книги) вы легко найдете реализацию всех паттернов проектирования из книги «Банды четырех». Именно поэтому мы сконцентрировались только на паттернах, которые являются наиболее распространенными и популярными в контексте языка Python.

Мы рассмотрели три наиболее важные группы паттернов проектирования (порождающие, структурные и поведенческие) и привели практические примеры их реализации. Это должно помочь вам улучшить структуру приложения. Но это не весь список. К счастью, после прочтения нашей книги вы сможете сами более подробно изучить и данную тему, и все остальные моменты языка Python.

Приложение reStructuredText Primer

Здесь мы приведем краткое руководство о том, как использовать reStructuredText (reST) (docutils.sourceforge.net/rst.html). Это простой язык разметки текста, широко используемый сообществом Python для пакетов документов. Самое замечательное в нем то, что текст остается читабельным, поскольку синтаксис разметки не усложняет текст, как, например, LaTeX.

В этом приложении:

- reStructuredText;
- структура раздела;
- списки;
- разметка;
- блоки литералов;
- ссылки.

reStructuredText

Язык reST — часть пакета `docutils`, предоставляющего набор скриптов для преобразования файла reST в различные форматы, такие как HTML, LaTeX, XML или даже S5, слайд-шоу системы Эрика Мейера (meyerweb.com/eric/tools/s5).

Вот пример такого документа:

```
=====
Title
=====

Section 1
=========
This *word* has emphasis.
```

```
Section 2
=========

Subsection
::::::::::

Text.
```

Автор документа может сначала сосредоточиться на содержании, а затем оформить его, в зависимости от своих потребностей. Например, сам Python документирован именно на reST, который затем превращается в HTML и другие различные форматы. Вы можете почитать официальную документацию Python, перейдя по ссылке docs.python.org.

Чтобы начать писать на reST, вам нужно знать по меньшей мере:

- структуру раздела;
- списки;
- разметку;
- блоки литералов;
- ссылки.

Данный раздел представляет собой краткий обзор синтаксиса. Справку можно почитать, перейдя по ссылке docutils.sourceforge.net/docs/user/rst/quickref.html, и это будет хорошим стартом работы с reST.

Чтобы установить reStructuredText, нужно установить `docutils`:

```
$ pip install docutils
```

Например, скрипт `rst2html` из `docutils` выдает HTML-файл из файла reST:

```
$ cat text.txt
Title
=====

content.

$ rst2html.py text.txt
<?xml version="1.0" encoding="utf-8" ?>
(...)
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
(...)
</head>
<body>
<div class="document" id="title">
```

```
<h1 class="title">Title</h1>
<p>content.</p>
</div>
</body>
</html>
```

Рассмотрим все элементы, которые нам следует знать

Структура раздела

Заголовок документа и его разделы подчеркнуты с помощью *не алфавитно-цифровых символов*. Они могут быть подчеркнуты или надчеркнуты, а обычно двойная разметка применяется для заголовка и простое подчеркивание — для разделов.

Наиболее часто используемые символы для подчеркивания заголовка раздела имеют следующий порядок приоритета: `=`, `-`, `_`, `:`, `#`, `+` и `^`.

В случае использования для раздела символ связывается с его уровнем и должен применяться и дальше на протяжении всего документа.

Рассмотрим следующий код:

```
==============
Document title
==============

Introduction to the document content.

Section 1
=========

First document section with two subsections.

Note the ``=`` used as heading underline.

Subsection A
------------
First subsection (A) of Section 1.

Note the ``-`` used as heading underline.

Subsection B
------------
Second subsection (B) of Section 1.

Section 2
=========
Second section of document with one subsection.

Subsection C
------------
Subsection (C) of Section 2.
```

На рис. П.1 показан результат.

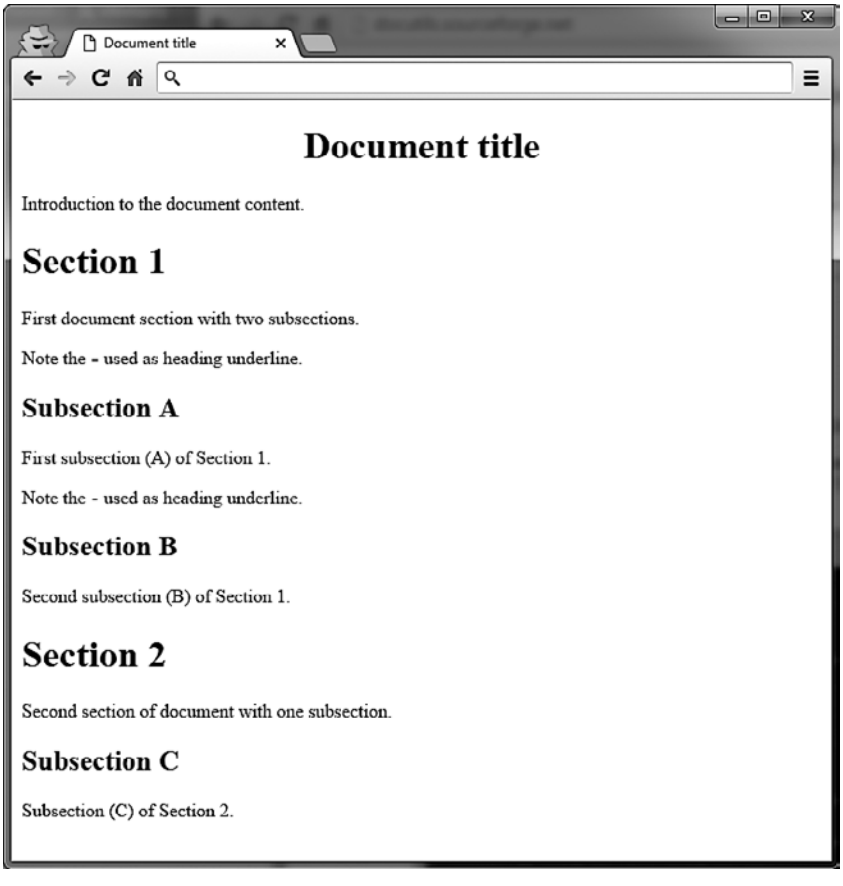


Рис. П.1. reStructuredText после преобразования в HTML и отображения в браузере

Поговорим о списках.

Списки

Язык reST предоставляет читабельный синтаксис следующих списков: маркированных, нумерованных и списков с автонумерацией. Пример показан ниже:

```
Bullet list:

- one
- two
- three
```

```
Enumerated list:

1. one
2. two
#. автоматическая нумерация

Definition list:

one
    one is a number.

two
    two is also a number.
```

Результат показан на рис. П.2.

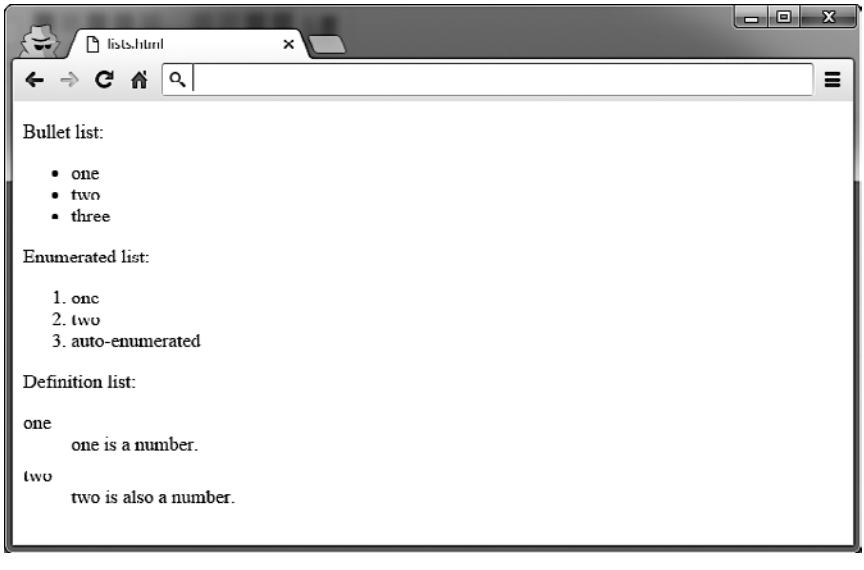


Рис. П.2. Различные списки в виде HTML-файла

В следующем подразделе поговорим о форматировании внутри строк.

Форматирование внутри строк

Текст можно стилизовать с помощью следующих конструкций:

- `*emphasis*` — курсив;
- `**strong emphasis**` — жирный;
- ``` ``inline preformatted`` ``` — переформатированный текст (обычно моноширинный, как в консоли);

- `` `a text with a link`_ `` — заменится гиперссылкой, если это предусмотрено в документе (см. подраздел «Ссылки» ниже).

В следующем подразделе поговорим о блоках литералов.

Блок литералов

Если вам нужно показать примеры кода, то можно использовать *блок литералов*. Два двоеточия служат для обозначения блока с отступом:

```
This is a code example

::

    >>> 1 + 1
    2

Let's continue our text
```

Не забудьте добавить пустую строку до и после ::, иначе отступа не будет.

Обратите внимание: символы двоеточия можно вставлять прямо в строку. В этом случае они будут заменены одним двоеточием в различных форматах рендеринга:

```
This is a code example:

    >>> 1 + 1
    2

Let's continue our text
```

Если вы не хотите использовать одно двоеточие, то можете вставить пробел между примером и `::`. В этом случае `::` будет интерпретировано и полностью удалено, как показано на рис. П.3.

В следующем подразделе поговорим о ссылках.

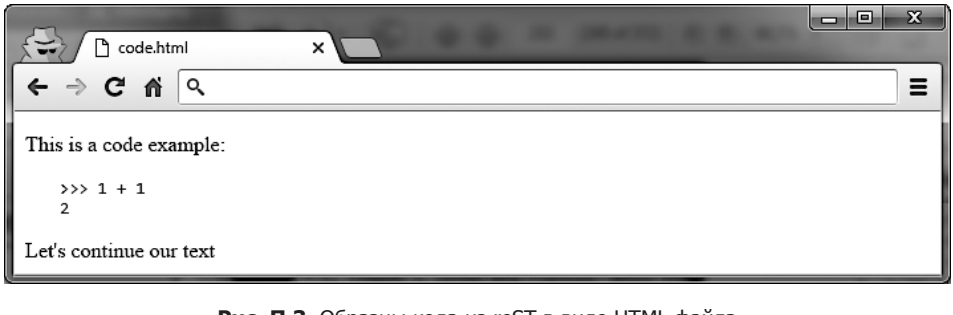


Рис. П.3. Образцы кода из reST в виде HTML-файла

Ссылки

Текст можно превратить во внешнюю *ссылку* с помощью специальной линии с двумя точками, если в документе есть такая ссылка:

```
Try `Plone CMS`_, it is great ! It is based on Zope_.

.. _`Plone CMS`: http://plone.org
.. _Zope: http://zope.org
```

Обычно все ссылки перечислены в конце документа. Если текст со ссылкой содержит пробелы, то должен быть окружен одинарными кавычками.

Внутренние ссылки также можно использовать при добавлении маркеров в текст:

```
This is a code example

.. _example:

::

    >>> 1 + 1
    2

Let's continue our text, or maybe go back to
the example_.
```

На разделы тоже можно ссылаться:

```
==============
Document title
==============

Introduction to the document content.

Section 1
=========

First document section.

Section 2
=========

-> go back to `Section 1`_
```

Михал Яворски, Тарек Зиаде

Python. Лучшие практики и инструменты

Перевел с английского А. Павлов

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*С. Бычковский*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

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: 04.2021. 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 25.03.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 45,150. Тираж 700. Заказ 0000.

Эрик Мэтиз

ИЗУЧАЕМ PYTHON: ПРОГРАММИРОВАНИЕ ИГР, ВИЗУАЛИЗАЦИЯ ДАННЫХ, ВЕБ-ПРИЛОЖЕНИЯ

3-е издание

«Изучаем Python» — это самое популярное в мире руководство по языку Python. Вы сможете не только максимально быстро его освоить, но и научитесь писать программы, устранять ошибки и создавать работающие приложения.

В первой части книги вы познакомитесь с основными концепциями программирования, такими как переменные, списки, классы и циклы, а простые упражнения приучат вас к шаблонам чистого кода. Вы узнаете, как делать программы интерактивными и как протестировать код, прежде чем добавлять в проект. Во второй части вы примените новые знания на практике и создадите три проекта: аркадную игру в стиле Space Invaders, визуализацию данных с удобными библиотеками Python и простое веб-приложение, которое можно быстро развернуть онлайн.

Работая с книгой, вы научитесь:

- Использовать мощные библиотеки и инструменты Python: Pygame, Matplotlib, Plotly и Django.
- Создавать 2D-игры разной сложности, которыми можно управлять с клавиатуры и мыши.
- Создавать интерактивную визуализацию данных.
- Разрабатывать, настраивать и развертывать веб-приложения.
- Разбираться с багами и ошибками.