Python
на практике

Марк Саммерфилд

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Марк Саммерфилд

Python
на практике

Создание качественных программ
с использованием параллелизма,
библиотек и паттернов

Python in Practice

Create Better Programs Using Concurrency, Libraries, and Patterns

Mark Summerfield

Addison-Wesley

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

Python на практике

Создание качественных программ
с использованием параллелизма,
библиотек и паттернов

Марк Саммерфилд

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2014

УДК 004.3'144:004.383.5Python
ББК 32.973.26-04
С17

С17 Марк Саммерфилд
Python на практике. / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2014. – 338 с.: ил.

ISBN 978-5-97060-095-5

Если вы – опытный программист на Python, то после прочтения данной книги ваши программы станут более качественными, надежными, быстрыми, удобными для сопровождения и использования.

В центре внимания Марка Саммерфилда находятся четыре основных темы: повышение элегантности кода с помощью паттернов проектирования, повышения быстродействия с помощью распараллеливания и компиляции Python-программ (Cython), высокоуровневое сетевое программирование и графика. Он описывает паттерны, доказавшие свою полезность в Python, иллюстрирует их на примерах высококачественного кода и объясняет, почему некоторые из них не слишком существенны.

Издание предназначено для программистов, уже работающих на Python, но также может быть полезно и начинающим пользователям языка.

ISBN 978-0-321-90563-5 (англ.)
ISBN 978-5-97060-095-5 (рус.)

Посвящается всем разработчикам бесплатного программного обеспечения с открытым исходным кодом.
Ваша щедрость – благо для всех нас.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Вот уже 15 лет как я пишу программы в разных областях на Python. Я видел, как сообщество росло и становилось более зрелым. Давно миновали те дни, когда нам приходилось «продавать» Python менеджерам, чтобы получить возможность использовать его в работе. Сегодня на программистов, пишущих на Python, большой спрос. На конференциях по Python всегда не протолкнуться, причем это относится не только к крупным национальным и международным мероприятиям, но и к местным собраниям. Благодаря проектам типа OpenStack язык захватывает новые территории, привлекая попутно новые таланты. Располагая здоровым и расширяющимся сообществом, мы теперь можем рассчитывать на более интересные и качественные книги о Python.

Марк Саммерфилд хорошо известен сообществу Python своими техническими текстами о Qt и Python. Книга Марка «Программирование на Python 3» занимает верхнее место в списке моих рекомендаций всем изучающим Python. Так я и отвечаю, когда мне как организатору группы пользователей в Атланте, штат Джорджия, задают этот вопрос. Эта книга тоже попадет в мой список, но для другой аудитории.

Большинство книг по программированию попадают в один из двух концов довольно широкого спектра, простирающегося от простого введения в язык (или программирование вообще) до более сложных книг, посвященных узкой теме, например, разработка веб-приложений, графические интерфейсы или биоинформатика. Работая над книгой «The Python Standard Library by Example», я рассчитывал на читателей, находящихся между этими крайностями, – сложившихся программистов-универсалов, которые знакомы с языком, но хотят отточить свои навыки выйти за пределы основ, но не ограничиваться какой-то узкой прикладной областью. Когда редактор попросил меня дать отзыв на предложение книги Марка, я с радостью увидел, что он ориентировал «Python на практике» на тот же круг читателей.

Давно уже не встречал я в книгах идей, которые можно было бы сразу же применить в каком-то из моих собственных проектов, не

привязываясь к конкретному каркасу или библиотеке. Последний год я работал над системой для измерения параметров облачных служб OpenStack. По ходу работы наша команда поняла, что данные, собираемые для выставления счетов, можно с пользой применить и для других целей, в том числе отчетности и мониторинга, поэтому мы спроектировали систему, которая рассылает их многим потребителям путем передачи выборок по конвейеру, составленному из повторно используемых трансформаций и издателей. Приблизительно одновременно с завершением кода конвейера я принялся писать техническую рецензию на эту книгу. Прочитав первые несколько разделов черновика главы 3, я понял, что наша реализация конвейера оказалась гораздо сложнее, чем нужно. Продемонстрированная Марком техника построения цепочки сопрограмм настолько элегантнее и проще для понимания, что я сразу же добавил в наш план задачу по перепроектированию в цикл подготовки следующей версии.

Книга «Python на практике» полна таких полезных советов и примеров так что вам будет чему поучиться. Универсалы вроде меня смогут познакомиться с некоторыми интересными инструментами, с которыми раньше не сталкивались. И будь вы опытным программистом или только-только вышедшим из начальной стадии карьеры, эта книга поможет взглянуть на проблему с разных точек зрения и подскажет, как создавать более эффективные решения.

Дуг Хэллман
старший разработчик, DreamHost
май, 2013

ВВЕДЕНИЕ

Эта книга ориентирована на программистов, пишущих на Python, которые хотели бы расширить и углубить знания языка, чтобы сделать свои программы более качественными, надежными, быстрыми, удобными для сопровождения и использования. В этой книге много практических примеров и идей. Рассматриваются четыре основных темы: применение паттернов проектирования для создания более элегантного кода, ускорение обработки за счет использования параллелизма и компиляции Python-кода (*Cython*), высокоуровневое сетевое программирование и графика.

Книга «Design Patterns: Elements of Reusable Object-Oriented Software»[1] (см. краткую библиографию) вышла еще в 1995 году, но по сей день оказывает огромное влияние на практическое объектно-ориентированное программирование. В книге «Python на практике» все паттерны проектирования рассмотрены в контексте языка Python – на примерах демонстрируется их полезность и объясняется, почему некоторые паттерны пишущим на Python неинтересны. Паттернам посвящены главы 1, 2 и 3.

Глобальная блокировка интерпретатора (GIL) в Python препятствует исполнению кода Python одновременно несколькими процессорными ядрами[2]. Отсюда пошел миф, будто программа на Python не может быть многопоточной и не способна воспользоваться преимуществами многоядерных процессоров. Но счетные задачи вполне можно распараллеливать с помощью модуля `multiprocessing`, который не связан ограничением GIL и может задействовать все имеющиеся ядра. При этом легко получить ожидаемое ускорение (примерно пропорциональное количеству ядер). Для программ, занятых преимущественно вводом-выводом, модуль `multiprocessing` тоже

---

[1] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования», ДМК Пресс, Питер, 2013. – *Прим. перев.*

[2] Это ограничение относится к CPython – эталонной реализации, которую использует большинство программистов. В некоторых реализациях Python такого ограничения нет, самой известной из них является Jython (Python, реализованный на Java).

можно использовать, а можно вместо этого обратиться к модулю `threading` или `concurrent.futures`. Если для распараллеливания таких программ используется модуль `threading`, то издержки GIL обычно маскируются сетевыми задержками, так что практического значения не имеют.

К сожалению, распараллеливание на низком и среднем уровне чревато ошибками (в любом языке). Этих проблем можно избежать, если воздержаться от явного использования блокировок, а работать с высокоуровневыми модулями `queue` и `multiprocessing` для реализации очередей или с модулем `concurrent.futures`. В главе 4 мы увидим, как с помощью высокоуровневого параллелизма достичь существенного повышения производительности.

Иногда программисты обращаются к C, C++ или другому компилируемому языку, поверив еще одному мифу – будто Python работает медленно. Да, вообще говоря, Python медленнее компилируемых языков, но при использовании современного оборудования его быстродействия более чем достаточно для большинства приложений. А в тех случаях, когда Python все-таки недостаточно шустр, мы все равно можем получать все преимущества от программирования на нем и при этом ускорить работу программы.

Для ускорения долго работающих программ можно использовать интерпретатор PyPy (`pypy.org`). Это JIT-компилятор, способный дать значительный выигрыш в скорости. Другой способ повысить производительность – пользоваться кодом, который работает со скоростью откомпилированного C; в счетных задачах так вполне можно добиться 100-кратного увеличения скорости. Получить такое быстродействие проще всего, воспользовавшись модулями Python, которые уже написаны на C, например, модулем `array` из стандартной библиотеки или сторонним модулем `numpy`, которые обеспечивают невероятно быструю и эффективную с точки зрения потребления памяти работу с массивами (в случае `numpy` – даже с многомерными). Другой вариант – выполнить профилирование программы с помощью модуля `cProfile` из стандартной библиотеки, найти узкие места и переписать критический в плане быстродействия код на Cython – это, по существу, вариант Python с расширенным синтаксисом, который компилируется в чистый C, что обеспечивает максимальную скорость во время выполнения.

Разумеется, иногда нужная функциональность уже имеется в какой-нибудь библиотеке, написанной на C, C++ или другом языке с таким же соглашением о вызовах, как в C. В большинстве случаев уже

имеется сторонний Python-модуль, реализующий интерфейс с этой библиотекой; его можно поискать в Указателе Python-пакетов (PyPI, `pypi.python.org`). Если же, что крайне маловероятно, такого модуля еще нет, то для доступа к функциям C-библиотеки можно воспользоваться модулем `ctypes` из стандартной библиотеки или сторонним пакетом Cython. Использование готовых C-библиотек заметно сокращает время разработки и обычно позволяет достичь очень высокой скорости работы. Cython и `ctypes` рассматриваются в главе 5.

В стандартной библиотеке Python есть много модулей для сетевого программирования, в том числе низкоуровневый модуль `socket`, модуль среднего уровня `socketserver` и высокоуровневый модуль `xmlrpclib`. Сетевое программирование на низком и среднем уровне оправдано при переносе кода с другого языка, но если программа с самого начала пишется на Python, то от низкоуровневых деталей можно уйти и сосредоточиться на функциональности приложения, воспользовавшись высокоуровневыми модулями. В главе 6 мы увидим, как это делается с помощью стандартного модуля `xmlrpclib` и мощного, но в то же время простого в использовании стороннего модуля `RPyC`.

Почти у всех программ есть какой-то пользовательский интерфейс, с помощью которого программе сообщают, что делать. На Python можно писать программы с интерфейсом командной строки, пользуясь модулем `argparse`, или с полноэкранным терминальным интерфейсом (например, в Unix для этого предназначен сторонний пакет `urwid`; `excess.org/urwid`). Есть также много веб-каркасов – от простого `bottle` (`bottlepy.org`) до таких тяжеловесных, как Django (`www.djangoproject.com`) и Pyramid (`www.pylonsproject.org`) – все они позволяют создавать приложения с веб-интерфейсом. И разумеется, на Python можно создавать приложения с графическим интерфейсом пользователя (ГИП).

Часто приходится слышать мнение, что ГИП-приложения скоро умрут, уступив место веб-приложениям. Но пока что этого не произошло. Более того, многие даже предпочитают приложения с графическим интерфейсом. Например, с тех пор как в начале 21 века обрели огромную популярность смартфоны, пользователи неизменно отдают предпочтение специально разработанным приложениям для повседневных задач, а не веб-страницам в браузере. На Python есть много сторонних пакетов для разработки графических приложений. Но в главе 7 мы рассмотрим, как создать приложение с современным графическим интерфейсом с помощью пакета Tkinter, входящего в стандартную библиотеку.

Большинство современных компьютеров – включая ноутбуки и даже смартфоны – оснащены мощными графическими средствами, часто в виде отдельного графического процессора (GPU), способного отрисовывать впечатляющую двухмерную и трехмерную графику. Почти все GPU поддерживают OpenGL API, а программист на Python может получить доступ к этому API с помощью сторонних пакетов. В главе 8 мы рассмотрим, как применить OpenGL для построения трехмерных изображений.

Цель этой книги – показать, как писать на Python более качественные приложения – высокопроизводительные, удобные для сопровождения и простые в использовании. Предполагается, что читатель уже умеет программировать на Python и изучил этот язык – по документации или по другим книгам, например, «Programming in Python 3, второе издание»[3] (см. краткую библиографию). В этой книге читатель найдет идеи, источник вдохновения и практические приемы, что позволит подняться на следующий уровень программирования.

Все примеры в книге протестированы в версии Python 3.3 (а при возможности также в Python 3.2 and Python 3.1) в Linux, OS X (в большинстве случаев) и Windows (в большинстве случаев). Код примеров можно скачать со страницы `www.qtrac.eu/pipbook.html`, он должен работать во всех будущих версиях Python 3.*x*.

Благодарности

Как и все написанные мной технические книги, эта не могла бы состояться без советов, помощи и поддержки со стороны многих людей. Всем им я благодарен.

Ник Кофлэн (Nick Coghlan), входящий в группу разработчиков ядра Python с 2005 года, высказал немало конструктивных критических замечаний, подкрепив их уймой идей и фрагментов кода, чтобы показать, как можно решить задачу по-другому и лучше. Помощь Ника была бесценной на протяжении работы над всей книгой, но особенно от нее выиграли первые главы.

Дуг Хеллман (Doug Hellmann), опытный программист на Python и автор книги, прислал мне массу полезных замечаний – как по исходному предложению, так и по каждой главе самой книги. Дуг подарил мне много идей и согласился написать предисловие.

3 Марк Саммерфилд «Программирование на Python 3. Подробное руководство», Символ-Плюс, 2009. – *Прим. перев.*

Два друга – Джасмин Бланшетт (Jasmin Blanchette) и Трентон Шульц (Trenton Schulz) – умудренные опытом программисты, а поскольку Python они знают совершенно с разных сторон, то оказались идеальными представителями предполагаемой читательской аудитории. Отзывы Джасмина и Трентона позволили улучшить и сделать понятнее много мест в тексте и примерах кода.

С удовольствием говорю спасибо заказавшему книгу редактору, Дэбре Уильямс Коули (Debra Williams Cauley), она уже в который раз оказывает мне практическую помощь и поддержку по ходу работы.

Спасибо Элизабет Райан (Elizabeth Ryan), которая прекрасно организовала процесс производства, и корректору Анне В. Попик (Anna V. Popick) за отлично проделанную работу.

И, как всегда, благодарю свою жену Андреа за любовь и поддержку.

ГЛАВА 1. Порождающие паттерны проектирования в Python

Порождающие паттерны проектирования описывают, как создавать объекты. Обычно объект создается путем вызова конструктора (то есть объекта его класса с аргументами), но иногда желательна большая гибкость – именно поэтому порождающие паттерны и полезны.

Для пишущих на Python некоторые из этих паттернов покажутся очень похожими друг на друга, а кое-какие, как мы скоро увидим, вообще не нужны. Объясняется это тем, что изначально паттерны проектирования предназначались, прежде всего, для C++, и приходилось преодолевать некоторые ограничения этого языка. В Python таких ограничений нет.

1.1. Паттерн Абстрактная фабрика

Паттерн Абстрактная фабрика предназначен для случаев, когда требуется создать сложный объект, состоящий из других объектов, причем все составляющие объекты принадлежат одному «семейству».

Например, в системе с графическим пользовательским интерфейсом может быть абстрактная фабрика виджетов, которой наследуют три конкретные фабрики: `MacWidgetFactory`, `XfceWidgetFactory` и `WindowsWidgetFactory`, каждая из которых предоставляет методы для создания одних и тех же объектов (`make_button()`, `make_spinbox()` и т. д.), стилизованных, однако, как принято на конкретной платформе. Это дает возможность создать обобщенную функцию `create_dialog()`, которая принимает экземпляр фабрики в качестве аргумента и создает диалоговое окно, выглядящее, как в OS X, Xfce или Windows, – в зависимости от того, какую фабрику мы передали.

1.1.1. Классическая Абстрактная фабрика

Для иллюстрации паттерна Абстрактная фабрика рассмотрим программу, которая создает нехитрый рисунок. Нам понадобятся две фабрики: одна будет выводить простой текст, другая – файл в формате SVG (Scalable Vector Graphics). Оба результата показаны на рис. 1.1. В первой версии программы, `diagram1.py`, демонстрируется паттерн в чистом виде. А во второй версии, `diagram2.py`, мы воспользуемся некоторыми особенностями Python, чтобы немного сократить код и сделать его элегантнее. Результат в обоих случаях один и тот же[1].

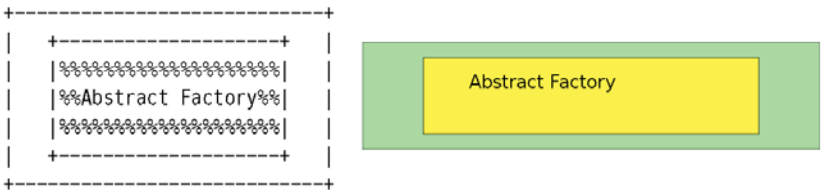

Рис. 1.1. Рисунки в виде простого текста и SVG

Сначала рассмотрим код, общий для обеих версий и начнем с функции `main()`.

```
def main():
    ...
    txtDiagram = create_diagram(DiagramFactory()) ❶
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory()) ❷
    svgDiagram.save(svgFilename)
```

Первым делом мы определяем два имени файла (не показано). Затем мы порождаем рисунок с помощью фабрики простого текста, подразумеваемой по умолчанию (❶), и сохраняем его. После этого мы точно так же порождаем и сохраняем рисунок, но на этот раз с помощью фабрики SVG (❷).

```
def create_diagram(factory):
    diagram = factory.make_diagram(30, 7)
    rectangle = factory.make_rectangle(4, 1, 22, 5, "yellow")
    text = factory.make_text(7, 3, "Abstract Factory")
    diagram.add(rectangle)
    diagram.add(text)
    return diagram
```

[1] Все приведенные в этой книге примеры можно скачать со страницы www.qtrac.eu/pipbook.html.

Эта функция принимает фабрику рисунков и с ее помощью создает требуемый рисунок. Ей совершенно безразлично, какую конкретно фабрику она получила, лишь бы та поддерживала определенный нами интерфейс. О методах `make_...()` мы поговорим чуть ниже.

Разобравшись с тем, как фабрики используются, мы можем обратиться к самим фабрикам. Вот код простой фабрики текстовых рисунков (которая заодно является базовым классом для фабрик):

```
class DiagramFactory:

   def make_diagram(self, width, height):
      return Diagram(width, height)

   def make_rectangle(self, x, y, width, height, fill="white",
         stroke="black"):
      return Rectangle(x, y, width, height, fill, stroke)

   def make_text(self, x, y, text, fontsize=12):
      return Text(x, y, text, fontsize)
```

Несмотря на слово «абстрактная» в названии паттерна, часто бывает, что один и тот же класс служит и базовым классом, определяющим интерфейс (то есть абстракцию), и самостоятельным конкретным классом. Именно так мы и поступили в классе `DiagramFactory`.

Вот первые несколько строчек фабрики SVG:

```
class SvgDiagramFactory(DiagramFactory):

   def make_diagram(self, width, height):
      return SvgDiagram(width, height)
   ...
```

Единственное различие между двумя методами `make_diagram()` состоит в том, что `DiagramFactory.make_diagram()` возвращает объект `Diagram`, а `SvgDiagramFactory.make_diagram()` – объект `SvgDiagram`. То же самое относится и к двум другим методам класса `SvgDiagramFactory` (не показаны).

Скоро мы увидим, что реализации классов `Diagram`, `Rectangle` и `Text` кардинально отличаются от реализаций классов `SvgDiagram`, `SvgRectangle` и `SvgText` – хотя все классы реализуют один и тот же интерфейс (то есть в классах `Diagram` и `SvgDiagram` одинаковый набор методов). Это означает, что нельзя смешивать классы из разных семейств (например, `Rectangle` и `SvgText`) – и это ограничение автоматически поддерживается фабричными классами.

В объектах `Diagram`, представляющих рисунки в виде простого текста, данные хранятся в виде списка списков односимвольных строк, в которых символ может быть пробелом, знаком `+`, `|`, `-` и т. д. Классы `Rectangle` и `Text` содержат список списков односимвольных строк, которые должны быть подставлены в общий рисунок в соответствующих позициях.

```
class Text:

   def __init__(self, x, y, text, fontsize):
      self.x = x
      self.y = y
      self.rows = [list(text)]
```

Это полный код класса `Text`. Параметры `fontsize` для простого текста мы просто игнорируем.

```
class Diagram:
   ...

   def add(self, component):
      for y, row in enumerate(component.rows):
         for x, char in enumerate(row):
            self.diagram[y + component.y][x + component.x] = char
```

А это метод `Diagram.add()`. Если вызвать его, передав объект класса `Rectangle` или `Text` (в параметре `component`), то он переберет все символы в списке списков односимвольных строк компонента (`component.rows`) и заменит соответствующие символы в рисунке. Метод `Diagram.__init__()` (не показан) уже инициализировал `self.diagram` списком списков пробелов (заданной ширины и высоты) при вызове `Diagram(width, height)`.

```
SVG_TEXT = """<text x="{x}" y="{y}" text-anchor="left" \
font-family="sans-serif" font-size="{fontsize}">{text}</text>"""

SVG_SCALE = 20

class SvgText:

   def __init__(self, x, y, text, fontsize):
      x *= SVG_SCALE
      y *= SVG_SCALE
      fontsize *= SVG_SCALE // 10
      self.svg = SVG_TEXT.format(**locals())
```

Это полный код класса `SvgText` и двух констант, которые в нем используются[2]. Кстати, использование `**locals()` позволяет обойтись без записи `SVG_TEXT.format(x=x, y=y, text=text, fontsize=fontsize)`. Начиная с версии Python 3.2, можно было бы писать вместо этого `SVG_TEXT.format_map(locals())`, потому что метод `str.format_map()` автоматически распаковывает отображение (см. врезку «Распаковка последовательностей и отображение» на стр. 24).

```
class SvgDiagram:
   ...

   def add(self, component):
      self.diagram.append(component.svg)
```

В объекте класса `SvgDiagram` хранится список строк в `self.diagram`, и каждая строка – это кусок кода на SVG. Поэтому добавление новых компонентов (например, типа `SvgRectangle` или `SvgText`) не вызывает никаких сложностей.

1.1.2. Абстрактная фабрика в духе Python

Класс `DiagramFactory`, его подкласс `SvgDiagramFactory`, а также классы, которые в них используются (`Diagram`, `SvgDiagram` и т. д.), прекрасно работают в полном соответствии с паттерном проектирования.

Тем не менее, у нашей реализации есть несколько недостатков. Во-первых, ни у одной фабрики нет собственного состояния, поэтому и создавать экземпляры фабрики по сути дела ни к чему. Во-вторых, код `SvgDiagramFactory` отличается от кода `DiagramFactory` только тем, что возвращает объекты типа `SvgText`, а не `Text` и т. д. – таким образом, налицо ненужное дублирование. В-третьих, в пространстве имен верхнего уровня находятся все классы: `DiagramFactory`, `Diagram`, `Rectangle`, `Text` и их SVG-эквиваленты. Но на самом деле нам нужен доступ только к двум фабрикам. Далее, мы вынуждены добавлять к именам классов префикс SVG (например, `SvgRectangle` вместо `Rectangle`), чтобы избежать конфликта имен, а это некрасиво. (Для устранения конфликта имен можно было бы поместить каждый класс в свой модуль. Но это не решает проблему дублирования кода.)

[2] Мы порождаем SVG-код без претензий на изящество, но для демонстрации паттерна проектирования этого достаточно. Сторонние модули, поддерживающие SVG, можно найти в указателе пакетов Python (PyPI) на сайте pypi.python.org.

В этом подразделе мы устраним все указанные недостатки (код находится в файле `diagram2.py`).

Первым делом мы вложим классы `Diagram`, `Rectangle` и `Text` в класс `DiagramFactory`. Это означает, что отныне к ним нужно обращаться так: `DiagramFactory.Diagram` и т. д. Соответственные классы для SVG можно вложить в классы `SvgDiagramFactory`, назвав их так же, как классы для простого текста, – ведь конфликта имен уже не будет – например, `SvgDiagramFactory.Diagram`. Мы сделаем вложенными также константы, которые используются в классах, и таким образом на верхнем уровне останутся только имена `main()`, `create_diagram()`, `DiagramFactory` и `SvgDiagramFactory`.

```
class DiagramFactory:

   @classmethod
   def make_diagram(Class, width, height):
      return Class.Diagram(width, height)

   @classmethod
   def make_rectangle(Class, x, y, width, height, fill="white",
         stroke="black"):
      return Class.Rectangle(x, y, width, height, fill, stroke)

   @classmethod
   def make_text(Class, x, y, text, fontsize=12):
      return Class.Text(x, y, text, fontsize)
   ...
```

Это начало нового класса `DiagramFactory`. Методы `make_…()` теперь стали методами класса, а не экземпляра. Это означает, что при обращении к ним первым аргументом передается класс (а не `self`, как для обычных методов). В данном случае при вызове `DiagramFactory.make_text()` в аргументе `Class` будет передан класс `DiagramFactory`, а в результате создается и возвращается объект типа `DiagramFactory.Text`.

Это изменение означает также, что подкласс `SvgDiagramFactory`, наследующий `DiagramFactory`, теперь вовсе не нуждается в методах `make_…()`. Если вызвать, к примеру, метод `SvgDiagramFactory.make_rectangle()`, то, поскольку в классе `SvgDiagramFactory` нет такого метода, будет вызван метод базового класса `DiagramFactory.make_rectangle()` – но в качестве аргумента `Class` будет передан `SvgDiagramFactory`. Поэтому будет создан и возвращен объект `SvgDiagramFactory.Rectangle`.

```
def main():
    ...
    txtDiagram = create_diagram(DiagramFactory)
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory)
    svgDiagram.save(svgFilename)
```

Благодаря этим изменениям мы можем также упростить функцию `main()`, потому что создавать экземпляры фабрик больше не требуется.

Остальной код почти не отличается от предыдущего, разница лишь в том, что, поскольку константы и нефабричные классы теперь вложены в фабрики, при обращении к ним нужно указывать имя фабрики.

```
class SvgDiagramFactory(DiagramFactory):
    ...
    class Text:

        def __init__(self, x, y, text, fontsize):
            x *= SvgDiagramFactory.SVG_SCALE
            y *= SvgDiagramFactory.SVG_SCALE
            fontsize *= SvgDiagramFactory.SVG_SCALE // 10
            self.svg = SvgDiagramFactory.SVG_TEXT.format(**locals())
```

Выше показан вложенный в `SvgDiagramFactory` класс `Text` (эквивалентный классу `SvgText` из файла `diagram1.py`), который знает, как обращаться к вложенным константам.

1.2. Паттерн Построитель

Паттерн Построитель аналогичен паттерну Абстрактная фабрика в том смысле, что оба предназначены для создания сложных объектов, составленных из других объектов. Но отличается он тем, что не только предоставляет методы для построения сложного объекта, но и хранит внутри себя его полное представление.

Этот паттерн допускает такую же композиционную структуру, как Абстрактная фабрика (то есть сложные объекты, составленные из нескольких более простых), но особенно удобен в ситуациях, когда представление составного объекта должно быть отделено от алгоритмов композиции.

Мы продемонстрируем использование Построителя на примере программы, которая умеет порождать формы – либо веб-формы с помощью HTML, либо ГИП-формы с помощью Python и Tkinter. Те и

другие имеют графический интерфейс и поддерживают ввод текста, однако кнопки в них не работают[3]. Внешний вид форм показан на рис. 1.2, а исходный код находится в файле `formbuilder.py`.

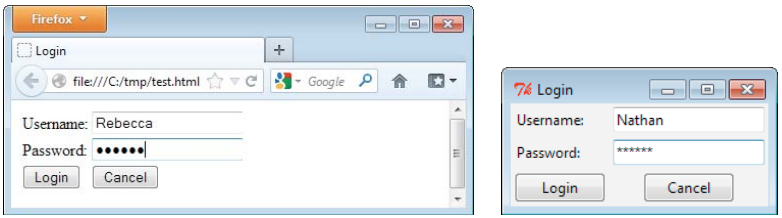

Рис. 1.2. Формы на HTML и Tkinter в Windows

Сначала рассмотрим код построения форм и начнем с вызовов верхнего уровня.

```
htmlForm = create_login_form(HtmlFormBuilder())
with open(htmlFilename, "w", encoding="utf-8") as file:
   file.write(htmlForm)

tkForm = create_login_form(TkFormBuilder())
with open(tkFilename, "w", encoding="utf-8") as file:
   file.write(tkForm)
```

Здесь мы создали обе формы и записали их в файл. Функция создания формы в обоих случаях одна и та же (`create_login_form()`), но в качестве параметра ей передается объект-построитель.

```
def create_login_form(builder):
   builder.add_title("Login")
   builder.add_label("Username", 0, 0, target="username")
   builder.add_entry("username", 0, 1)
   builder.add_label("Password", 1, 0, target="password")
   builder.add_entry("password", 1, 1, kind="password")
   builder.add_button("Login", 2, 0)
   builder.add_button("Cancel", 2, 1)
   return builder.form()
```

Эта функция может создать произвольную форму – HTML, Tkinter или любую другую – при условии, что имеется подходящий построитель. Метод `builder.add_title()` служит для задания заголовка формы. Остальные методы добавляют в форму тот или иной виджет в позиции с указанными координатами (строка и столбец).

3 Во всех примерах приходится выдерживать баланс между реалистичностью и пригодностью для обучения, поэтому некоторые – этот в том числе – обладают урезанной функциональностью.

Классы `HtmlFormBuilder` и `TkFormBuilder` наследуют абстрактному базовому классу `AbstractFormBuilder`.

```
class AbstractFormBuilder(metaclass=abc.ABCMeta):

   @abc.abstractmethod
   def add_title(self, title):
      self.title = title
      @abc.abstractmethod

   def form(self):
      pass

   @abc.abstractmethod
   def add_label(self, text, row, column, **kwargs):
      pass
   ...
```

Любой класс, производный от этого, должен реализовать все абстрактные методы. Мы опустили абстрактные методы `add_entry()` и `add_button()`, потому что они не отличаются от метода `add_label()` ничем, кроме названия. Кстати, мы должны связать с `AbstractFormBuilder` метакласс `abc.ABCMeta`, чтобы можно было воспользоваться декоратором `@abstractmethod` из модуля `abc` (о декораторах см. раздел 2.4).

Распаковка последовательностей и отображений

Под распаковкой понимается извлечение всех элементов последовательности или отображения по отдельности. Простой случай распаковки последовательности – извлечение первого или нескольких начальных элементов, а затем всех остальных. Например:

```
first, second, *rest = sequence
```

Здесь предполагается, что в последовательности `sequence` есть хотя бы три элемента: `first == sequence[0]`, `second == sequence[1]`, `rest == sequence[2:]`.

Пожалуй, чаще всего распаковка применяется при вызове функций. Если имеется функция, которая ожидает сколько-то позиционных параметров или определенные ключевые параметры, то для их предоставления можно воспользоваться распаковкой, например:

```
args = (600, 900)
kwargs = dict(copies=2, collate=False)
print_setup(*args, **kwargs)
```

Функция `print_setup()` ожидает получить два обязательных позиционных аргумента (`width` и `height`) и готова принять необязательные ключевые

аргументы (`copies` и `collate`). Вместо того чтобы передавать их значения напрямую, мы создали кортеж `args` и словарь `kwargs`, а затем применили распаковку последовательности (`*args`) и отображения (`**kwargs`). Результат точно такой же, как если бы мы написали `print_setup(600, 900, copies=2, collate=False)`.

Другое применение – создание функции, которая принимает произвольное количество позиционных аргументов, ключевых аргументов или тех и других сразу. Например:

```
def print_args(*args, **kwargs):
   print(args.__class__.__name__, args,
         kwargs.__class__.__name__, kwargs)

print_args() # печатается: tuple () dict {}
print_args(1, 2, 3, a="A") # печатается: tuple (1, 2, 3) dict {'a': 'A'}
```

Функция `print_args()` принимает произвольное количество позиционных и ключевых аргументов. Внутри нее `args` имеет тип `tuple`, а `kwargs` – тип `dict`. Захоти мы передать эти аргументы функции, вызываемой из `print_args()`, можно было бы воспользоваться распаковкой (`function(*args, **kwargs)`). Другой частый случай употребления распаковки отображений встречается при вызове метода `str.format()` –например, `s.format(**locals())` вместо набора всех пар `key=value` вручную (см., например, `SvgText.__init__()`; на стр. 19).

Связывая с классом метакласс `abc.ABCMeta`, мы говорим, что этот класс нельзя инстанцировать, то есть можно использовать только в качестве абстрактного базового класса. Это имеет смысл при переносе кода, написанного, например, на C++ или Java, но влечет незначительные накладные расходы во время выполнения. Впрочем, многие программисты на Python предпочитают более либеральный подход: они вообще не используют метакласс, а просто пишут в документации, что данный класс следует рассматривать как абстрактный базовый.

```
class HtmlFormBuilder(AbstractFormBuilder):

   def __init__(self):
      self.title = "HtmlFormBuilder"
      self.items = {}

   def add_title(self, title):
      super().add_title(escape(title))

   def add_label(self, text, row, column, **kwargs):
      self.items[(row, column)] = ('<td><label for="{}">{}:</label></td>'
            .format(kwargs["target"], escape(text)))

   def add_entry(self, variable, row, column, **kwargs):
```

```
        html = """<td><input name="{}" type="{}" /></td>""".format(
                variable, kwargs.get("kind", "text"))
        self.items[(row, column)] = html
    ...
```

Это начало класса `HtmlFormBuilder`. На случай, если строится форма без заголовка, мы предусмотрели заголовок по умолчанию. Все составляющие форму виджеты хранятся в словаре `items`, ключами которого служат кортежи из 2 элементов *`строка, столбец`*, а значениями – HTML-код виджета.

Мы должны переопределить абстрактный метод `add_title()`, но поскольку в базовом классе уже имеется реализация, мы можем ее просто вызвать. В данном случае необходимо пропустить заголовок через функцию `html.escape()` (или функцию `xml.sax.saxutil.escape()` в версии Python 3.2 и более ранних).

Метод `add_button()` (не показан) структурно похож на остальные методы `add_...()`.

```
def form(self):
    html = ["<!doctype html>\n<html><head><title>{}</title></head>"
            "<body>".format(self.title), '<form><table border="0">']
    thisRow = None
    for key, value in sorted(self.items.items()):
        row, column = key
        if thisRow is None:
            html.append(" <tr>")
        elif thisRow != row:
            html.append(" </tr>\n <tr>")
        thisRow = row
        html.append(" " + value)
    html.append(" </tr>\n</table></form></body></html>")
    return "\n".join(html)
```

Метод `HtmlFormBuilder.form()` создает HTML-страницу, содержащую тег `<form>`, внутри которого находится `<table>`, а внутри последнего – строки и столбцы, состоящие из виджетов. После того как все кусочки добавлены в список `html`, мы возвращаем этот список в виде одной строки (расставив в ней знаки новой строки, чтобы было понятнее человеку).

```
class TkFormBuilder(AbstractFormBuilder):

    def __init__(self):
        self.title = "TkFormBuilder"
        self.statements = []

    def add_title(self, title):
```

```
        super().add_title(title)

    def add_label(self, text, row, column, **kwargs):
        name = self._canonicalize(text)
        create = """self.{}Label = ttk.Label(self, text="{}:")""".format(
                name, text)
        layout = """self.{}Label.grid(row={}, column={}, sticky=tk.W, \
padx="0.75m", pady="0.75m")""".format(name, row, column)
        self.statements.extend((create, layout))
    ...

    def form(self):
        return TkFormBuilder.TEMPLATE.format(title=self.title,
                name=self._canonicalize(self.title, False),
                statements="\n       ".join(self.statements))
```

Это фрагмент класса `TkFormBuilder`. Мы сохраняем составляющие форму виджеты в виде списка предложений (строк Python-кода), по два предложения на виджет.

У методов `add_entry()` и `add_button()` (не показаны) такая же структура, как у `add_label()`. Все они начинают с построения канонического имени виджета, а затем конструируют две строки: `create`, которая содержит код создания виджета, и `layout`, которая содержит код размещения виджета в форме. В конце каждый метод добавляет обе строки в список предложений.

Метод `form()` совсем простой: он всего лишь возвращает строку `TEMPLATE` с подставленными заголовком и предложениями.

```
    TEMPLATE = """#!/usr/bin/env python3
import tkinter as tk
import tkinter.ttk as ttk

class {name}Form(tk.Toplevel): ❶

    def __init__(self, master):
        super().__init__(master)
        self.withdraw() # скрыть, пока не будем готовы показать
        self.title("{title}") ❷
        {statements} ❸
        self.bind("<Escape>", lambda *args: self.destroy())
        self.deiconify() # показать, когда виджеты созданы и размещены
        if self.winfo_viewable():
            self.transient(master)
        self.wait_visibility()
        self.grab_set()
        self.wait_window(self)

if __name__ == "__main__":
```

```
    application = tk.Tk()
    window = {name}Form(application) ❹
    application.protocol("WM_DELETE_WINDOW", application.quit)
    application.mainloop()
"""
```

Форме назначается уникальное имя класса, основанное на заголовке (например, `LoginForm`, ❶; ❹). Сначала устанавливается заголовок окна (например, «Login», ❷), затем следуют все предложения, в которых создаются и размещаются виджеты (❸).

Порожденный шаблоном Python-код можно запускать автономно благодаря блоку `if __name__` ... в конце.

```
def _canonicalize(self, text, startLower=True):
    text = re.sub(r"\W+", "", text)
    if text[0].isdigit():
        return "_" + text
    return text if not startLower else text[0].lower() + text[1:]
```

Метод `_canonicalize()` показан для полноты картины. Кстати, хотя на первый взгляд кажется, что мы заново создаем регулярное выражение при каждом вызове функции, на самом деле Python поддерживает весьма объемный внутренний кэш откомпилированных регулярных выражений, поэтому при втором и последующих вызовах интерпретатор просто находит регулярное выражение в кэше, а не компилирует его повторно[4].

1.3. Паттерн Фабричный метод

Паттерн Фабричный метод применяется, когда мы хотим, чтобы подклассы выбирали, какой класс инстанцировать, когда запрашивается объект. Это полезно само по себе, но можно пойти дальше и использовать в случае, когда класс заранее неизвестен (например, зависит от информации, прочитанной из файла или введенных пользователем данных).

В этом разделе мы рассмотрим программу, которую можно использовать для создания игровой доски (например, шашечной или шахматной). На рис. 1.3 показано, что программа выводит, а в фай-

[4] Предполагается, что читатель знаком с регулярными выражениями и модулем `re`. Если вам необходимо подучить этот вопрос, скачайте бесплатный PDF-файл «Chapter 13. Regular Expressions» с сайта книги «Programming in Python 3, Second Edition» того же автора по адресу www.qtrac.eu/py3book.html.

лах `gameboard1.py`, ... `gameboard4.py` представлены четыре варианта кода[5].

Нам хотелось бы иметь абстрактный класс доски, подклассы которого создают доски для разных игр. Каждый класс на этапе инициализации создает начальное расположение фигур. И еще мы хотим, чтобы для каждой фигуры был свой класс (например, `BlackDraught`, `WhiteDraught`, `BlackChessBishop`, `WhiteChessKnight` и т. д.). Кстати, в качестве имени класса мы выбрали `WhiteDraught`, а не `WhiteChecker`, поскольку именно так соответствующая литера (шашка) называется в Unicode.

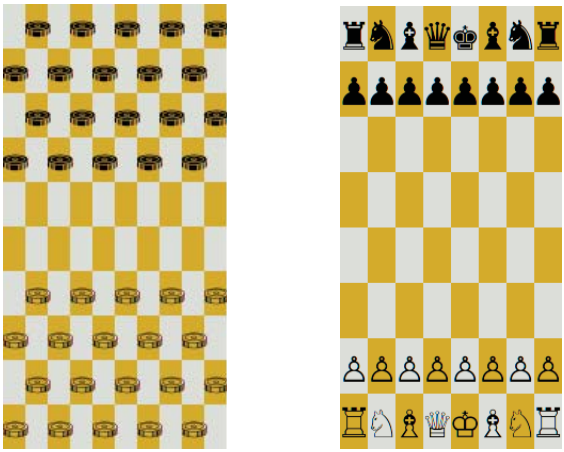

Рис. 1.3. Шашечная и шахматная доска на консоли Linux

Мы начнем с кода верхнего уровня, который создает объекты досок и распечатывает их. Затем познакомимся с классами досок и некоторыми из классов фигур – в первом варианте фигуры будут зашиты в код. Далее мы рассмотрим вариации, которые позволят избежать зашивания данных в классы и заодно уменьшить количество строк кода.

```
def main():
    checkers = CheckersBoard()
```

[5] К сожалению, поддержка UTF-8 на консоли Windows оставляет желать лучшего, очень много символов отсутствует, даже если выбрать кодовую страницу 65001. Поэтому в Windows эти программы выводят результаты во временный файл, а на консоли печатают его имя. Ни в одном из стандартных моноширинных шрифтов Windows, похоже, нет символов шашек и шахматных фигур, хотя в пропорциональных шрифтах шахматные фигуры есть. Все эти символы есть в бесплатном шрифте DejaVu Sans, который распространяется в исходном виде (dejavu-fonts.org).

```
print(checkers)

chess = ChessBoard()
print(chess)
```

Это общая для всех вариантов программы функция. Она просто создает доски разных типов и распечатывает их на консоли, полагаясь на то, что метод `__str__()` класса `AbstractBoard` преобразует внутреннее представление доски в строку.

```
BLACK, WHITE = ("BLACK", "WHITE")

class AbstractBoard:

    def __init__(self, rows, columns):
        self.board = [[None for _ in range(columns)] for _ in range(rows)]
        self.populate_board()

    def populate_board(self):
        raise NotImplementedError()

    def __str__(self):
        squares = []
        for y, row in enumerate(self.board):
            for x, piece in enumerate(row):
                square = console(piece, BLACK if (y + x) % 2 else WHITE)
                squares.append(square)
            squares.append("\n")
        return "".join(squares)
```

Константы `BLACK` и `WHITE` служат для обозначения цвета клеток доски. Позже они будут использоваться также для обозначения цвета фигур. Код этого класса взят из файла `gameboard1.py`, но на самом деле он одинаков во всех версиях.

Было бы привычнее задать константы в виде `BLACK, WHITE = range(2)`. Но строки оказываются гораздо полезнее, когда дело доходит до отладки, а с точки зрения быстродействия они мало чем уступают целым числам, поскольку в Python реализован механизм внутреннего представления с проверкой идентичности.

Доска представлена списком горизонталей, каждая из которых есть список строк из одного символа, причем незанятым клеткам соответствует `None`. Функция `console()` (не показана, но есть в исходном коде) возвращает строку, представляющую заданную фигуру в клетке заданного цвета (в Unix-подобных системах строка содержит также управляющие последовательности для задания цвета фона).

Можно было бы сделать `AbstractBoard` абстрактным классом формально, ассоциировав с ним метакласс `abc.ABCMeta` (как мы поступили с классом `AbstractFormBuilder` на стр. 24). Однако на этот раз мы избрали другой подход и просто возбуждаем исключение `NotImplementedError` во всех методах, которые должны быть реализованы в подклассах.

```
class CheckersBoard(AbstractBoard):

   def __init__(self):
      super().__init__(10, 10)

   def populate_board(self):
      for x in range(0, 9, 2):
         for row in range(4):
            column = x + ((row + 1) % 2)
            self.board[row][column] = BlackDraught()
            self.board[row + 6][column] = WhiteDraught()
```

Этот подкласс создает доску для игры в международные шашки размером 10×10. Метод `populate_board()` этого класса *не* является фабричным методом, потому что пользуется зашитыми в код конкретными классами; он приведен лишь в качестве шага на пути к фабричному методу.

```
class ChessBoard(AbstractBoard):

   def __init__(self):
      super().__init__(8, 8)

   def populate_board(self):
      self.board[0][0] = BlackChessRook()
      self.board[0][1] = BlackChessKnight()
      ...
      self.board[7][7] = WhiteChessRook()
      for column in range(8):
         self.board[1][column] = BlackChessPawn()
         self.board[6][column] = WhiteChessPawn()
```

Эта версия метода `populate_board()` класса `ChessBoard` также *не* является фабричным методом, но показывает, как заполняется шахматная доска.

```
class Piece(str):

   __slots__ = ()
```

Этот класс является базовым для фигур. Можно было бы просто использовать класс `str`, но это не позволило бы узнать, является ли объект фигурой (например, с помощью `isinstance(x, Piece)`). Предложение `__slots__ = ()` гарантирует, что в экземплярах нет данных; эту тему мы обсудим ниже в разделе 2.6.

```
class BlackDraught(Piece):

   __slots__ = ()

   def __new__(Class):
      return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):

   __slots__ = ()

   def __new__(Class):
      return super().__new__(Class, "\N{white chess king}")
```

Эти два класса являются типичными представителями всех классов фигур. Каждый из них – неизменяемый подкласс класса `Piece` (который сам является подклассом `str`), инициализируемый строкой, содержащей единственный символ Unicode, который представляет соответствующую фигуру. Всего будет четырнадцать таких крохотных подклассов, отличающихся только именем и хранимой строкой. Конечно, хотелось бы избавиться от такого «почти дублирования».

```
def populate_board(self):
   for x in range(0, 9, 2):
      for y in range(4):
         column = x + ((y + 1) % 2)
         for row, color in ((y, "black"), (y + 6, "white")):
            self.board[row][column] = create_piece("draught", color)
```

Эта версия метода `CheckersBoard.populate_board()` (взятая из файла `gameboard2.py`) уже является фабричным методом, потому что в ней используется новая фабричная функция `create_piece()`, а не зашитые в код классы. Функция `create_piece()` возвращает объект того или иного типа (например, `BlackDraught` или `WhiteDraught`) в зависимости от переданных аргументов. Метод `ChessBoard.populate_board()` (не показан) в этой версии программы аналогичен, в нем тоже используются строки, задающие цвет и названия фигур, и та же самая функция `create_piece()`.

```
def create_piece(kind, color):
   if kind == "draught":
      return eval("{}{}()".format(color.title(), kind.title()))
   return eval("{}Chess{}()".format(color.title(), kind.title()))
```

В этой фабричной функции используется встроенная функция `eval()` для создания экземпляров классов. Например, если переданы аргументы `"knight"` и `"black"`, то `eval()` будет вызвана с аргументом `"BlackChessKnight()"`. Хотя этот код работает, потенциально он представляет опасность, так как с помощью `eval()` на свет можно произвести что угодно. Ниже мы увидим решение, основанное на использовании встроенной функции `type()`.

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
   char = chr(code)
   name = unicodedata.name(char).title().replace(" ", "")
   if name.endswith("sMan"):
      name = name[:-4]
   exec("""\
class {}(Piece):

   __slots__ = ()

   def __new__(Class):
      return super().__new__(Class, "{}")""".format(name, char))
```

Вместо того чтобы выписывать код четырнадцати очень похожих классов, мы создаем все нужные нам классы внутри одного блока.

Функция `itertools.chain()` принимает один или несколько итерируемых объектов и возвращает единственный итерируемый объект, который обходит первый переданный итерируемый объект, затем второй и т. д. В данном случае мы передали два итерируемых объекта: первый – кортеж из двух кодовых точек Unicode, представляющих черную и белую шашку, второй – объект `range` (по существу, генератор), представляющий черные и белые шахматные фигуры.

Для каждой кодовой точки создается строка из одного символа (например, "♞"), а затем имя класса, определяемое названием символа в Unicode (например, «black chess knight» преобразуется в `BlackChessKnight`). Зная символ и имя, мы с помощью `exec()` создаем нужный класс. В этом коде всего дюжина строк – сравните с сотней строк, которые пришлось бы написать, если бы мы создавали каждый класс по отдельности.

К сожалению, `exec()` потенциально еще опаснее, чем `eval()`, поэтому поищем решение получше.

```
DRAUGHT, PAWN, ROOK, KNIGHT, BISHOP, KING, QUEEN = ("DRAUGHT", "PAWN",
         "ROOK", "KNIGHT", "BISHOP", "KING", "QUEEN")

class CheckersBoard(AbstractBoard):
   ...
   def populate_board(self):
      for x in range(0, 9, 2):
         for y in range(4):
            column = x + ((y + 1) % 2)
            for row, color in ((y, BLACK), (y + 6, WHITE)):
               self.board[row][column] = self.create_piece(DRAUGHT, color)
```

Этот вариант метода `CheckersBoard.populate_board()` взят из файла `gameboard3.py`. Он отличается от предыдущего тем, что фигура и цвет задаются константами, а не строковыми литералами, при вводе которых так легко опечататься. Кроме того, для создания фигур используется новая фабрика `create_piece()`.

Другая реализация `CheckersBoard.populate_board()` находится в файле `gameboard4.py` (не показана) – в ней используется хитроумная комбинация спискового включения и двух функций `itertools`.

```
class AbstractBoard:

   __classForPiece = {(DRAUGHT, BLACK): BlackDraught,
         (PAWN, BLACK): BlackChessPawn,
         ...
         (QUEEN, WHITE): WhiteChessQueen}
   ...
   def create_piece(self, kind, color):
      return AbstractBoard.__classForPiece[kind, color]()
```

Эта версия фабрики `create_piece()` (конечно, тоже из файла `gameboard3.py`) – метод класса `AbstractBoard`, которому наследуют `CheckersBoard` и `ChessBoard`. Она принимает две константы и ищет их в статическом (то есть определенном на уровне класса) словаре, ключами которого являются 2-кортежи (`piece kind, color`), а значениями – объектов классов. Найденное значение – класс – сразу же вызывается (с помощью оператора вызова `()`), и возвращается объект, представляющий фигуру.

Хранящиеся в словаре классы можно было бы написать вручную (как в файле `gameboard1.py`) или создавать динамически рискованным способом (как в `gameboard2.py`). Однако в `gameboard3.py` мы создаем их динамически, но в то же время безопасно, не используя ни `eval()`, ни `exec()`.

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
   char = chr(code)
   name = unicodedata.name(char).title().replace(" ", "")
   if name.endswith("sMan"):
      name = name[:-4]
   new = make_new_method(char)
   Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
   setattr(sys.modules[__name__], name, Class) # Можно сделать лучше!
```

Структурно этот код аналогичен приведенному выше коду создания четырнадцати подклассов фигур (стр. 33). Только вместо использования `eval()` и `exec()` мы поступили более безопасно.

Зная символ и название, мы создаем новую функцию (с именем `new()`), обращаясь к написанной нами функции `make_new_method()`. Затем с помощью встроенной функции `type()` создается новый класс. Чтобы таким способом создать класс, мы должны передать имя типа, кортеж, содержащий базовые классы (в данном случае только один – `Piece`), и словарь атрибутов класса. Здесь значением атрибута `__slots__` является пустой кортеж (чтобы предотвратить создание ненужного нам закрытого атрибута `__dict__` в экземплярах классов), а в атрибут-метод `__new__` мы записываем нашу функцию `new()`.

И наконец, с помощью встроенной функции `setattr()` мы добавляем во вновь созданный класс (`Class`) текущий модуль (`sys.modules[__name__]`) – в виде атрибута с именем `name` (например, `"WhiteChessPawn"`). В файле `gameboard4.py` последняя строка этого фрагмента выглядит более изящно:

```
globals()[name] = Class
```

Здесь мы получили ссылку на словарь глобальных объектов и добавили в него новый объект, ключ которого – значение, хранящееся в `name`, а значение – созданный нами класс `Class`. Результат точно такой же, как при использовании `setattr()` в файле `gameboard3.py`.

```
def make_new_method(char): # Приходится каждый раз создавать метод заново
   def new(Class): # Нельзя использовать super() или super(Piece, Class)
      return Piece.__new__(Class, char)
   return new
```

Эта функция служит для создания функции `new()` (которая станет методом `__new__()` класса). Мы не можем вызвать `super()`, поскольку в момент, когда создается `new()`, нет никакого контекста класса, к которому могла бы обратиться функция `super()`. Обратите внима-

ние, что в классе `Piece` (стр. 31) нет метода `__new__()`, зато он есть в его базовом классе (`str`), и именно этот метод и будет вызван.

Кстати, строку `new = make_new_method(char)` можно было бы удалить из показанного выше фрагмента кода, равно как и саму функцию `make_new_method()`, если заменить обращение к `make_new_method()` такой конструкцией:

```
new = (lambda char: lambda Class: Piece.__new__(Class, char))(char)
new.__name__ = "__new__"
```

Здесь мы создаем функцию, которая создает функцию, и сразу же вызываем внешнюю функцию, передавая ей `char`, чтобы та вернула функцию `new()` (этот код включен в файл `gameboard4.py`).

Все лямбда-функции называются `"lambda"`, что не очень удобно во время отладки. Поэтому здесь, создав функцию, мы явно присваиваем ей нужное имя.

```
def populate_board(self):
   for row, color in ((0, BLACK), (7, WHITE)):
      for columns, kind in (((0, 7), ROOK), ((1, 6), KNIGHT),
            ((2, 5), BISHOP), ((3,), QUEEN), ((4,), KING)):
         for column in columns:
            self.board[row][column] = self.create_piece(kind, color)
   for column in range(8):
      for row, color in ((1, BLACK), (6, WHITE)):
         self.board[row][column] = self.create_piece(PAWN, color)
```

Для полноты выше показан метод `ChessBoard.populate_board()` из файла `gameboard3.py` (и `gameboard4.py`). В нем используются константы, задающие цвета и фигуры (которые можно было бы прочитать из файла или получить из параметров меню, а не зашивать в код). В файле `gameboard3.py` вызывается показанная выше версия фабричной функции `create_piece()` (стр. 34), а в файле `gameboard4.py` – окончательная ее версия.

```
def create_piece(kind, color):
   color = "White" if color == WHITE else "Black"
   name = {DRAUGHT: "Draught", PAWN: "ChessPawn", ROOK: "ChessRook",
           KNIGHT: "ChessKnight", BISHOP: "ChessBishop",
           KING: "ChessKing", QUEEN: "ChessQueen"}[kind]
   return globals()[color + name]()
```

Это вариант фабричной функции `create_piece()` из файла `gameboard4.py`. В ней используются те же константы, что в `gameboard3.py`, но вместо создания словаря объектов классов она

динамически ищет нужный класс в словаре, возвращенном встроенной функцией `globals()`. Найденный объект класса сразу же вызывается и возвращается созданный в результате объект, представляющий фигуру.

1.4. Паттерн Прототип

Паттерн Прототип применяется для создания нового объекта путем клонирования исходного с последующей модификацией клона.

Как мы видели выше, а особенно в предыдущем разделе, Python поддерживает много разных способов создания объектов – даже в случае, когда тип становится известен только во время выполнения, и даже когда имеется лишь имя типа.

```
class Point:

   __slots__ = ("x", "y")

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

При таком классическом определении класса `Point` создать новую точку можно семью способами:

```
def make_object(Class, *args, **kwargs):
   return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Опасно
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12

# Можно было использовать любой из объектов от point1 до point6
point7 = point1.__class__(7, 14)
```

Точка `point1` создается традиционным способом (статически), используя объект класса `Point` как конструктор[6]. Все остальные точ-

[6] Строго говоря, метод `__init__()` является инициализатором, а метод `__new__()` – конструктором. Но поскольку мы почти всегда используем `__init__()` и очень редко `__new__()`, то в этой книге будем называть оба конструкторами.

ки создаются динамически, причем при создании `point2`, `point3` и `point4` имя класса передается в качестве параметра. Поскольку создание `point3` (и `point4`) выглядит вполне понятно, нет нужды использовать опасную функцию `eval()` для создания экземпляров (как в случае `point2`). Точка `point4` создается так же, как `point3`, только синтаксис более приятный благодаря встроенной в Python функции `globals()`. Для создания точки `point5` используется обобщенная функция `make_object()`, которая принимает объект класса и соответствующие аргументы. При создании `point6` применяется классический подход на основе прототипа: сначала мы клонируем существующий объект, а затем инициализируем и конфигурируем его. Для создания `point7` мы берем объект класса точки `point1` и передаем ему другие аргументы.

На примере `point6` мы видим, что в Python уже встроена поддержка прототипов в виде функции `copy.deepcopy()`. Однако пример `point7` показывает, что в Python есть средства и получше: вместо того чтобы клонировать существующий объект и модифицировать клон, Python предоставляет доступ к объекту класса имеющегося объекта, и таким образом мы можем создать новый объект непосредственно, что гораздо эффективнее клонирования.

1.5. Паттерн Одиночка

Паттерн Одиночка (Синглтон) применяется, когда необходим класс, у которого должен быть единственный экземпляр во всей программе.

В некоторых объектно-ориентированных языках создать одиночку на удивление трудно, но Python к ним не относится. В сборнике рецептов для Python (`code.activestate.com/recipes/langs/python/`) приводится простой класс `Singleton`, которому любой класс может унаследовать, чтобы стать одиночкой. И класс `Borg`, который дает тот же результат совсем другим способом.

Однако самый простой способ получить функциональность одиночки в Python – создать модуль с глобальным состоянием, которое хранится в закрытых переменных, и предоставить открытые функции для доступа к нему. Например, в примере `currency` из главы 7 нам понадобится функция, которая возвращает словарь курсов валют (ключ – название валюты, значение – обменный курс). Возможно, эту функцию понадобится вызывать несколько раз, но в большинстве случаев извлекать откуда-то курсы нужно единожды. Реализовать это поможет паттерн Одиночка.

```
_URL = "http://www.bankofcanada.ca/stats/assets/csv/fx-seven-day.csv"

def get(refresh=False):
   if refresh:
      get.rates = {}
   if get.rates:
      return get.rates
   with urllib.request.urlopen(_URL) as file:
      for line in file:
         line = line.rstrip().decode("utf-8")
         if not line or line.startswith(("#", "Date")):
            continue
         name, currency, *rest = re.split(r"\s*,\s*", line)
         key = "{} ({})".format(name, currency)
         try:
            get.rates[key] = float(rest[-1])
         except ValueError as err:
            print("error {}: {}".format(err, line))
   return get.rates
get.rates = {}
```

Этот код взят из модуля `currency/Rates.py` (предложения импорта, как обычно, опущены). Здесь мы создаем словарь `rates` в виде атрибута функции `Rates.get()` – это наше закрытое значение. Когда открытая функция `get()` вызывается в первый раз (а также при вызове с параметром `refresh=True`), мы загружаем список курсов; в противном случае просто возвращаем последние загруженные курсы. В классе вообще нет необходимости, тем не менее, мы все же получили одиночное значение – курсы валют – и без труда могли бы добавить другие, столь же одинокие.

Все порождающие паттерны проектирования реализуются на Python тривиально. Паттерн Одиночка можно реализовать непосредственно с помощью модуля, а паттерн Прототип вообще ни к чему (хотя его и можно реализовать с помощью модуля `copy`), так как Python дает динамический доступ к объектам классов. Из порождающих паттернов наиболее полезны Фабрика и Построитель, и реализовать их можно несколькими способами. Умея создавать простые объекты, мы часто испытываем потребность в более сложных, которые можно получить путем композиции или адаптации существующих. Как это делается, мы рассмотрим в следующей главе.

ГЛАВА 2. Структурные паттерны проектирования в Python

Структурные паттерны проектирования описывают, как из одних объектов составляются другие, более крупные. Рассматриваются три основных круга вопросов: адаптация интерфейсов, добавление функциональности и работа с коллекциями объектов.

2.1. Паттерн Адаптер

Паттерн Адаптер описывает технику адаптации интерфейса. Задача состоит в том, чтобы один класс мог воспользоваться другим – с несовместимым интерфейсом – без внесения каких-либо изменений в оба класса. Это полезно, например, когда требуется воспользоваться не подлежащим изменению классом в контексте, на который он не был рассчитан.

Допустим, имеется простой класс `Page`, который можно использовать для отрисовки страницы, зная название, текстовые абзацы и имея экземпляр класса отрисовщика. (Весь код, демонстрируемый в этом разделе, находится в файле `render1.py`.)

```
class Page:

   def __init__(self, title, renderer):
      if not isinstance(renderer, Renderer):
         raise TypeError("Expected object of type Renderer, got {}".
               format(type(renderer).__name__))
      self.title = title
      self.renderer = renderer
      self.paragraphs = []

   def add_paragraph(self, paragraph):
      self.paragraphs.append(paragraph)

   def render(self):
```

```
        self.renderer.header(self.title)
        for paragraph in self.paragraphs:
            self.renderer.paragraph(paragraph)
        self.renderer.footer()
```

Класс `Page` ничего не знает о классе отрисовщика, кроме того, что он предоставляет интерфейс отрисовки страницы, то есть три метода: `header(`*`str`*`)`, `paragraph(`*`str`*`)` и `footer()`.

Мы хотим проверить, что переданный отрисовщик – экземпляр класса `Renderer`. Вот простое, но плохое решение: `assert isinstance(renderer, Renderer)`. У него два недостатка. Во-первых, возбуждается исключение `AssertionError`, а не более естественное и специфичное `TypeError`. Во-вторых, при запуске программы с флагом `-O` («оптимизировать») предложение `assert` игнорируется, и впоследствии будет возбуждено исключение `AttributeError` в методе `render()`. Написанное же нами предложение `if not isinstance(…)` правильно возбуждает исключение `TypeError` и работает вне зависимости от наличия флага `-O`.

У этого решения, на первый взгляд, есть проблема: вроде бы необходимо делать все отрисовщики подклассами базового класса `Renderer`. Если бы мы писали на C++, то так оно и было бы, да и на Python можно было бы создать такой класс. Однако Python-модуль `abc` (**a**bstract **b**ase **c**lass – абстрактный базовый класс) предлагает другой и более гибкий подход, в котором проверяемость интерфейса, свойственная абстрактным базовым классам, сочетается с гибкостью динамической типизации (duck typing). Это означает, что мы можем создавать объекты, которые гарантированно согласованы с некоторым интерфейсом (то есть предоставляют определенный API), но не являются подклассами заранее известного базового класса.

```
class Renderer(metaclass=abc.ABCMeta):

    @classmethod
    def __subclasshook__(Class, Subclass):
        if Class is Renderer:
            attributes = collections.ChainMap(*(Superclass.__dict__
                for Superclass in Subclass.__mro__))
                methods = ("header", "paragraph", "footer")
                if all(method in attributes for method in methods):
                    return True
        return NotImplemented
```

В классе `Renderer` переопределен специальный метод `__subclasshook__()`. Этот метод используется встроенной функцией

isinstance(), чтобы определить, является ли объект, переданный ей в качестве первого аргумента, подклассом класса (или любого из классов, перечисленных в кортеже), переданного во втором аргументе.

Код довольно хитрый – и будет работать только в версии Python 3.3 – потому что в нем используется класс collections.ChainMap()[1]. Ниже объясняется, как этот код работает, но понимать это необязательно, так как всю трудную работу можно переложить на декоратор класса @Qtrac.has_methods, который включен в примеры к этой книге (и рассматривается ниже, стр. 47).

Специальный метод __subclasshook__() сначала проверяет, что класс объекта, от имени которого он вызван (Class), – Renderer; в противном случае сразу возвращается NotImplemented. Это означает, что поведение метода __subclasshook__ *не* наследуется подклассами. Мы поступаем так, потому что предполагаем, что подкласс добавляет к абстрактному классу новый критерий, а не новое поведение. Естественно, при желании можно было бы унаследовать поведение, просто явно вызвав Renderer.__subclasshook__() из переопределенного метода __subclasshook__().

Если бы мы вернули True или False, то механизм абстрактного базового класса прекратил бы работу, и метод вернул бы булево значение. Но, вернув NotImplemented, мы говорим, что не собираемся вмешиваться в обычное функционирование механизма наследования (подклассы, подклассы явно зарегистрированных классов, подклассы подклассов).

Если условие предложения if выполнено, то мы обходим все классы, которым наследует Subclass (в том числе его самого), получая их от его специального метода __mro__(), и обращаемся к закрытому словарю (__dict__) каждого из них. Это дает нам кортеж __dict__s, который мы немедленно распаковываем с помощью оператора распаковки последовательности (*), так что все словари передаются функции collections.ChainMap(). Эта функция принимает в качестве аргументов произвольное число отображений (например, словарей) и возвращает единственное представление отображения, как если бы все элементы находились в одном и том же отображении. Далее мы создаем кортеж методов, наличие которых хотели бы проверить. И наконец, мы перебираем все методы и смотрим, присутствуют ли они

[1] В файле render1.py и в модуле Qtrac.py, который используется в файле render1.py, имеются два варианта кода: для версии Python 3.3 и для более ранних.

в отображении attributes, ключами которого являются имена методов и свойства класса Subclass и всех его суперклассов. Если все методы присутствуют, то мы возвращаем True.

Отметим, что мы проверяем лишь, что подкласс (или какой-нибудь из его базовых классов) обладает атрибутами, имена которых совпадают с требуемыми методами, – даже свойство подойдет. Если нужно проверять только методы, то надо было бы добавить к условию method in attributes еще условие callable(method); но на практике это так редко составляет проблему, так что мы не станем этого делать.

Включать в класс метод __subclasshook__() для проверки интерфейсов – вещь очень полезная, но писать в каждом таком классе 10 строк сложного кода, которые отличаются только именем базового класса и перечнем методов, – как раз тот вид дублирования, от которого мы стремимся уйти. В следующем разделе мы создадим декоратор класса, благодаря чему сможем писать классы с проверкой интерфейсов, добавляя лишь две неповторяющихся строки кода. (В примерах имеется также файл render2.py, в котором этот декоратор используется.)

```
class TextRenderer:

   def __init__(self, width=80, file=sys.stdout):
      self.width = width
      self.file = file
      self.previous = False

   def header(self, title):
      self.file.write("{0:^{2}}\n{1:^{2}}\n".format(title,
             "=" * len(title), self.width))
```

Это начало простого класса, поддерживающего интерфейс отрисовщика страницы. Метод header() выводит переданный заголовок, центрированный в области заданной ширины, а в следующей строке выводит символ = под каждым символом заголовка.

```
def paragraph(self, text):
   if self.previous:
      self.file.write("\n")
   self.file.write(textwrap.fill(text, self.width))
   self.file.write("\n")
   self.previous = True

def footer(self):
   pass
```

Метод `paragraph()` пользуется стандартным библиотечным модулем `textwrap` для вывода переданного абзаца в области заданной ширины. Флаг `self.previous` нужен для того, чтобы каждый абзац отделялся пустой строкой от предыдущего. Метод `footer()` не делает ничего, но должен присутствовать, т. к. это часть интерфейса отрисовщика.

```
class HtmlWriter:

    def __init__(self, file=sys.stdout):
        self.file = file

    def header(self):
        self.file.write("<!doctype html>\n<html>\n")

    def title(self, title):
        self.file.write("<head><title>{}</title></head>\n".format(
              escape(title)))

    def start_body(self):
        self.file.write("<body>\n")

    def body(self, text):
        self.file.write("<p>{}</p>\n".format(escape(text)))

    def end_body(self):
        self.file.write("</body>\n")

    def footer(self):
        self.file.write("</html>\n")
```

Класс `HtmlWriter` умеет выводить простую HTML-страницу и берет на себя заботу об экранировании, вызывая функцию `html.escape()` (или `xml.sax.saxutil.escape()` в версии Python 3.2 или более ранней).

Хотя в этом классе имеются методы `header()` и `footer()`, ведут они себя не так, как обещал интерфейс отрисовщика страницы. Поэтому передать экземпляр `HtmlWriter` объекту `Page` в качестве отрисовщика (как мы поступили с `TextRenderer`) не получится.

Для решения проблемы можно было бы создать подкласс `HtmlWriter`, наделив его методами, которые должны быть у отрисовщика страницы. Но такой подход чересчур хрупкий, потому что в получившемся классе будут как методы `HtmlWriter`, так и методы интерфейса отрисовщика. Гораздо более элегантное решение – создать адаптер, то есть класс, который агрегирует имеющийся у нас класс

и предоставляет нужный интерфейс, беря на себя посреднические функции. Место этого класса-адаптера в структуре программы показано на рис. 2.1.

```
class HtmlRenderer:

   def __init__(self, htmlWriter):
      self.htmlWriter = htmlWriter

   def header(self, title):
      self.htmlWriter.header()
      self.htmlWriter.title(title)
      self.htmlWriter.start_body()

   def paragraph(self, text):
      self.htmlWriter.body(text)

   def footer(self):
      self.htmlWriter.end_body()
      self.htmlWriter.footer()
```

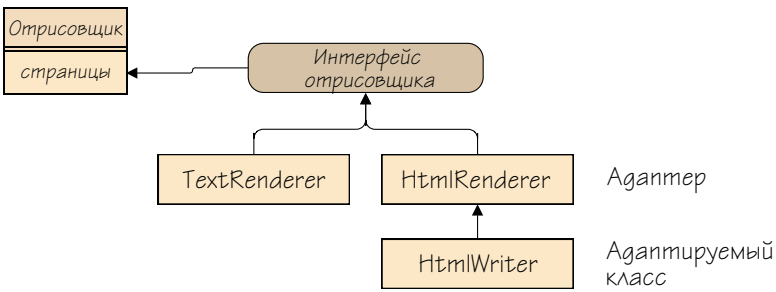

Рис. 2.1. Класс адаптера отрисовщика страницы в контексте

Это и есть наш адаптер. На этапе конструирования он принимает объект `htmlWriter` типа `HtmlWriter` и предоставляет методы, определенные в интерфейсе отрисовщика страницы. Вся содержательная работа делегируется агрегированному объекту `HtmlWriter`, а класс `HtmlRenderer` – не более чем обертка, предоставляющая новый интерфейс к этому объекту.

```
textPage = Page(title, TextRenderer(22))
textPage.add_paragraph(paragraph1)
textPage.add_paragraph(paragraph2)
textPage.render()

htmlPage = Page(title, HtmlRenderer(HtmlWriter(file)))
htmlPage.add_paragraph(paragraph1)
```

```
htmlPage.add_paragraph(paragraph2)
htmlPage.render()
```

Это два примера, показывающие, как создаются экземпляры класса `Page` с различными отрисовщиками. В данном случае мы задали в `TextRenderer` ширину по умолчанию 22 символа. А объекту `HtmlWriter`, который используется адаптером `HtmlRenderer`, мы передали файл, открытый для записи (его создание не показано), вместо подразумеваемого по умолчанию `sys.stdout`.

2.2. Паттерн Мост

Паттерн Мост используется в ситуациях, когда требуется отделить абстракцию (например, интерфейс или алгоритм) от ее реализации.

Традиционный подход без использования паттерна Мост состоит в том, чтобы создать один или несколько абстрактных базовых классов, а затем предоставить две или более конкретных реализации каждого базового класса. А паттерн Мост предлагает создать две независимых иерархии классов: «абстрактную», которая определяет операции (например, интерфейс и алгоритмы верхнего уровня) и конкретную, предоставляющую реализацию, которые в конечном итоге и будут вызваны из абстрактных операций. «Абстрактный» класс агрегирует экземпляр одного из конкретных классов с реализацией – и этот последний служит *мостом* между абстрактным интерфейсом и конкретными операциями.

В рассмотренном выше примере паттерна Адаптер мы могли сказать, что класс `HtmlRenderer` использует паттерн Мост, потому что он агрегирует `HtmlWriter` для предоставления средств отрисовки.

В этом разделе мы хотим создать класс для рисования столбчатых диаграмм с помощью конкретного алгоритма, но собственно рисование оставить другим классам. В программе `barchart1.py` эта идея реализована с помощью паттерна Мост.

```
class BarCharter:

   def __init__(self, renderer):
      if not isinstance(renderer, BarRenderer):
         raise TypeError("Expected object of type BarRenderer, got {}".
                  format(type(renderer).__name__))
      self.__renderer = renderer

   def render(self, caption, pairs):
      maximum = max(value for _, value in pairs)
```

```
        self.__renderer.initialize(len(pairs), maximum)
        self.__renderer.draw_caption(caption)
        for name, value in pairs:
            self.__renderer.draw_bar(name, value)
        self.__renderer.finalize()
```

Класс `BarCharter` реализует алгоритм рисования столбчатой диаграммы (в методе `render()`), который пользуется переданной ему реализацией отрисовщика, согласованной с интерфейсом построения диаграмм. В этом интерфейсе определены методы `initialize(`*`int, int`*`)`, `draw_caption(`*`str`*`)`, `draw_bar(`*`str, int`*`)` и `finalize()`.

Как и в предыдущем разделе, мы с помощью функции `isinstance()`, проверяем, что переданный объект `renderer` поддерживает нужный нам интерфейс, – не требуя, чтобы все отрисовщики диаграмм наследовали общему базовому классу. Но вместо класса из 10 строк, как раньше, мы создали класс проверки интерфейса всего из двух строчек:

```
@Qtrac.has_methods("initialize", "draw_caption", "draw_bar", "finalize")
class BarRenderer(metaclass=abc.ABCMeta): pass
```

Здесь с классом `BarRenderer` ассоциирован метакласс, необходимый для работы с модулем `abc`. Этот класс затем передается функции `Qtrac.has_methods()`, которая возвращает декоратор класса. Декоратор добавляет в класс специальный метод класса `__subclasshook__()`. И вот этот новый метод как раз и проверяет наличие указанных методов при передаче `BarRenderer` в качестве типа при вызове `isinstance()`. (Читателям, не знакомым с концепцией декоратора класса, рекомендуется забежать вперед, прочитать раздел 2.4, а особенно подраздел 2.4.2, а затем вернуться сюда.)

```
def has_methods(*methods):
    def decorator(Base):
        def __subclasshook__(Class, Subclass):
            if Class is Base:
                attributes = collections.ChainMap(*(Superclass.__dict__
                        for Superclass in Subclass.__mro__))
                if all(method in attributes for method in methods):
                    return True
            return NotImplemented
        Base.__subclasshook__ = classmethod(__subclasshook__)
        return Base
    return decorator
```

Функция `has_methods()` из модуля `Qtrac.py` захватывает требуемые методы и создает функцию декоратора класса, которую затем и

возвращает. Сам декоратор создает функцию `__subclasshook__()` и добавляет ее в базовый класс как метод класса с помощью встроенной функции `classmethod()`. Код функции `__subclasshook__()` по существу такой же, как мы рассматривали выше (стр. 42), только теперь вместо зашитого к код базового класса мы используем декорируемый класс (`Base`), а вместо зашитого набора имен методов – имена, переданные декоратору класса (`methods`).

Той же функциональности проверки наличия методов можно достичь и путем наследования обобщенному абстрактному базовому классу. Например:

```
class BarRenderer(Qtrac.Requirer):
   required_methods = {"initialize", "draw_caption", "draw_bar",
                       "finalize"}
```

Этот фрагмент взят из файла `barchart3.py`. Класс `Qtrac.Requirer` (не показан, но есть в файле `Qtrac.py`) – это абстрактный базовый класс, который выполняет те же проверки, что и декоратор класса `@has_methods`.

```
def main():
   pairs = (("Mon", 16), ("Tue", 17), ("Wed", 19), ("Thu", 22),
            ("Fri", 24), ("Sat", 21), ("Sun", 19))
   textBarCharter = BarCharter(TextBarRenderer())
   textBarCharter.render("Forecast 6/8", pairs)

   imageBarCharter = BarCharter(ImageBarRenderer())
   imageBarCharter.render("Forecast 6/8", pairs)
```

Эта функция `main()` подготавливает данные, создает два построителя столбчатых диаграмм – с разными реализациями алгоритма отрисовки – и изображает данные на диаграммах. Результаты показаны на рис. 2.2, а интерфейс и классы – на рис. 2.3.

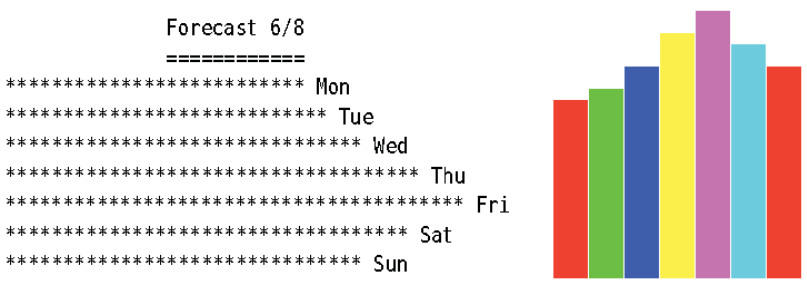

Рис. 2.2. Примеры текстовой и графической столбчатых диаграмм

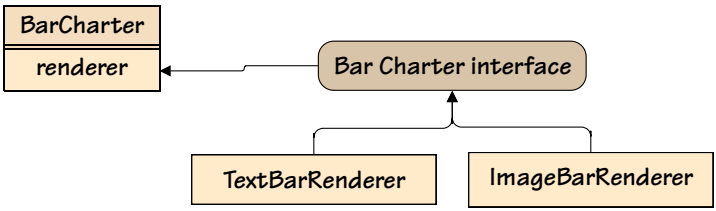

Рис. 2.3. Интерфейс построителя столбчатых диаграмм и реализующие его классы

```
class TextBarRenderer:

   def __init__(self, scaleFactor=40):
      self.scaleFactor = scaleFactor

   def initialize(self, bars, maximum):
      assert bars > 0 and maximum > 0
      self.scale = self.scaleFactor / maximum

   def draw_caption(self, caption):
      print("{0:^{2}}\n{1:^{2}}".format(caption, "=" * len(caption),
            self.scaleFactor))

   def draw_bar(self, name, value):
      print("{} {}".format("*" * int(value * self.scale), name))

   def finalize(self):
      pass
```

Этот класс реализует интерфейс построителя диаграмм и выводит текстовое представление в `sys.stdout`. Естественно, было бы несложно добавить возможность задания файла пользователем, а в системах Unix использовать для рисования псевдографику и раскрасить диаграмму в разные цвета.

Отметим, что хотя метод `finalize()` класса `TextBarRenderer` ничего не делает, присутствовать он должен, потому что таково требование интерфейса.

Хотя стандартная библиотека Python очень обширна («батарейки в комплекте»), в ней есть одно на удивление серьезное упущение: не существует пакета для чтения и записи изображений в стандартных растровых и векторных форматах. Можно воспользоваться сторонней библиотекой – либо поддерживающей несколько форматов типа Pillow (`github.com/python-imaging/Pillow`), либо рассчитанной на один формат, либо даже какой-нибудь библиотекой для построения ГИП. Другое решение – написать свою библиотеку обработки

изображений, этим мы займемся позже в разделе 3.12. Если мы готовы ограничиться только форматом GIF (плюс PNG, если Python поставляется вместе с Tk/Tcl 8.6), то можно взять Tkinter[2].

В классе `ImageBarRenderer` из файла `barchart1.py` используется модуль `cyImage` (или, если его нет, модуль `Image`). Мы будем называть оба модуля `Image`, если существующие различия несущественны. Эти модули входят в состав кода к этой книге и будут рассмотрены ниже (`Image` в разделе 3.12, `cyImage` в подразделе 5.2.2). Для полноты в состав примеров включен также файл `barchart2.py` – версия `barchart1.py`, в которой вместо `cyImage` или `Image` используется Tkinter, но код этой версии в тексте не приводится.

Поскольку класс `ImageBarRenderer` сложнее, чем `TextBarRenderer`, мы сначала рассмотрим его статические данные, а затем все методы по очереди.

```
class ImageBarRenderer:

   COLORS = [Image.color_for_name(name) for name in ("red", "green",
             "blue", "yellow", "magenta", "cyan")]
```

В модуле `Image` пиксели представлены в виде 32-разрядных целых без знака, в которых закодированы четыре компоненты цвета: альфа-канал (прозрачность), красная, зеленая и синяя. Модуль предоставляет функцию `Image.color_for_name()`, которая принимает название цвета – либо из файла `rgb.txt`, являющегося частью X11 (например, `"sienna"`), либо в HTML-формате (например, `"#A0522D"`) – и возвращает соответствующее целое без знака.

В строке выше создается список цветов, которые мы будем использовать при построении столбчатых диаграмм.

```
def __init__(self, stepHeight=10, barWidth=30, barGap=2):
   self.stepHeight = stepHeight
   self.barWidth = barWidth
   self.barGap = barGap
```

Этот метод позволяет задать параметры, определяющие, как будут рисоваться столбики диаграммы.

```
def initialize(self, bars, maximum):
   assert bars > 0 and maximum > 0
   self.index = 0
```

[2] Отметим, что Tkinter допускает обработку изображений только в главном потоке (потоке ГИП). Чтобы распараллелить обработку, понадобится другой подход, мы познакомимся с ним ниже в разделе 4.1.

```
    color = Image.color_for_name("white")
    self.image = Image.Image(bars * (self.barWidth + self.barGap),
            maximum * self.stepHeight, background=color)
```

Этот метод (и следующий за ним) должен присутствовать, потому что является частью интерфейса построителя диаграмм. Здесь мы создаем новое изображение, размер которого определяется числом столбиков, их шириной и максимальной высотой. Первоначально изображение закрашено белым цветом.

В переменной `self.index` хранится индекс текущего столбика (нумерация начинается с 0).

```
def draw_caption(self, caption):
    self.filename = os.path.join(tempfile.gettempdir(),
            re.sub(r"\W+", "_", caption) + ".xpm")
```

Модуль `Image` не поддерживает рисование текста, поэтому переданную надпись мы используем для формирования имени файла с изображением.

Модуль `Image` изначально поддерживает два графических формата: XBM (`.xbm`) для монохромных изображений и XPM (`.xpm`) – для цветных (если установлен модуль PyPNG – см. `pypi.python.org/pypi/pypng` – то модуль `Image` поддерживает также формат PNG (`.png`)). В данном случае мы выбрали цветной формат XPM, поскольку наша диаграмма должна быть цветной, а этот формат поддерживается всегда.

```
def draw_bar(self, name, value):
    color = ImageBarRenderer.COLORS[self.index %
             len(ImageBarRenderer.COLORS)]
    width, height = self.image.size
    x0 = self.index * (self.barWidth + self.barGap)
    x1 = x0 + self.barWidth
    y0 = height - (value * self.stepHeight)
    y1 = height - 1
    self.image.rectangle(x0, y0, x1, y1, fill=color)
    self.index += 1
```

Этот метод выбирает цвет из последовательности `COLORS` (возвращаясь к ее началу, если столбиков больше, чем цветов). Затем он вычисляет координаты (левого верхнего и правого нижнего угла) текущего (с индексом `self.index`) столбика и просит объект `self.image` (типа `Image.Image`) нарисовать в области диаграммы прямоугольник с указанными координатами и цветом. После этого индекс увеличивается на 1, чтобы перейти к следующему столбику.

```
def finalize(self):
    self.image.save(self.filename)
    print("wrote", self.filename)
```

Здесь мы просто сохраняем изображение и сообщаем об этом пользователю.

Очевидно, что реализации `TextBarRenderer` и `ImageBarRenderer` отличаются радикально. И тем не менее, каждую можно использовать как мост для предоставлению классу `BarCharter` конкретного объекта для рисования диаграммы.

2.3. Паттерн Компоновщик

Паттерн Компоновщик позволяет единообразно обрабатывать объекты, образующие иерархию, вне зависимости от того, содержат они другие объекты или нет. Такие объекты называются *составными*. В классическом решении составные объекты – как отдельные, так и коллекции – имеют один и тот же базовый класс. У составных и несоставных объектов обычно одни и те же основные методы, но составные объекты добавляют методы для поддержки добавления, удаления и перебора дочерних объектов.

Этот паттерн часто применяется в программах рисования, например Inkscape, для поддержки группировки и разгруппировки. Его полезность в таких ситуациях объясняется тем, что подлежащие группировке или разгруппировке компоненты могут быть как одиночными (например, прямоугольник), так и составными (например, рожица, составленная из нескольких разных фигур).

В качестве практического примера рассмотрим функцию `main()`, которая создает несколько простых объектов и несколько составных, а затем все их распечатывает. Код взят из файла `stationery1.py`, после него показан результат работы программы.

```
def main():
    pencil = SimpleItem("Карандаш", 0.40)
    ruler = SimpleItem("Линейка", 1.60)
    eraser = SimpleItem("Ластик", 0.20)
    pencilSet = CompositeItem("Набор карандашей", pencil, ruler, eraser)
    box = SimpleItem("Коробка", 1.00)
    boxedPencilSet = CompositeItem("Набор карандашей в коробке",
                                   box, pencilSet)
    boxedPencilSet.add(pencil)
    for item in (pencil, ruler, eraser, pencilSet, boxedPencilSet):
        item.print()
```

```
$0.40 Карандаш
$1.60 Линейка
$0.20 Ластик
$2.20 Набор карандашей
      $0.40 Карандаш
      $1.60 Линейка
      $0.20 Ластик
$3.60 Набор карандашей в коробке
      $1.00 Коробка
      $2.20 Набор карандашей
            $0.40 Карандаш
            $1.60 Линейка
            $0.20 Ластик
      $0.40 Карандаш
```

У каждого объекта `SimpleItem` есть название и цена, а у каждого `CompositeItem` – название и произвольное число содержащихся в нем объектов `SimpleItem` – или `CompositeItem` – то есть составные объекты могут быть вложенными до любой глубины. Цена составного объекта равна сумме цен входящих в него объектов.

В данном примере набор карандашей состоит из карандаша, линейки и ластика. Для получения набора карандашей в коробке мы сначала создаем коробку и вложенный в нее набор карандашей, а затем добавляем еще один карандаш. Иерархия набора карандашей в коробке показана на рис. 2.4.

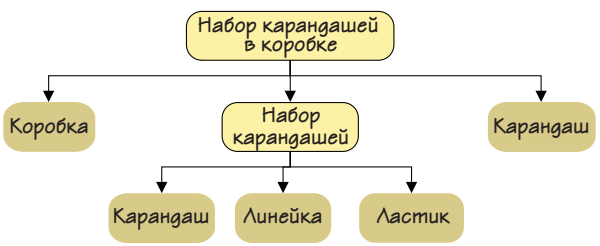

Рис. 2.4. Иерархия составных и несоставных объектов

Мы рассмотрим две разных реализации паттерна Компоновщик – сначала классическую, а потом такую, в которой для представления составных и несоставных объектов используется единственный класс.

2.3.1. Классическая иерархия составных и несоставных объектов

Классическое решение основано на наличии абстрактного базового класса, общего для всех видов объектов (составных и несоставных).

Иерархия классов показана на рис. 2.5. Начнем рассмотрение с базового класса `AbstractItem`.

```
class AbstractItem(metaclass=abc.ABCMeta):

    @abc.abstractproperty
    def composite(self):
        pass

    def __iter__(self):
        return iter([])
```

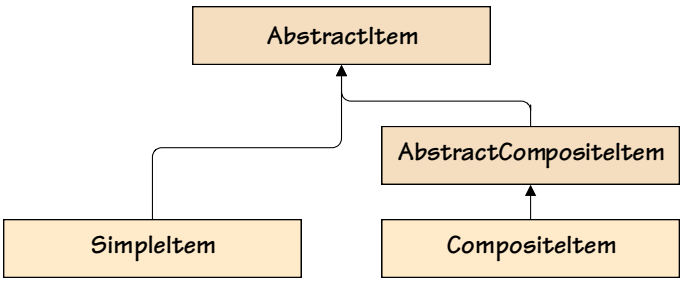

Рис. 2.5. Иерархия составных и несоставных классов

Мы хотим, чтобы все подклассы умели сообщать, составные они или нет. Кроме того, все подклассы должны быть итерируемыми, причем по умолчанию должен возвращаться итератор пустой последовательности.

Поскольку класс `AbstractItem` имеет по меньшей мере один абстрактный метод или свойство, создавать объект класса `AbstractItem` нельзя (кстати, начиная с версии Python 3.3, можно писать `@property @abstractmethod def` *`method`*`(…): …` вместо `@abstractproperty def` *`method`*`(…): …`).

```
class SimpleItem(AbstractItem):

    def __init__(self, name, price=0.00):
        self.name = name
        self.price = price

    @property
    def composite(self):
        return False
```

Класс `SimpleItem` используется для несоставных объектов. В нашем примере у объектов `SimpleItem` есть свойства `name` и `price`.

Поскольку SimpleItem наследует классу AbstractItem, он должен переопределить все абстрактные свойства и методы – в данном случае только свойство composite. Так как метод __iter__() класса AbstractItem не абстрактный, и мы его не переопределили, то работает версия из базового класса, которая возвращает итератор пустой последовательности. Это разумно, потому что объекты класса SimpleItem несоставные, но при этом позволяет работать с SimpleItem и CompositeItem единообразно (по крайней мере, в части итерирования), например, передавать такие объекты в любой комбинации функции типа itertools.chain().

```
def print(self, indent="", file=sys.stdout):
   print("{}${:.2f} {}".format(indent, self.price, self.name),
         file=file)
```

Мы добавили метод print(), чтобы можно было печать составные и несоставные объекты, выделяя вложенные объекты отступом соответствующего уровня.

```
class AbstractCompositeItem(AbstractItem):

   def __init__(self, *items):
      self.children = []
      if items:
         self.add(*items)
```

Этот класс является базовым для CompositeItem и предоставляет средства для добавления, удаления и итерирования. Создать экземпляр класса AbstractCompositeItem невозможно, потому что он наследует абстрактное свойство composite, но не реализует его.

```
def add(self, first, *items):
   self.children.append(first)
   if items:
      self.children.extend(items)
```

Этот метод принимает один или более объектов (как SimpleItem, так и CompositeItem) и добавляет их в список потомков данного составного объекта. Нельзя отказаться от параметра first и оставить только *items, потому что тогда мы получили бы возможность добавлять нуль объектов, и хотя здесь это безвредно, но могло бы замаскировать логическую ошибку в коде клиента. (Подробнее о распаковке – в частности, конструкции *items – см. врезку «Распаковка последовательностей и отображений» на стр. 24.) Кстати, мы ничего

не делаем, чтобы запретить циклические ссылки, например, добавление составного объекта в качестве собственного потомка.

Позже мы реализуем этот метод всего в одной строчке (стр. 59).

```
def remove(self, item):
   self.children.remove(item)
```

Удаление мы реализовали по-простому, разрешив удалять только по одному элементу. Конечно, удаляемый объект может быть составным, и тогда его удаление повлечет за собой удаление всех его потомков, их потомков и т. д.

```
def __iter__(self):
   return iter(self.children)
```

Реализовав специальный метод `__iter__()`, мы дали возможность перебирать потомков составных объектов в циклах `for`, списковых включениях и генераторах. Во многих случаях мы записали бы тело метода в виде `for item in self.children: yield item`, но поскольку `self.children` – последовательность (список), мы можем воспользоваться встроенной функцией `iter()`, которая сделает за нас всю работу.

```
class CompositeItem(AbstractCompositeItem):

   def __init__(self, name, *items):
      super().__init__(*items)
      self.name = name

   @property
   def composite(self):
      return True
```

Этот класс применяется для конкретных составных объектов. В нем определено свойство `name`, но все действия, связанные с составлением (добавление, удаление, обход дочерних объектов) оставлены базовому классу. Экземпляры `CompositeItem` можно создавать, потому что этот класс предоставляет реализацию абстрактного свойства `composite`, а никаких других абстрактных свойств или методов нет.

```
   @property
   def price(self):
      return sum(item.price for item in self)
```

Это допускающее только чтение свойство нуждается в пояснениях. В нем вычисляется цена составного объекта – путем суммирования цен

его потомков, цен их потомков (если потомок сам является составным объектом) – и так далее рекурсивно с использованием генераторного выражения в качестве аргумента встроенной функции `sum()`.

Выражение `for item in self` заставляет интерпретатор Python вызвать `iter(self)`, чтобы получить итератор для `self`. Это приводит к вызову специального метода `__iter__()`, который возвращает итератор, указывающий на `self.children`.

```
def print(self, indent="", file=sys.stdout):
   print("{}${:.2f} {}".format(indent, self.price, self.name),
         file=file)
   for child in self:
      child.print(indent + " ")
```

И снова мы предоставили вспомогательный метод `print()`, правда, к сожалению, первое предложение в нем – копия тела метода `SimpleItem.print()`.

В этом примере классов `SimpleItem` и `CompositeItem` достаточно для большинства сценариев. Но если требуется более детальная иерархия, то им – или их абстрактным базовым классам – можно унаследовать.

Показанные здесь классы `AbstractItem`, `SimpleItem`, `AbstractCompositeItem` и `CompositeItem` отлично работают. Но создается впечатление, что код длиннее, чем необходимо, а интерфейс недостаточно единообразный, поскольку у составных объектов имеются методы (`add()` и `remove()`), отсутствующие у несоставных объектов. Этими проблемами мы займемся в следующем подразделе.

2.3.2. Единый класс для составных и несоставных объектов

Для разработки четырех классов (двух абстрактных и двух конкретных) в предыдущем разделе потребовалось слишком много работы. И интерфейс у них не одинаковый, потому что методы `add()` и `remove()` имеются только у составных объектов. Если мы согласны смириться с небольшими накладными расходами – один пустой список на каждый несоставной объект и один атрибут типа `float` на каждый составной объект – то можно написать общий класс для представления как составных, так и несоставных объектов. Таким образом, мы получим единообразный интерфейс, так как сможем вызывать методы `add()` и `remove()` для любого объекта, а не только составного, получая разумное поведение.

В этом подразделе мы напишем новый класс `Item`, который может использоваться для представления составных и несоставных объектов, больше никаких классов нам не понадобится. Демонстрируемый здесь код взят из файла `stationery2.py`.

```
class Item:

   def __init__(self, name, *items, price=0.00):
      self.name = name
      self.price = price
      self.children = []
      if items:
         self.add(*items)
```

Аргументы метода `__init__()` выглядят не очень презентабельно, но это не страшно, поскольку, как мы скоро увидим, вызывающая программа и не должна обращаться к `Item()` для создания объектов.

У каждого объекта должно быть имя. У объектов есть также цена, но для нее мы предоставляем значение по умолчанию. Кроме того, у любого объекта может быть нуль или более дочерних объектов (`*items`), который хранятся в `self.children` – для несоставных объектов этот список пуст.

```
   @classmethod
   def create(Class, name, price):
      return Item(name, price=price)

   @classmethod
      def compose(Class, name, *items):
      return Item(name, *items)
```

Вместо того чтобы вызывать объекты класса для создания объектов, мы завели два вспомогательных фабричных метода, которые принимают более симпатичные аргументы и возвращают `Item`. Так, вместо `SimpleItem("Ruler", 1.60)` или `CompositeItem("Pencil Set", pencil, ruler, eraser)` мы пишем `Item.create("Ruler", 1.60)` и `Item.compose("Pencil Set", pencil, ruler, eraser)`. И теперь все объекты имеют один и тот же тип: `Item`. Естественно, пользователь может вызвать `Item()` напрямую, если захочет; например, `Item("Ruler", price=1.60)` и `Item("Pencil Set", pencil, ruler, eraser)`.

```
   def make_item(name, price):
      return Item(name, price=price)

   def make_composite(name, *items):
      return Item(name, *items)
```

Мы предоставили также две фабричные функции, которые делают то же самое, что методы класса. Такие функции удобны при использовании модулей. Например, если бы наш класс `Item` находился в модуле `Item.py`, то мы могли бы написать, скажем, `Item.make_item("Ruler", 1.60)` вместо `Item.Item.create("Ruler", 1.60)`.

```
@property
def composite(self):
    return bool(self.children)
```

Это свойство отличается от написанного ранее, потому что теперь любой объект может быть составным или несоставным. В случае класса `Item` составным будет объект, в котором список `self.children` не пуст.

```
def add(self, first, *items):
    self.children.extend(itertools.chain((first,), items))
```

Мы слегка изменили метод `add()`, применив более эффективное решение. Функция `itertools.chain()` принимает произвольное количество итерируемых объектов и возвращает единственный итерируемый объект, который по существу является конкатенацией всех переданных ей объектов.

Этот метод можно вызывать для любого объект, составного или нет. А если он вызван для несоставного объекта, то этот объект становится составным. У преобразования несоставного объекта в составной есть один тонкий побочный эффект – собственная цена объекта оказывается скрытой, потому что теперь его ценой является сумма цен дочерних объектов. Разумеется, возможны и другие проектные решения – например, оставить собственную цену.

```
def remove(self, item):
    self.children.remove(item)
```

Если удаляется последний потомок составного объекта, то этот объект становится несоставным. Но теперь его ценой является значение закрытого атрибута `self.__price`, а не сумма цен дочерних объектов (которых просто нет). Чтобы это решение всегда работало, мы задаем начальную цену любого объекта в методе `__init__()`.

```
def __iter__(self):
    return iter(self.children)
```

Этот метод возвращает итератор, указывающий на список потомков составного объекта или – если объект несоставной – на пустую последовательность.

```
@property
def price(self):
   return (sum(item.price for item in self) if self.children else
           self.__price)

@price.setter
def price(self, price):
   self.__price = price
```

Свойство `price` должно работать как для составных объектов (и тогда оно равно сумме цен дочерних объектов), так и для несоставных (и тогда это цена самого объекта).

```
def print(self, indent="", file=sys.stdout):
   print("{}${:.2f} {}".format(indent, self.price, self.name),
         file=file)
   for child in self:
      child.print(indent + " ")
```

И этот метод должен работать для составных и несоставных объектов, хотя его код такой же, как у метода `CompositeItem.print()`. При обходе несоставного объекта возвращаемый итератор указывает на пустую последовательность, поэтому не возникнет бесконечная рекурсия при переборе потомков.

Благодаря присущей Python гибкости мы можем легко создавать классы составных и несоставных объектов – либо в виде отдельных классов, чтобы минимизировать накладные расходы на хранение, либо в виде единого класса, чтобы иметь полностью единообразный интерфейс.

Ниже, при обсуждении паттерна Команда (раздел 3.2) мы встретимся еще с одной разновидностью паттерна Компоновщик.

2.4. Паттерн Декоратор

Вообще говоря, *декоратором* называется функция, которая принимает функцию в качестве своего единственного аргумента и возвращает новую функцию с таким же именем, как у исходной, но с расширенной функциональностью. Декораторы часто используются в каркасах (например, в веб-каркасах), чтобы упростить интеграцию пользовательских функций с каркасом.

Паттерн Декоратор настолько полезен, что в Python встроена специальная поддержка для него. В Python декорировать можно как функции, так и методы. Кроме того, Python поддерживает декораторы классов: функции, которые принимают класс в качестве аргумента и возвращают новый класс с таким же именем, как у исходного, но с дополненной функциональностью. Иногда декораторы классов удобно использовать как альтернативу производным классам.

Мы уже видели, что встроенную в Python функцию `property()` можно использовать как декоратор (например, в свойствах `composite` и `price` из предыдущего раздела). Несколько встроенных декораторов есть в стандартной библиотеке Python. Например, декоратор класса `@functools.total_ordering` можно применить к классу, в котором реализованы специальные методы `__eq__()` и `__lt__()` (на основе которых построены операторы сравнения == и <). В результате этот класс заменяется другим, в котором имеются и все остальные специальные методы, так что декорированный класс поддерживает весь спектр операторов сравнения (<, <=, ==, !=, => и >).

Декоратор может принимать только один аргумент – функцию, метод или класс, так что теоретически параметризовать декораторы невозможно. Но на практике это не такое уж серьезное ограничение, потому что, как мы увидим, есть возможность создать параметризованную фабрику декораторов, которая умеет возвращать декоратор функции, а его уже можно использовать для декорирования функции, метода или класса.

2.4.1. Декораторы функций и методов

Все декораторы функций (и методов) устроены одинаково. Прежде всего, они создают функцию-обертку (которую в этой книге мы всегда будем называть `wrapper()`). Из обертки мы должны вызвать исходную функцию. Однако мы вправе выполнить до этого вызова произвольную предобработку, затем получить результат вызова, выполнить произвольную постобработку и вернуть все, что пожелаем – исходный результат, модифицированный результат или еще что-нибудь. Наконец, мы возвращаем функцию-обертку в виде результата декоратора, и эта функция заменяет исходную с тем же именем.

Чтобы применить декоратор к функции, методу или классу, нужно написать знак `@` с тем же отступом, что соответствующее предложение `def` или `class`, а сразу за ним указать имя декоратора. Ничто не мешает создавать вложенные декораторы, то есть применить декоратор

к уже декорированной функции и так сколь угодно много раз – как показано на рис. 2.6 (мы приведем пример чуть ниже).

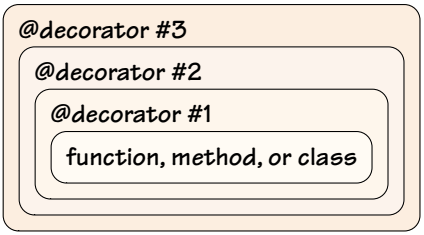

Рис. 2.6. Вложенные декораторы

```
@float_args_and_return
def mean(first, second, *rest):
   numbers = (first, second) + rest
   return sum(numbers) / len(numbers)
```

Здесь мы воспользовались декоратором `@float_args_and_return` (его код приведен ниже) для декорирования функции `mean()`. Исходная функция `mean()` принимает два или более числовых аргументов и возвращает их среднее арифметическое в виде `float`. *Декорированная* же функция `mean()` – которая также называется `mean()`, поскольку заменяет исходную, – может принимать два или более аргументов любого типа, которые преобразует к типу `float`. Не будь декоратора, обращение `mean(5, "6", "7.5")` привело бы к исключению `TypeError`, потому что нельзя складывать `int` и `str`, но декорированная версия отлично с этим справляется, так как вызовы `float("6")` и `float("7.5")` порождают допустимые числа.

Кстати говоря, синтаксис декоратора – не более чем синтаксическая глазурь. Приведенный выше код можно было бы записать и так:

```
def mean(first, second, *rest):
   numbers = (first, second) + rest
   return sum(numbers) / len(numbers)
mean = float_args_and_return(mean)
```

Здесь мы создали функцию без декоратора, а затем заменили ее декорированной версией, вызвав декоратор самостоятельно. Хотя декораторы – вещь очень удобная, иногда необходимо вызывать их напрямую. Пример мы увидим в конце этого раздела, когда будем вызывать встроенный декоратор `@property` в функции `ensure()` (стр. 70). Мы и раньше делали нечто подобное, когда вызывали встроенный декоратор `@classmethod` в функции `has_methods()` (стр. 47).

```
def float_args_and_return(function):
   def wrapper(*args, **kwargs):
      args = [float(arg) for arg in args]
      return float(function(*args, **kwargs))
   return wrapper
```

Функция `float_args_and_return()` – это декоратор функции, поэтому она принимает функцию в качестве своего единственного аргумента. Функции-обертки обычно принимают аргументы `*args` и `**kwargs`, то есть все что угодно. (См. врезку «Распаковка последовательностей и отображений» на стр. 24). Все ограничения на аргументы должны обрабатываться исходной (обернутой) функцией, мы должны лишь передать ей все, что получили.

В данном случае функция-обертка заменяет все полученные позиционные аргументы списком чисел с плавающей точкой. После этого мы вызываем исходную функцию, передавая ей модифицированный список `*args` и преобразуем результат в число типа `float`, которой и возвращаем.

Созданная обертка возвращается в качестве результата декоратора.

К сожалению, в том виде, как написано, атрибут `__name__` возвращенной декорированной функции содержит строку `"wrapper"`, а не имя исходной функции, и строка документации тоже исчезла, даже если в исходной функции она была. Так что замена оказалась неидеальной. Чтобы исправить это упущение, в стандартной библиотеке Python имеется декоратор `@functools.wraps`, который можно использовать для декорирования функции-обертки внутри декоратора. Он гарантирует, что атрибуты `__name__` и `__doc__` обертки содержат те же значения, что имя и строка документации исходной функции.

```
def float_args_and_return(function):
   @functools.wraps(function)
   def wrapper(*args, **kwargs):
      args = [float(arg) for arg in args]
      return float(function(*args, **kwargs))
   return wrapper
```

Это еще одна версия декоратора. В ней используется декоратор `@functools.wraps`, благодаря которому у созданной внутри декоратора функции `wrapper()` атрибут `__name__` содержит имя переданной функции (`"mean"`), а строка документации (в данном случае пустая) функции-обертки совпадает со строкой документации исходной функции. Рекомендуется всегда использовать декоратор `@functools.wraps`, потому что тогда имена декорированных функций

в трассировке будут показываться правильно (а не как "wrapper"), и будет доступ к строкам документации исходных функций.

```
@statically_typed(str, str, return_type=str)
def make_tagged(text, tag):
   return "<{0}>{1}</{0}>".format(tag, escape(text))

@statically_typed(str, int, str) # Will accept any return type
def repeat(what, count, separator):
   return ((what + separator) * count)[:-len(separator)]
```

Функция `statically_typed()`, применяемая для декорирования функций `make_tagged()` и `repeat()`, – это фабрика декораторов, то есть функция, порождающая декораторы. Сама она декоратором не является, потому что не принимает функцию, метод или класс в качестве единственного параметра. Но в данном случае мы хотим параметризовать декоратор, задав количество и типы позиционных аргументов, которые может принимать декорированная функция (и факультативно задать еще и тип возвращаемого ей значения), а эти характеристики от функции к функции меняются. Поэтому мы написали функцию `statically_typed()`, которая принимает интересующие нас параметры – по одному типу на каждый позиционный аргумент и необязательный ключевой аргумент для задания типа возвращаемого значения – и возвращает декоратор.

Итак, встретив в программе конструкцию `@statically_typed(…)`, Python вызовет эту функцию с указанными аргументами, а возвращенную ей функцию будет использовать как декоратор для следующей далее функции (в нашем примере `make_tagged()` или `repeat()`).

Фабрики декораторов создаются по единому образцу. Сначала мы создаем функцию-декоратор, а внутри нее функцию-обертку; обертка устроена так же, как раньше. Как обычно, в самом конце обертка возвращает результат исходной функции, быть может, модифицированный или подмененный. А функция-декоратор в конце возвращает обертку. И наконец, фабрика декораторов возвращает сам декоратор.

```
def statically_typed(*types, return_type=None):
   def decorator(function):
      @functools.wraps(function)
      def wrapper(*args, **kwargs):
         if len(args) > len(types):
            raise ValueError("too many arguments")
         elif len(args) < len(types):
            raise ValueError("too few arguments")
```

```
            for i, (arg, type_) in enumerate(zip(args, types)):
                if not isinstance(arg, type_):
                    raise ValueError("argument {} must be of type {}"
                            .format(i, type_.__name__))
            result = function(*args, **kwargs)
            if (return_type is not None and
                not isinstance(result, return_type)):
                raise ValueError("return value must be of type {}".format(
                    return_type.__name__))
            return result
        return wrapper
    return decorator
```

Здесь мы начинаем с создания функции-декоратора. Мы назвали ее `decorator()`, но вообще-то имя не играет роли. Внутри функции-декоратора создается обертка – как и раньше. В данном случае обертка довольно хитрая, потому что она должна сначала проверить количество и типы всех позиционных аргументов, потом вызвать исходную функцию, а потом проверить тип возвращенного значения, если он был задан. В конце возвращается результат.

Создав обертку, декоратор возвращает ее. А в самом конце возвращается сам декоратор. Таким образом, когда Python встречает в программе конструкцию `@statically_typed(str, int, str)`, он вызывает функцию `statically_typed()`. Эта функция возвращает созданную ей функцию `decorator()` – захватив все аргументы, переданные функции `statically_typed()`. Далее, вернувшись к лексеме `@`, Python выполняет возвращенную функцию `decorator()`, передавая следующую за ней функцию – либо созданную с помощью предложения `def`, либо возвращенную каким-то другим декоратором. В данном случае это функция `repeat()`, она и передается в качестве единственного аргумента функции `decorator()`. Функция `decorator()` теперь создает новую функцию `wrapper()`, параметризованную захваченным состоянием (то есть аргументами, которые были переданы функции `statically_typed()`), и возвращает обертку, которой Python затем заменяет исходную функцию `repeat()`.

Отметим, что функция `wrapper()`, созданная при вызове функции `decorator()`, созданной функцией `statically_typed()`, захватила все окружающее состояние и, в частности, кортеж `types` и ключевой аргумент `return_type`. Когда функция (или метод) таким образом захватывает состояние, говорят, что она является *замыканием*. Именно благодаря поддержке замыканий в Python и появляется возможность создавать параметризованные фабричные функции, декораторы и фабрики декораторов.

Использование декораторов для статической проверки типов аргументов и – факультативно – типа возвращаемого функцией значения может показаться полезной вещью программистам, пришедшим в Python из статически типизированных языков (например, C, C++ или Java), но расплачиваться за это приходится потерей производительности во *время выполнения* – компилируемые языки такую цену не платят. Кроме того, проверять типы в динамически типизированном языке вообще не по-питонски, но все-таки демонстрирует гибкость Python. (А если нам действительно нужна статическая типизация на этапе компиляции, то можно взять Cython, как мы увидим ниже в разделе 5.2). Пожалуй, более полезна проверка значений параметров – и этим мы займемся в следующем подразделе.

Схема написания декораторов совсем не сложна, хотя, конечно, к ней нужно привыкнуть. Если нужен непараметризованный декоратор функции или метода, то просто пишем функцию-декоратор, которая создает и возвращает обертку. Этот вариант продемонстрирован на примере декоратора `@float_args_and_return` выше и декоратора `@Web.ensure_logged_in`, который показан ниже. Если же нужен параметризованный декоратор, то пишем фабрику декораторов, которая создает декоратор (а тот, в свою очередь, создает обертку), – пример дает функция `statically_typed()`.

```
@application.post("/mailinglists/add")
@Web.ensure_logged_in
def person_add_submit(username):
   name = bottle.request.forms.get("name")
   try:
      id = Data.MailingLists.add(name)
      bottle.redirect("/mailinglists/view")
   except Data.Sql.Error as err:
      return bottle.mako_template("error", url="/mailinglists/add",
          text="Add Mailinglist", message=str(err))
```

Этот фрагмент взят из веб-приложения для управления списками рассылки, в котором используется облегченный веб-каркас `bottle` (`bottlepy.org`). Декоратор `@application.post` предоставлен каркасом и позволяет ассоциировать функцию с URL. В данном случае мы хотим, чтобы к странице `mailinglists/add` имели доступ только аутентифицированные пользователи, а всех остальных переадресуем на страницу `login`. Вместо того чтобы вставлять в каждую функцию, порождающую веб-страницу один и тот же код, проверяющий, что пользователь аутентифицирован, мы создали декоратор `@Web.ensure_logged_in`, который берет эту проверку на себя, так что код

наших функций не будет загроможден вещами, относящимися к аутентификации.

```
def ensure_logged_in(function):
   @functools.wraps(function)
   def wrapper(*args, **kwargs):
      username = bottle.request.get_cookie(COOKIE,
         secret=secret(bottle.request))
      if username is not None:
         kwargs["username"] = username
         return function(*args, **kwargs)
      bottle.redirect("/login")
   return wrapper
```

Когда пользователь заходит на сайт, код страницы `login` проверяет имя и пароль и, если все правильно, устанавливает в браузере пользователя кук, существующий в течение сеанса.

Когда пользователь запрашивает страницу, для которой ассоциированная функция защищена декоратором `@ensure_logged_in`, как, например, функция `person_add_submit()` на странице `mailinglists/add`, вызывается определенная выше функция `wrapper()`. Эта обертка первым делом извлекает имя пользователя из кука. Если кук отсутствует, значит, пользователь не аутентифицирован, поэтому мы переадресуем его на страницу `login`. Если же пользователь аутентифицирован, то мы добавляем его имя в состав ключевых аргументов и возвращаем результат вызова исходной функции. Это означает, что исходная функция может без опаски предполагать, что пользователь уже аутентифицирован и его имя известно.

2.4.2. Декораторы классов

Классы с большим числом свойств, доступных для чтения и записи, – обычное дело. Нередко в них много повторяющегося или почти повторяющегося кода в методах чтения и установки свойств. Возьмем, к примеру, класс `Book`, в котором имеются свойства, содержащие название книги, ее код ISBN, цену и количество. Нам понадобилось бы четыре практически одинаковых декоратора `@property` (в каждом был бы код вида `@property def title(self): return title`). Еще нужно было бы написать четыре метода установки, каждый со своей валидацией – хотя код валидации цены и количества отличался бы только величиной минимального и максимального значения. Если таких классов будет много, то у нас на руках окажется уйма почти одинакового кода.

К счастью, благодаря поддержке декораторов классов в Python существует возможность исключить такое дублирование. Так, выше в этой главе мы использовали декоратор класса для создания специальных классов с контролем интерфейса, избежав повторения в каждом из них десяти строк кода (раздел 2.2). А вот еще один пример – реализация класса `Book` с четырьмя проверяемыми свойствами (плюс еще одно вычисляемое свойство, допускающее только чтение):

```
@ensure("title", is_non_empty_str)
@ensure("isbn", is_valid_isbn)
@ensure("price", is_in_range(1, 10000))
@ensure("quantity", is_in_range(0, 1000000))
class Book:

   def __init__(self, title, isbn, price, quantity):
      self.title = title
      self.isbn = isbn
      self.price = price
      self.quantity = quantity

   @property
   def value(self):
      return self.price * self.quantity
```

`self.title`, `self.isbn` и т. д. – это всё свойства, поэтому присваивания, выполняемые в методе `__init__()`, проходят валидацию соответствующим методом установки. Но вместо того чтобы вручную писать код методов чтения и установки этих свойств, мы четыре раза воспользовались декоратором класса, который сделал всю работу за нас.

Функция `ensure()` принимает два параметра – имя свойства и функцию-валидатор, а возвращает декоратор класса. Этот декоратор затем применяется к следующему далее классу.

Таким образом, здесь создается практически пустой класс `Book`, затем производится первый вызов `ensure()` (с параметром `"quantity"`), после чего применяется возвращенный декоратор класса. В результате в классе `Book` появляется свойство `quantity`. Затем производится следующий вызов `ensure()` (с параметром `"price"`), применяется очередной возвращенный декоратор, и в классе `Book` появляется новое свойство – `price`. Этот процесс повторяется еще дважды, и в итоге мы получаем класс `Book` со всеми четырьмя свойствами.

На первый взгляд кажется, что действия производятся задом наперед, но на самом деле происходит вот что:

```
ensure("title", is_non_empty_str)( # псевдокод
   ensure("isbn", is_valid_isbn)(
```

```
        ensure("price", is_in_range(1, 10000))(
            ensure("quantity", is_in_range(0, 1000000))(class Book: ...))))
```

Предложение `class Book` должно быть выполнено первым, потому что получающийся в результате объект класса должен быть передан в качестве параметра первому вызову `ensure()`, а возвращенный им объект класс – предыдущему вызову и т. д.

Отметим, что для цены и количества задана одна и та же функция-валидатор, только с разными параметрами. На самом деле `is_in_range()` – фабричная функция, которая создает и возвращает новую функцию `is_in_range()`, в код которой зашиты заданные минимум и максимум.

Скоро мы увидим, что декоратор класса, возвращаемый функцией `ensure()`, добавляет в класс свойство. Метод установки этого свойства вызывает функцию-валидатор данного свойства, передавая ей два аргумента: имя свойства и его новое значение. Валидатор не должен делать ничего, если это значение допустимо, в противном случае он должен возбудить исключение (например, `ValueError`). Но прежде чем знакомиться с реализацией `ensure()`, рассмотрим еще два валидатора.

```
def is_non_empty_str(name, value):
    if not isinstance(value, str):
        raise ValueError("{} must be of type str".format(name))
    if not bool(value):
        raise ValueError("{} may not be empty".format(name))
```

Этот валидатор проверяет, что свойство `title` – непустая строка. Как видно из параметров `ValueError`, имя свойства полезно в сообщениях об ошибках.

```
def is_in_range(minimum=None, maximum=None):
    assert minimum is not None or maximum is not None
    def is_in_range(name, value):
        if not isinstance(value, numbers.Number):
            raise ValueError("{} must be a number".format(name))
        if minimum is not None and value < minimum:
            raise ValueError("{} {} is too small".format(name, value))
        if maximum is not None and value > maximum:
            raise ValueError("{} {} is too big".format(name, value))
    return is_in_range
```

Это фабричная функция, которая создает новый валидатор, проверяющий, что переданное ему значение – число (для чего используется абстрактный базовый класс `numbers.Number`), находящееся в указанном диапазоне. Созданный валидатор возвращается.

```
def ensure(name, validate, doc=None):
   def decorator(Class):
      privateName = "__" + name
      def getter(self):
         return getattr(self, privateName)
      def setter(self, value):
         validate(name, value)
         setattr(self, privateName, value)
      setattr(Class, name, property(getter, setter, doc=doc))
      return Class
   return decorator
```

Функция `ensure()` создает декоратор класса, параметризованный именем свойства, функцией-валидатором и необязательной строкой документации. Всякий раз, как декоратор, возвращенный `ensure()`, применяется к некоторому классу, этот класс дополняется новым свойством.

Функция `decorator()` получает класс в качестве своего единственного аргумента. Эта функция сначала создает закрытое имя; значение свойства будет храниться в атрибуте с таким именем. (Так, в нашем классе `Book` значение свойства `self.title` хранится в закрытом атрибуте `self.__title`.) Затем она создает метод чтения, который возвращает значение, хранящееся в атрибуте с закрытым именем. Встроенная функция `getattr()` принимает объект и имя атрибута и возвращает значение этого атрибута – или возбуждает исключение `AttributeError`. Затем функция создает метод установки, который вызывает захваченную функцию `validate()`, после чего (если `validate()` не возбудила исключение) записывает в атрибут с закрытым именем новое значение. Встроенная функция `setattr()` принимает объект, имя атрибута и значение и записывает указанное значение в указанный атрибут. Если такого атрибута раньше не было, он создается.

Созданные методы чтения и установки используются для создания нового свойства, которое добавляется в переданный класс в виде атрибута с указанным (открытым) именем с помощью встроенной функции `setattr()`. Встроенная функция `property()` принимает метод чтения и необязательные метод установки, метод удаления и строку документации, а возвращает свойство; как мы видели, ее можно использовать также в качестве декоратора метода. Затем модифицированный класс возвращается функцией `decorator()`, а сама функция `decorator()` возвращается фабричной функцией `ensure()`, порождающей декоратор класса.

2.4.2.1. Использование декоратора класса для добавления свойств

В предыдущем примере мы вынуждены были использовать декоратор класса `@ensure` для каждого атрибута, который хотели подвергнуть валидации. Некоторые программисты не любят такие нагромождения декораторов, а предпочитают комбинировать единственный декоратор класса с атрибутами в теле класса, считая, что получающийся таким образом код проще читать.

```
@do_ensure
class Book:

   title = Ensure(is_non_empty_str)
   isbn = Ensure(is_valid_isbn)
   price = Ensure(is_in_range(1, 10000))
   quantity = Ensure(is_in_range(0, 1000000))

   def __init__(self, title, isbn, price, quantity):
      self.title = title
      self.isbn = isbn
      self.price = price
      self.quantity = quantity

   @property
   def value(self):
      return self.price * self.quantity
```

В этой версии класса `Book` используется единственный декоратор класса `@do_ensure` в сочетании с экземплярами `Ensure`. Каждый экземпляр `Ensure` принимает функцию валидации, а декоратор класса `@do_ensure` заменяется все экземпляры `Ensure` прошедшим валидацию свойством с тем же именем. Сами же функции валидации (`is_non_empty_str()` и т. д.) ничем не отличаются от показанных выше.

```
class Ensure:
   def __init__(self, validate, doc=None):
      self.validate = validate
      self.doc = doc
```

Этот крохотный класс служит для хранения функции валидации (которая в конечном итоге будет использована в методе установки свойства) и необязательной строки документации. Например, атрибут `title` класса `Book` начинает свою жизнь как экземпляр `Ensure`, но после создания класса `Book` декоратор `@do_ensure` заменяет все

экземпляры Ensure свойствами. Таким образом, атрибут title становится свойством title (в методе установки которого вызывается функция валидации, хранившаяся в исходном экземпляре Ensure).

```
def do_ensure(Class):
   def make_property(name, attribute):
      privateName = "__" + name
      def getter(self):
         return getattr(self, privateName)
      def setter(self, value):
         attribute.validate(name, value)
         setattr(self, privateName, value)
      return property(getter, setter, doc=attribute.doc)
   for name, attribute in Class.__dict__.items():
      if isinstance(attribute, Ensure):
         setattr(Class, name, make_property(name, attribute))
   return Class
```

В этом декораторе класса три части. В первой части мы определяем вложенную функцию make_property(). Эта функция принимает имя (например, "title") и атрибут типа Ensure и создает и возвращает свойство, значение которого хранится в закрытом атрибуте (например, "__title"). Когда впоследствии производится обращение к методу установки этого свойства, тот вызывает функцию валидации. Во второй части мы обходим все атрибуты класса и заменяем экземпляры Ensure новыми свойствами. В третьей части возвращается модифицированный класс.

После того как декоратор закончит работу, в декорированном классе все атрибуты Ensure будут заменены валидируемым свойством с таким же именем.

Теоретически можно было бы избежать вложенной функции и просто поместить этот код после проверки if isinstance(). Однако на практике это не работает из-за проблем с поздним связыванием, поэтому наличие отдельной функции здесь обязательно. Эта неприятность часто встречается при создании декораторов или фабрик декораторов, но за счет использования отдельной – возможно, вложенной – функции ее обычно удается решить.

2.4.2.2. Использование декоратора класса вместо наследования

Иногда мы создаем базовый класс с какими-то методами или данными только для того, чтобы два или более раз унаследовать ему. Это позволяет уйти от дублирования методов и данных и хорошо масшта-

бируется на случай, когда требуется создавать дополнительные подклассы. Но если унаследованные методы или данные в подклассах не модифицируются, то того же эффекта можно достичь с помощью декоратора класса.

Например, ниже нам понадобится базовый класс `Mediated`, который предоставляет атрибут `self.mediator` и метод `on_change()` (раздел 3.5). Этому классу наследуют два класса, `Button` и `Text`, которые пользуются этим данными и методом, но не модифицируют их.

```
class Mediated:

   def __init__(self):
      self.mediator = None

   def on_change(self):
      if self.mediator is not None:
         self.mediator.on_change(self)
```

Это базовый класс, взятый из файла `mediator1.py`. Для наследования ему применяется обычный синтаксис, то есть `class Button(Mediated):…` и `class Text(Mediated):….` Но поскольку никакому классу не нужно модифицировать унаследованный метод `on_change()`, мы можем воспользоваться декоратором класса вместо создания подкласса.

```
def mediated(Class):
   setattr(Class, "mediator", None)
   def on_change(self):
      if self.mediator is not None:
         self.mediator.on_change(self)
   setattr(Class, "on_change", on_change)
   return Class
```

Этот код взят из файла `mediator1d.py`. Декоратор класса применяется как обычно, то есть `@mediated class Button: …` и `@mediated class Text: ….` Декорированные классы обладают точно таким же поведением, как подклассы.

Декораторы функций и классов – очень мощный и в то же время простой в использовании механизм Python. И, как мы видели, иногда декораторы классов позволяют заменить наследование. Создание декораторов – простая форма метапрограммирования, а декораторы классов зачастую можно использовать вместо более сложных видов метапрограммирования, например метаклассов.

2.5. Паттерн Фасад

Паттерн Фасад служит для того, чтобы предоставить упрощенный унифицированный интерфейс к подсистеме, чей истинный интерфейс слишком сложен или написан на таком низком уровне, что работать с ним неудобно.

В стандартной библиотеке Python имеются модули для работы с файлами, сжатыми программами gzip, tar+gzip и zip, но у этих модулей разные интерфейсы. Но представим, что хочется иметь простой унифицированный интерфейс для получения списка файлов в архиве и извлечения файлов. Одно из возможных решений – применить паттерн Фасад для предоставления очень простого высокоуровневого интерфейса, который делегирует всю грязную работу стандартной библиотеке.

На рис. 2.7 показано, какой интерфейс мы хотели бы предложить пользователям (свойство `filename` и методы `names()` и `unpack()`), а также интерфейсы, скрывающиеся за фасадом. В экземпляре класса `Archive` хранится имя одного архивного файла, и открывается этот файл только тогда, когда объект просят вернуть список имен файлов в архиве или распаковать архив (код, приведенный в этом разделе, взят из файла `Unpack.py`).

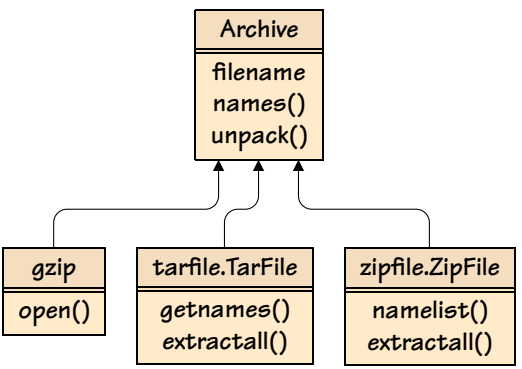

Рис. 2.7. Фасад Archive

```
class Archive:

    def __init__(self, filename):
        self._names = None
        self._unpack = None
        self._file = None
        self.filename = filename
```

Ожидается, что в переменной `self._names` хранится вызываемый объект, который возвращает список имен файлов в архиве. Аналогично в переменной `self._unpack` хранится вызываемый объект, который содержит внутри себя объект открытого файла архива. А `self.filename` – допускающее чтение и запись свойство для хранения имени архивного файла.

```
@property
def filename(self):
   return self.__filename

@filename.setter
def filename(self, name):
   self.close()
   self.__filename = name
```

Если пользователь изменяет имя файла (с помощью присваивания *archive*`.filename = newname`), то текущий архивный файл закрывается (если был открыт). Но мы не открываем сразу же новый архив, потому что в классе `Archive` применяется отложенное вычисление, то есть архив открывается не раньше, чем необходимо.

```
def close(self):
   if self._file is not None:
      self._file.close()
   self._names = self._unpack = self._file = None
```

Теоретически предполагается, что пользователь класса `Archive` вызовет метод `close()` по завершении работы с экземпляром. Этот метод закрывает файловый объект (если он открыт) и записывает в переменные `self._names`, `self._unpack` и `self._file` значение `None`, чтобы показать, что ничего полезного в них нет.

Но мы сделали класс `Archive` контекстным менеджером (как скоро станет ясно), так что на практике пользователь может и не вызывать `close()` самостоятельно при условии, что объект используется внутри предложения `with`. Например:

```
with Archive(zipFilename) as archive:
   print(archive.names())
   archive.unpack()
```

Здесь мы создаем объект `Archive` для zip-файла, печатаем список имен содержащихся в нем файлов на консоли, а затем извлекаем все файлы в текущий каталог. А поскольку `archive` – контекстный менеджер, метод `archive.close()` вызывается автоматически, когда `archive` покидает область видимости `with`.

```
   def __enter__(self):
      return self

   def __exit__(self, exc_type, exc_value, traceback):
      self.close()
```

Этих двух методов достаточно, чтобы превратить `Archive` в контекстный менеджер. Метод `__enter__()` возвращает `self` (экземпляр `Archive`), и это значение присваивается переменной, определенной в предложении `with … as`. Метод `__exit__()` закрывает объект архивного файла (если тот открыт) и, так как он (неявно) возвращает `None`, все возникшие внутри него исключения нормально распространяются.

```
   def names(self):
      if self._file is None:
         self._prepare()
      return self._names()
```

Этот метод возвращает список имен файлов в архиве, причем если архив еще не открыт, то предварительно открывает его и записывает в переменные `self._names` и `self._unpack` соответствующие вызываемые объекты (вызывая метод `self._prepare()`).

```
def unpack(self):
   if self._file is None:
      self._prepare()
   self._unpack()
```

Этот метод извлекает из архива все файлы, но, как мы увидим ниже, только если имена «безопасны».

```
def _prepare(self):
   if self.filename.endswith((".tar.gz", ".tar.bz2", ".tar.xz",
         ".zip")):
      self._prepare_tarball_or_zip()
   elif self.filename.endswith(".gz"):
      self._prepare_gzip()
   else:
      raise ValueError("unreadable: {}".format(self.filename))
```

Этот метод делегирует подготовку методам, зависящим от типа архива. Для сжатых tar-архивов и zip-файлов код очень похож, поэтому они готовятся в одном методе. Файлы же, сжатые gzip, обрабатываются по-другому, так что для них имеется отдельный метод.

В методах подготовки в переменные `self._names` и `self._unpack` необходимо записать вызываемые объекты, чтобы их можно было вызвать из методов `names()` и `unpack()` (это мы уже видели выше).

```
def _prepare_tarball_or_zip(self):
   def safe_extractall():
      unsafe = []
      for name in self.names():
         if not self.is_safe(name):
            unsafe.append(name)
      if unsafe:
         raise ValueError("unsafe to unpack: {}".format(unsafe))
      self._file.extractall()
   if self.filename.endswith(".zip"):
      self._file = zipfile.ZipFile(self.filename)
      self._names = self._file.namelist
      self._unpack = safe_extractall
   else:              # расширения .tar.gz, .tar.bz2, .tar.xz
      suffix = os.path.splitext(self.filename)[1]
      self._file = tarfile.open(self.filename, "r:" + suffix[1:])
      self._names = self._file.getnames
      self._unpack = safe_extractall
```

Этот метод сначала создает вложенную функцию `safe_extractall()`, которая проверяет все имена в архиве и возбуждает исключение `ValueError`, обнаружив хотя бы одно небезопасное имя; что значит «небезопасное», определяет метод `is_safe()`. Если все имена безопасны, то вызывается метод `tarball.TarFile.extractall()` или `zipfile.ZipFile.extractall()`.

В зависимости от расширения имени архива мы открываем файл методом `tarfile.open` или `zipfile.ZipFile` и записываем результат в `self._file`. Затем в `self._names` записывается соответствующий связанный метод (`namelist()` или `getnames()`), а в `self._unpack` – созданная выше функция `safe_extractall()`. Эта функция представляет собой замыкание, которое захватывает `self`, поэтому в ней можно обратиться к `self._file` и вызвать хранящийся в этой переменной метод `extractall()`. (См. врезку «Связанные и несвязанные методы».)

Связанные и несвязанные методы

Связанным называется метод, уже ассоциированный с экземпляром своего класса. Представьте себя класс `Form` с методом `update_ui()`. Если написать *`bound = self.update_ui`* внутри одного из методов класса `Form`, то в *`bound`* будет помещена ссылка на объект, представляющий метод `Form.update_ui()`, связанный с конкретным экземпляром формы (`self`). Связанный метод можно вызывать непосредственно, например: `bound()`.

С *несвязанным* методом не ассоциирован никакой экземпляр. Например, если написать *`unbound = Form.update_ui`*, то в переменную *`unbound`* будет помещена ссылка на объект, представляющий метод `Form.update_ui()`,

но без привязки к конкретному экземпляру. Это означает, что при вызове несвязанного метода необходимо указать подходящий экземпляр в первом аргументе, например: *form = Form(); unbound(form)*. (Строго говоря, в Python 3 нет несвязанных методов, поэтому *unbound* – на самом деле внутренний объект-функция, хотя это различие существенно лишь в некоторых закоулках метапрограммирования.)

```
def is_safe(self, filename):
   return not (filename.startswith(("/", "\\")) or
      (len(filename) > 1 and filename[1] == ":" and
       filename[0] in string.ascii_letter) or
      re.search(r"[.][.][/\\]", filename))
```

Если распаковать специально созданный вредоносный архив, то можно затереть важные системные файлы, заменив их неработающими или вредными. Поэтому никогда не следует открывать архивы, содержащие файлы с абсолютными или относительными путями, и, кроме того, архивы нужно распаковывать от имени непривилегированного пользователя (ни в коем случае не root или Administrator).

Этот метод возвращает `False`, если переданное ему имя файла начинается знаком прямой или обратной косой черты (то есть содержит абсолютный путь) или включает строку `../` либо `..\` (относительный путь, который может вести куда угодно) или начинается строкой *D*`:`, где *D* – буква диска в Windows. Иными словами, любое имя файла, содержащее абсолютную или относительную часть, считается небезопасным. Для остальных имен метод возвращает `True`.

```
def _prepare_gzip(self):
   self._file = gzip.open(self.filename)
   filename = self.filename[:-3]
   self._names = lambda: [filename]
   def extractall():
      with open(filename, "wb") as file:
         file.write(self._file.read())
   self._unpack = extractall
```

Этот метод записывает открытый файловый объект в `self._file` и подходящие вызываемые объекты в `self._names` и `self._unpack`. В функции `extractall()` мы должны самостоятельно читать и записывать данные.

Паттерн Фасад бывает очень полезен для создания упрощенных и удобных интерфейсов. Плюс в том, что нам не нужно возиться с низкоуровневыми деталями, минус – в том, что приходится отказаться от точной регулировки. Однако фасад не скрывает находящейся

за ним функциональности, поэтому можно в основном пользоваться фасадом, но переходить на нижний уровень, когда требуется доступ к деталям.

Паттерны Фасад и Адаптер на первый взгляд кажутся похожими. Но это впечатление обманчиво: разница в том, что фасад надстраивает простой интерфейс поверх сложного, а адаптер надстраивает унифицированный интерфейс над каким-то другим (необязательно сложным). Оба паттерна можно использовать совместно. Например, можно было бы определить интерфейс для работы с архивными файлами (.tar.gz, .zip, .cab-файлами в Windows и т. д.), написать адаптер для каждого формата и надстроить поверх них фасад, чтобы пользователям не нужно было беспокоиться о том, какой конкретно формат файла используется.

2.6. Паттерн Приспособленец

Паттерн Приспособленец предназначен для обработки большого числа сравнительно небольших объектов, когда многие из этих объектов являются дубликатами. Реализация паттерна предполагает, что каждый уникальный объект представляется всего один раз, и именно этот экземпляр отдается в ответ на запросы.

В Python подход к реализации приспособленцев естественный – благодаря наличию ссылок на объекты. Например, если бы у нас был длинный список строк, в котором много дубликатов, то, сохраняя ссылки на объекты (то есть переменные), а не литеральные строки, мы могли бы существенно сэкономить память.

```
red, green, blue = "red", "green", "blue"
x = (red, green, blue, red, green, blue, red, green)
y = ("red", "green", "blue", "red", "green", "blue", "red", "green")
```

Здесь кортеж `x` содержит 8 ссылок на объекты, которые ссылаются всего на 3 строки. А в кортеже `y` хранятся 8 ссылок на 8 строк, так как то, что мы написали, – по существу, не более чем синтаксическая глазурь, за которой скрывается конструкция `_anonymous_item0 = "red", …, _anonymous_item7 = "green"; y = (_anonymous_item0, … _anonymous_item7)`.

Пожалуй, самый простой способ воспользоваться паттерном Приспособленец в Python – завести словарь, в котором каждый уникальный объект хранится в виде значения, идентифицируемого уникальным ключом. Например, если мы создаем кучу HTML-страниц, в которых

шрифты задаются в таблице CSS-стилей, а не указываются всякий раз явно, то можно было подготовить нужные шрифты заранее (или по мере необходимости) и поместить их в словарь. Тогда всякий раз, как нам потребуется шрифт, мы сможем взять его из словаря. Таким образом, каждый уникальный шрифт создается только один раз, сколько бы раз он ни использовался.

Иногда имеется много необязательно мелких объектов, причем все они или большинство из них уникальны. В таких случаях есть простой способ уменьшить потребление памяти – воспользоваться переменной `__slots__`.

```
class Point:

    __slots__ = ("x", "y", "z", "color")

    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

Это простой класс `Point`, в котором хранятся три координаты и цвет точки. Благодаря `__slots__` ни у одного объекта `Point` нет собственного закрытого словаря (`self.__dict__`). Однако это одновременно означает, что состав атрибутов для отдельных точек нельзя расширить. (Код этого класса находится в файле `pointstore1.py`.)

На тестовой машине создание кортежа, содержащего миллион таких точек, заняло примерно 2,5 секунды, и программа (которая больше почти ничего не делала) потребила 183 МиБ памяти. Без слотов та же программа работала на долю секунды быстрее, но потребила 312 МиБ памяти.

По умолчанию Python всегда жертвует памятью ради скорости, но мы можем поступить наоборот, если сочтем, что так выгоднее.

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

Это начало второй версии класса `Point` (ее код находится в файле `pointstore2.py`). В ней используется база данных DBM (хранилище ключей и значений), которая находится в файле на диске. Ссылка на объект DBM хранится в статической (уровня класса) переменной `Point.__dbm`. Все объекты `Point` разделяют один и тот же DBM-

файл. Сначала мы открываем файл DBM, подготавливая его для работы. По умолчанию модуль `shelve` автоматически создает DBM-файл, если он еще не существует (ниже мы увидим, как гарантировать корректное закрытие DBM-файла).

Модуль `shelve` сериализует значения при сохранении и десериализует при выборке. (Формат сериализации – pickle – в Python принципиально небезопасен, потому что в процессе десериализации по существу выполняется произвольный Python-код. Поэтому никогда не следует использовать данные в этом формате, полученные из ненадежных источников, а также хранить в нем данные, к которым возможен доступ со стороны ненадежных программ. Или, если все же необходимо использовать формат pickle в подобных обстоятельствах, следует предусматривать собственные меры защиты, например, контрольные суммы и шифрование.)

```
def __init__(self, x=0, y=0, z=0, color=None):
    self.x = x
    self.y = y
    self.z = z
    self.color = color
```

Этот метод делает в точности то же самое, что метод из файла `pointstore1.py`, но под капотом присваиваемые значения записываются в DBM-файл.

```
def __key(self, name):
    return "{:X}:{}".format(id(self), name)
```

Этот метод создает строковый ключ для любого из атрибутов `x`, `y`, `z` и `color` объекта `Point`. Ключ состоит из идентификатора экземпляра (уникального числа, которое возвращает встроенная функция `id()`) в шестнадцатеричном виде и имени атрибута. Например, если идентификатор точки равен 3 954 827, то ее атрибут `x` будет храниться с ключом `"3C588B:x"`, атрибут `y` – с ключом `"3C588B:y"` и т. д.

```
def __getattr__(self, name):
    return Point.__dbm[self.__key(name)]
```

Этот метод вызывается при обращении к атрибуту объекта `Point` (например, `x = point.x`).

Ключи и значения в базе данных DBM должны быть последовательностями байтов (`bytes`). По счастью, модули Python для работы с DBM принимают ключи как в виде `bytes`, так и в виде `str`, самостоятельно преобразуя последние в `bytes` с применением кодировки по

умолчанию (UTF-8). А если мы пользуемся модулем `shelve` (как в данном случае), то можем сохранять любые сериализуемые значения, поскольку `shelve` преобразует их в `bytes` и обратно.

Итак, мы можем получить нужный ключ и извлечь соответствующее ему значение. А благодаря модулю `shelve` прочитанное значение преобразуется из сериализованной последовательности байтов в объект исходного типа (например, в `int` или `None` в случае цвета точки).

```
def __setattr__(self, name, value):
    Point.__dbm[self.__key(name)] = value
```

Этот метод вызывается при установке любого атрибута `Point` (например, `point.y = y`). Здесь мы получаем ключ и устанавливаем соответствующее ему значение, а модуль `shelve` автоматически преобразует это значение в сериализованную последовательность байтов.

```
atexit.register(__dbm.close)
```

В конце класса `Point` мы регистрируем метод DBM `close()`, так чтобы он вызывался при завершении программы. Для этого используется функция `register()` из модуля `atexit`.

На тестовой машине создание базы данных с миллионом точек заняло примерно минуту, и программа потребила 29 МиБ памяти (и создала файл размером 361 МиБ на диске). Сравните с 183 МиБ, понадобившихся первой версии. Конечно, на заполнение DBM-файла уходит заметное время, но, коль скоро это сделано, поиск в нем производится очень быстро, поскольку в большинстве операционных систем часто используемые файлы кэшируются в памяти.

2.7. Паттерн Заместитель

Паттерн Заместитель применяется, когда мы хотим подставить один объект вместо другого. В книге «Паттерны проектирования» описаны четыре таких случая. Первый – удаленный прокси, когда доступ к удаленному объекту проксируется локальным объектом. Идеальный пример такого рода дает библиотека RPyC: она позволяет создавать объекты на сервере и их заместители на одном или нескольких клиентах (мы познакомимся с этой библиотекой в главе 6, раздел 6.2). Второй случай – виртуальный прокси, который позволяет создавать облегченные объекты вместо тяжеловесных, а последние создавать только тогда, когда необходимо. Пример мы рассмотрим в этом раз-

деле. Третий случай – защитный прокси, который предоставляет различные уровни доступа в зависимости от прав клиента. И четвертый – интеллектуальная ссылка, которая «выполняет дополнительные действия при обращении к объекту». Для кодирования всех вариантов заместителей применим общий подход, хотя поведение, описанное в четвертом случае, можно было бы получить и с помощью дескриптора (например, заменить объект свойством, применив декоратор `@property`)[3].

Этот паттерн можно использовать и для автономного тестирования. Если, например, нам нужно протестировать код, обращающийся к ресурсу, который не всегда бывает доступен, или класс, который еще не до конца написан, то можно было бы создать заместитель ресурса или класса, у которого точно такой же интерфейс, но нереализованная функциональность заменена заглушками. Такой подход настолько полезен, что в состав Python 3.3 даже включена библиотека `unittest.mock` для создания подставных объектов и добавления заглушек вместо отсутствующих методов (см. `docs.python.org/py3k/library/unittest.mock.html`).

В этом разделе мы предположим, что нужно создать несколько изображений на пробу, из которых в конце останется только одно. В нашем распоряжении есть модуль `Image` и почти эквивалентный ему, но более быстрый модуль `cyImage` (рассматривается в разделе 3.12 и 5.2.2), однако эти модули создают изображения в памяти. Поскольку нам понадобится только одно из гипотетических изображений, то лучше было бы создать облегченные заместители, а когда поймем, какое изображение нас устраивает, тогда и создать настоящее.

Интерфейс класса `Image.Image` включает десять методов, помимо конструктора: `load()`, `save()`, `pixel()`, `set_pixel()`, `line()`, `rectangle()`, `ellipse()`, `size()`, `subsample()` и `scale()` (сюда не входят некоторые статические вспомогательные методы, доступные также в виде функций модуля, например, `Image.Image.color_for_name()` и `Image.color_for_name()`).

В классе заместителя мы реализуем только часть методов `Image.Image`, достаточную для наших целей. Сначала посмотрим, как используется заместитель. Приведенный ниже код взят из файла `imageproxy1.py`, а нарисованное им изображение показано на рис. 2.8.

---

[3] Дескрипторы рассматриваются в книге «Программирование на Python 3. Второе издание» (см. проложение B) и в онлайновой документации: `docs.python.org/3/reference/datamodel.html#descriptors`.

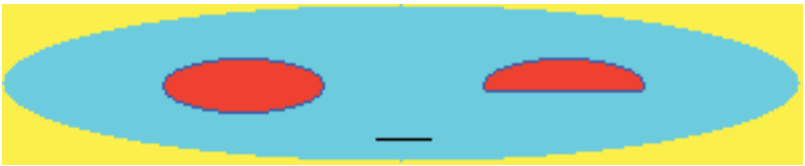

Рис. 2.8. Нарисованное изображение

```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
   for color in ("yellow", "cyan", "blue", "red", "black"))
```

Прежде всего, создадим константы для обозначения цветов, воспользовавшись функцией `color_for_name()` из модуля `Image`.

```
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```

Здесь мы создаем прокси-изображение, передавая конструктору класс изображения, который хотим использовать. Затем мы на нем рисуем и в конце сохраняем результат. Этот код работал бы с тем же успехом, если бы мы создавали изображение с помощью `Image.Image()`, а не `ImageProxy()`. Однако при работе с заместителем изображение не создается, пока не будет вызван метод `save()`, поэтому плата за создание изображения до сохранения крайне мала (как в плане потребления памяти, так и с точки зрения скорости обработки) и, выбросив изображение без сохранения, мы потеряем совсем немного. Сравните с `Image.Image` – тут нам пришлось бы заплатить высокую цену авансом (то есть создать массив цветов размером `width` × `height`) и выполнить дорогостоящую обработку при рисовании (установить каждый пиксель в заполненном прямоугольнике, да еще вычислить цвет этого пикселя), даже если в итоге мы это изображение выбросим.

```
class ImageProxy:

   def __init__(self, ImageClass, width=None, height=None, filename=None):
      assert (width is not None and height is not None) or \
              filename is not None
```

```
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
        else:
            self.commands = [(self.Image, width, height)]

    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```

Класс `ImageProxy` можно подставить вместо `Image.Image` (или любого другого класса изображения, который поддерживает интерфейс класса `Image`), при условии, что неполного интерфейса, предоставляемого заместителем, достаточно. Класс `ImageProxy` не хранит изображение, он просто сохраняет список кортежей команд, где первым элементом каждого кортежа является функция или несвязанный метод, а остальные – передаваемые этой функции или методу аргументы.

При создании объекта `ImageProxy` необходимо указать ширину и высоту (чтобы создать изображение требуемого размера) или имя файла. Если задано имя файла, то сохраняются те же команды, что при вызове метода `ImageProxy.load()`: конструктор `Image.Image()` и его аргументы – `None`, `None` вместо ширины и высоты и имя файла. Отметим, что если впоследствии метод `ImageProxy.load()` будет вызван еще раз, то все прежние команды отбрасываются, команда загрузки становится первым и единственным элементом `self.commands`. Если же заданы ширина и высота, то запоминается конструктор `Image.Image()` и ширина и высота в качестве его аргументов.

При вызове любого неподдерживаемого метода (например, `pixel()`) Python не найдет его и сделает именно то, что нам нужно: возбудит исключение `AttributeError`. Альтернативный подход к обработке методов, отсутствующих у заместителя, – создать фактическое изображение при вызове любого из них и, начиная с этого момента, работать с этим изображением (такой подход принят в программе из файла `imageproxy2.py`, которая здесь не показана).

```
    def set_pixel(self, x, y, color):
        self.commands.append((self.Image.set_pixel, x, y, color))

    def line(self, x0, y0, x1, y1, color):
        self.commands.append((self.Image.line, x0, y0, x1, y1, color))

    def rectangle(self, x0, y0, x1, y1, outline=None, fill=None):
```

```
        self.commands.append((self.Image.rectangle, x0, y0, x1, y1,
                outline, fill))

    def ellipse(self, x0, y0, x1, y1, outline=None, fill=None):
        self.commands.append((self.Image.ellipse, x0, y0, x1, y1,
                outline, fill))
```

Интерфейс рисования класса `Image.Image` состоит из четырех методов: `line()`, `rectangle()`, `ellipse()` и `set_pixel()`. Наш класс `ImageProxy` полностью поддерживает этот интерфейс, только вместо выполнения команд он добавляет их – вместе с аргументами – в список `self.commands`.

```
def save(self, filename=None):
    command = self.commands.pop(0)
    function, *args = command
    image = function(*args)
    for command in self.commands:
        function, *args = command
        function(image, *args)
    image.save(filename)
    return image
```

И лишь когда мы вызываем метод `save()`, создается настоящее изображение, и мы расплачиваемся за расходы на обработку и потребление памяти. Класс `ImageProxy` устроен так, что первой всегда идет команда создания нового изображения (пустого с заданной шириной и высотой или загружаемого из файла). Поэтому первую команду мы обрабатываем особо – сохраняем возвращенное ей значение, которое, как нам точно известно, есть объект класса `Image.Image` (или `cyImage.Image`). Затем мы обходим остальные команды, вызывая каждую по очереди и передавая ей `image` в качестве первого аргумента (`self`), т. к. команды на самом деле представляют собой вызовы несвязанных методов. В самом конце мы сохраняем изображение, обращаясь к методу `Image.Image.save()`.

Метод `Image.Image.save()` ничего не возвращает (хотя может возбуждать исключение в случае ошибки). Однако в классе `ImageProxy` мы немного модифицировали его интерфейс, так что этот метод возвращает только что созданный объект `Image.Image` на случай, если его понадобится дополнительно обработать. Это безобидное изменение, потому что если возвращенное значение игнорируется (как если бы мы вызывали `Image.Image.save()`), то оно просто молча отбрасывается. В программе `imageproxy2.py` эта модификация не нужна, так как в ней имеется свойство `image` типа `Image.Image`, при обраще-

нии к которому изображение гарантированно создается, если не было создано ранее.

Все структурные паттерны проектирования можно реализовать на языке Python. Паттерны Адаптер и Фасад упрощают повторное использование классов в новых контекстах, а паттерн Мост дает возможность внедрить сложную функциональность одного класса в другой. Паттерн Компоновщик упрощает создание иерархий объектов – хотя в Python это не особенно нужно, т. к. для этой цели часто хватает словарей. Паттерн Декоратор настолько полезен, что в Python для него имеется прямая поддержка, причем идея декоратора распространена даже на классы. Использование ссылок на объекты в Python означает, что в сам язык встроена вариация на тему паттерна Приспособленец. А паттерн Заместитель в Python реализовать особенно просто. Паттерны проектирования не ограничиваются одним лишь созданием простых и сложных объектов, а затрагивают также и поведение: каким образом отдельные объекты или их группы могут решить ту или иную задачу. В следующей главе мы как раз и займемся поведенческими паттернами.

ГЛАВА 3. Поведенческие паттерны проектирования в Python

Предмет поведенческих паттернов – способ решения задачи, то есть они имеют дело с алгоритмами и взаимодействиями объектов. Эти паттерны предлагают действенные способы обдумывания и организации вычислений. Для некоторых из них, как и для части рассмотренных выше паттернов, имеется прямая поддержка в синтаксисе Python.

Хорошо известный девиз языка Perl – «есть более одного способа решить задачу». Напротив, в «кладезе мудрости Python» (Zen of Python), составленном Тимом Петерсом, говорится «должен быть один – и в идеале только один – очевидный способ решить задачу»[1]. Но, как и в любом языке программирования, иногда способов два или больше, особенно после появления в Python списковых включений (можно использовать либо включение, либо цикл `for`) и генераторов (использовать выражения-генераторы или функцию с предложением `yield`). И, как мы увидим в этой главе, поддержка сопрограмм в Python привносит новый способ решения определенных задач.

3.1. Паттерн Цепочка ответственности

Паттерн Цепочка ответственности предназначен, для того чтобы разорвать связь между отправителем запроса и получателем, который этот запрос обрабатывает. Вместо непосредственного вызова одной функции из другой первая функция отправляет запрос цепочке получателей. Первый получатель в цепочке может либо обработать запрос и прервать цепочку (не передавать запрос дальше), либо передать за-

[1] Чтобы познакомиться с кладезем мудрости, введите в интерактивной оболочке Python команду `import this`.

прос следующему получателю. У второго получателя есть такой же выбор и так далее, пока запрос не дойдет до последнего получателя (который может отбросить запрос или возбудить исключение).

Представьте себе пользовательский интерфейс, который получает события для обработки. Одни события поступают от пользователя (например, события мыши и клавиатуры), другие – от системы (например, события таймера). В двух следующих подразделах мы рассмотрим традиционный подход к созданию цепочки обработчиков события и другой подход – на основе конвейера с использованием сопрограмм.

3.1.1. Традиционная Цепочка

В этом подразделе мы рассмотрим традиционную цепочку обработчиков события, когда каждому событию соответствует отдельный класс обработчика.

```
handler1 = TimerHandler(KeyHandler(MouseHandler(NullHandler())))
```

Вот так можно построить цепочку из четырех отдельных обработчиков. Она показана на рис. 3.1. Поскольку необработанные события мы отбрасываем, можно было бы просто передать `MouseHandler` аргумент `None` – или вообще ничего не передавать.

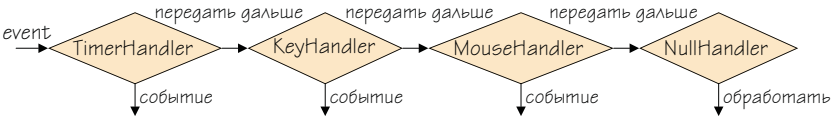

Рис. 3.1. Цепочка обработчиков событий

Порядок создания обработчиков не должен играть роли, потому что каждый обрабатывает только те события, для которых предназначен.

```
while True:
   event = Event.next()
   if event.kind == Event.TERMINATE:
      break
   handler1.handle(event)
```

Обычно события обрабатываются в цикле. Здесь мы выходим из цикла и завершаем приложение, если получено событие `TERMINATE`; во всех остальных случаях событие передается по цепочке обработчиков.

```
handler2 = DebugHandler(handler1)
```

Здесь мы создали новый обработчик (хотя могли бы с тем же успехом присвоить результат той же переменной `handler1`). Этот обработчик *обязан* быть первым в цепочке, потому что в его задачу входит прослушивание всех поступающих в нее событий и уведомление о них, однако он их никак не обрабатывает (то есть передает каждое полученное им событие дальше).

Если теперь вызвать `handler2.handle(event)` в цикле, то в дополнение к обычной обработке мы будем видеть отладочную информацию обо всех полученных событиях.

```
class NullHandler:

   def __init__(self, successor=None):
      self.__successor = successor

   def handle(self, event):
      if self.__successor is not None:
         self.__successor.handle(event)
```

Это базовый класс для всех обработчиков событий, который определяет инфраструктуру обработки. Если при создании экземпляра был указан следующий обработчик (`successor` – преемник), то, получив событие, этот экземпляр просто передаст его дальше по цепочке преемнику. Если же преемника нет, то событие будет просто отброшено. Это стандартный подход при программировании графического интерфейса пользователя (ГИП), хотя можно было бы вместо этого помещать информацию о необработанных событиях в журнал или возбуждать в этом случае исключение (например, если бы мы писали сервер).

```
class MouseHandler(NullHandler):

   def handle(self, event):
      if event.kind == Event.MOUSE:
         print("Click: {}".format(event))
      else:
         super().handle(event)
```

Поскольку мы не стали переопределять метод `__init__()`, будет вызван метод базового класса, который честно создаст переменную `self.__successor`.

Этот класс обрабатывает только интересующие его события (типа `Event.MOUSE`), а все остальные передает своему преемнику (если таковой существует).

Классы KeyHandler и TimerHandler (не показаны) устроены точно так же, как MouseHandler. Различаются они только типами обрабатываемых событий (Event.KEYPRESS и Event.TIMER) и тем, как именно они их обрабатывают (печатают разные сообщения).

```
class DebugHandler(NullHandler):

   def __init__(self, successor=None, file=sys.stdout):
      super().__init__(successor)
      self.__file = file

   def handle(self, event):
      self.__file.write("*DEBUG*: {}\n".format(event))
      super().handle(event)
```

Класс DebugHandler отличается от остальных обработчиков тем, что не обрабатывает вообще ни одно событие, и должен быть первым в цепочке. Он принимает файл или файлоподобный объект, в который выводит сведения. Получив событие, он записывает данные о нем в этот объект и передает событие дальше.

3.1.2. Цепочка на основе сопрограмм

Генератор – это функция или метод, в котором есть хотя бы одно выражение yield вместо return. Когда поток управления доходит до yield, порождается *уступаемое* значение, и функция (или метод) приостанавливается с сохранением всего состояния. В этот момент функция уступает процессор (получателю порожденного значения), поэтому, хотя функция и приостановлена, она не блокирует выполнение программы. При следующем обращении к той же функции (или методу) выполнение продолжится с предложения, следующего за yield. Таким образом, значения *вытягиваются* из генератора при итерировании (например, в цикле for *value* in generator:) или при вызове функции next(), которой передается генератор.

В сопрограмме используется то же выражение yield, что и в генераторе, но с другим поведением. Сопрограмма исполняет бесконечный цикл и приостанавливается, дойдя до первого (или единственного выражения) yield. В этот момент она ждет, когда ему будет послано значение. Когда такое произойдет (если вообще произойдет), сопрограмма получит его в виде значения выражения yield. Затем сопрограмма может делать, что ей угодно, а когда закончит, вернется в начало цикла и снова окажется приостановленной в ожидании значения для следующего выражения yield. Таким образом, значения

проталкиваются в сопрограмму путем вызова ее методов `send()` или `throw()`.

В Python любая функция или метод, содержащий ключевое слово `yield`, является генератором. Однако, воспользовавшись декоратором `@coroutine` и организовав бесконечный цикл, мы можем превратить генератор в сопрограмму (декораторы вообще и декоратор `@functools.wraps` в частности мы обсуждали в разделе 2.4 предыдущей главы).

```
def coroutine(function):
   @functools.wraps(function)
   def wrapper(*args, **kwargs):
      generator = function(*args, **kwargs)
      next(generator)
      return generator
   return wrapper
```

Обертка `wrapper` вызывает генераторную функцию только один раз и захватывает порожденный ей генератор, запоминая его в переменной `generator`. Этот генератор на самом деле представляет собой исходную функцию вместе с ее аргументами и локальными переменными, захваченными как состояние. Затем обертка продвигает генератор вперед – один раз, используя встроенную функцию `next()`, – чтобы он дошел до первого выражения `yield`. После этого генератор – вместе с захваченным состоянием – возвращается. Этот возвращенный генератор и есть сопрограмма, готовая принять значение для своего первого (или единственного) выражения `yield`.

Если мы вызовем генератор, то он возобновит выполнение с того места, где остановился (то есть с предложения, следующего за последним – или единственным – ранее выполненным выражением `yield`). Если же мы пошлем значение сопрограмме (воспользовавшись записью *`generator`*`.send(value)`), то оно будет получено сопрограммой как результат, возвращенный текущим выражением `yield`, и выполнение возобновится с этой точки.

Поскольку мы можем как получать значения от сопрограммы, так и посылать ей значения, то сопрограммы могут выступать в роли конвейера, в том числе цепочки обработчиков событий. Более того, нам не нужна инфраструктура преемников, т. к. вместо нее можно использовать синтаксис генераторов.

```
pipeline = key_handler(mouse_handler(timer_handler()))
```

Здесь мы создаем цепочку (pipeline) с помощью серии вложенных вызовов функций. Каждая вызываемая функция является сопрограммой, и каждая выполняется до первого (или единственного) выражения yield, после чего приостанавливается, ожидая, пока ее снова вызовут или пошлют ей значение. Таким образом, конвейер создается сразу, без блокировки.

Вместо того чтобы заводить Null-обработчик, мы можем ничего не передавать последнему обработчику в цепочке. Как это работает, мы увидим при рассмотрении типичного обработчика-сопрограммы (key_handler()).

```
while True:
   event = Event.next()
   if event.kind == Event.TERMINATE:
      break
   pipeline.send(event)
```

Как и в случае традиционного подхода, подготовив цепочку обработчиков событий, мы приступаем к их обработке в цикле. Поскольку каждая функция-обработчик – сопрограмма (генераторная функция), у нее есть метод send(). Поэтому, получив очередное событие, мы посылаем его конвейеру. В данном случае событие сначала будет послано сопрограмме key_handler(), которая либо обработает его, либо передаст дальше. Как и раньше, порядок обработчиков не играет роли.

```
pipeline = debug_handler(pipeline)
```

А это тот случай, когда порядок обработчиков имеет значение. Поскольку задача сопрограммы debug_handler() – наблюдать за всеми событиями и передавать их дальше, она должна быть первым обработчиком в цепочке. Подготовив новый конвейер, мы снова запускаем цикл обработки событий, посылая их по очереди в конвейер методом pipeline.send(event).

```
@coroutine
def key_handler(successor=None):
   while True:
      event = (yield)
      if event.kind == Event.KEYPRESS:
         print("Press: {}".format(event))
      elif successor is not None:
         successor.send(event)
```

Эта сопрограмма принимает сопрограмму-преемник, которой передавать событие, (или None) и запускает бесконечный цикл. Благо-

даря декоратору `@coroutine` `key_handler()` выполняется до выражения `yield`, поэтому при создании цепочки `pipeline` эта функция уже дошла до своего выражения `yield` и остановилась в ожидании, пока `yield` отдаст – посланное – значение (разумеется, заблокирована только эта сопрограмма, а не вся программа в целом).

Когда этой сопрограмме будет послано событие – напрямую или из другой сопрограммы в цепочке, – оно будет получено в виде значения переменной `event`. Если сопрограмма может обработать данное событие (то есть оно имеет тип `Event.KEYPRESS`), то обрабатывает – в данном случае просто печатает – и дальше не посылает. Если же сопрограмма о полученном событии ничего не знает и при условии, что имеется сопрограмма-преемник, то событие посылается преемнику. Если же преемника нет, то событие отбрасывается.

Обработав, переслав или отбросив событие, сопрограмма возвращается в начало цикла `while` и снова начинает ждать, когда `yield` отдаст значение, посланное в конвейер.

Сопрограммы `mouse_handler()` и `timer_handler()` (не показаны) устроены точно так же, как `key_handler()`; разница лишь в типе обрабатываемого события и печатаемом сообщении.

```
@coroutine
def debug_handler(successor, file=sys.stdout):
    while True:
        event = (yield)
        file.write("*DEBUG*: {}\n".format(event))
        successor.send(event)
```

Сопрограмма `debug_handler()` ждет события, печатает сведения о нем и пересылает его следующей сопрограмме для обработки.

Хотя в основе сопрограмм и генераторов лежит один и тот же механизм, работают они совершенно по-разному. В случае генератора мы *вытягиваем* значения по одному (например, `for x in range(10):`). В случае же сопрограмм мы *проталкиваем* значения по одному с помощью функции `send()`. Благодаря такой гибкости Python может выразить разнообразные алгоритмы элегантно и естественно. Так, для реализации цепочки на основе сопрограмм понадобилось гораздо меньше кода, чем в традиционной цепочке из предыдущего подраздела.

Мы снова встретимся с сопрограммами при рассмотрении паттерна Посредник в разделе 3.5.

Паттерн Цепочка ответственности, конечно, находит применение и во многих других контекстах. Например, его можно было бы использовать для обработки запросов серверу.

3.2. Паттерн Команда

Паттерн Команда применяется для инкапсуляции команд в виде объектов. Например, с его помощью можно построить последовательность команд для отложенного выполнения или создать команды, допускающие отмену. С основами этого паттерна мы уже встречались в примере `ImageProxy` (раздел 2.7), а в этом разделе пойдем дальше и напишем классы для отдельных команд, допускающих отмену, и для допускающих отмену макросов (то есть последовательностей допускающих отмену команд).

Начнем с кода, в котором паттерн Команда используется, а затем рассмотрим, как выглядят использованные в нем классы (`UndoableGrid` и `Grid`) и модуль `Command`, предоставляющий инфраструктуру выполнения-отмены и макросов.

```
grid = UndoableGrid(8, 3)      # (1) Пустая
redLeft = grid.create_cell_command(2, 1, "red")
redRight = grid.create_cell_command(5, 0, "red")
redLeft()                      # (2) вставить красные ячейки
redRight.do()                  # ИЛИ: redRight()
greenLeft = grid.create_cell_command(2, 1, "lightgreen")
greenLeft()                    # (3) вставить зеленую ячейку
rectangleLeft = grid.create_rectangle_macro(1, 1, 2, 2, "lightblue")
rectangleRight = grid.create_rectangle_macro(5, 0, 6, 1, "lightblue")
rectangleLeft()                # (4) вставить синие ячейки
rectangleRight.do()            # ИЛИ: rectangleRight()
rectangleLeft.undo()           # (5) отменить левую синюю ячейку
greenLeft.undo()               # (6) отменить левую зеленую ячейку
rectangleRight.undo()          # (7) отменить правую синюю ячейку
redLeft.undo()                 # (8) отменить красные ячейки
redRight.undo()
```

На рис. 3.2 показано, как выглядит сетка, представленная HTML-таблицей, в восемь моментов времени. В первый момент сетка только что создана и пуста. Затем показаны ее состояния после создания и вызова команд и макросов (напрямую или с помощью метода `do()`) и после каждого вызова `undo()`.

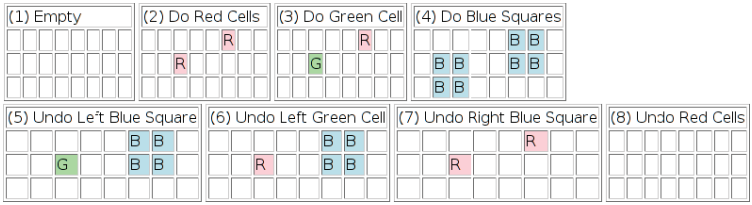

Рис. 3.2. Заполнение и отмена заполнения решетки

```
class Grid:

   def __init__(self, width, height):
      self.__cells = [["white" for _ in range(height)]
                      for _ in range(width)]

   def cell(self, x, y, color=None):
      if color is None:
         return self.__cells[x][y]
      self.__cells[x][y] = color

   @property
   def rows(self):
      return len(self.__cells[0])

   @property
   def columns(self):
      return len(self.__cells)
```

Простой класс `Grid` напоминает класс изображения тем, что хранит список списков имен цветов.

Метод `cell()` служит одновременно для чтения цвета (если аргумент `color` равен `None`) и для его установки (если цвет `color` задан). `rows` и `columns` – допускающие только чтение свойства, возвращающие размеры сетки.

```
class UndoableGrid(Grid):

   def create_cell_command(self, x, y, color):
      def undo():
         self.cell(x, y, undo.color)
      def do():
         undo.color = self.cell(x, y) # Subtle!
         self.cell(x, y, color)
      return Command.Command(do, undo, "Cell")
```

Чтобы класс `Grid` поддерживал допускающие отмену команды, мы создали подкласс с двумя дополнительными методами. Выше показан первый из них.

Любая команда должна иметь тип `Command.Command` или `Command.Macro`. Команды первого типа принимают вызываемые объекты, реализующие выполнение и отмену, и необязательное описание. Команды второго типа принимают необязательное описание, и в них можно добавить произвольное количество команд типа `Command.Command`.

Метод `create_cell_command()` принимает позицию и цвет закрашиваемой ячейки, после чего порождает обе функции, необходимые

для создания объекта `Command.Command`. Та и другая просто устанавливают цвет указанной ячейки.

Разумеется, в момент создания функций `do()` и `undo()` мы еще не знаем, каким будет цвет ячейки *непосредственно перед* выполнением команды `do()`, поэтому не знаем, и в какой цвет ее перекрасить в случае отмены. Чтобы решить эту проблему, мы читаем цвет ячейки внутри функции `do()` – в момент ее вызова – и запоминаем его в виде атрибута функции `undo()`. И только после этого устанавливаем новый цвет. Это работает, потому что функция `do()` – замыкание, которое захватывает в качестве своего состояния не только параметры `x`, `y` и `color`, но и только что созданную функцию `undo()`.

Имея функции `do()` и `undo()`, мы создаем объект `Command.Command`, который включает их и простое описание, и возвращаем эту команду вызывающей программе.

```
def create_rectangle_macro(self, x0, y0, x1, y1, color):
   macro = Command.Macro("Rectangle")
   for x in range(x0, x1 + 1):
      for y in range(y0, y1 + 1):
         macro.add(self.create_cell_command(x, y, color))
   return macro
```

Это второй метода класса `UndoableGrid` для создания команд, допускающих выполнение и отмену. Он создает макрос, охватывающий прямоугольник с заданными координатами. Для каждой подлежащей перекрашиванию ячейки создается команда с помощью уже рассмотренного выше метода `create_cell_command()`, и эта команда затем добавляется в макрос. Добавив все команды, мы возвращаем макрос.

Как мы увидим ниже, и команды, и макросы поддерживают методы `do()` и `undo()`. Коль скоро команды и макросы поддерживают одинаковые методы, а макрос содержит команды, отношения между ними описываются паттерном Составитель (раздел 2.3).

```
class Command:

   def __init__(self, do, undo, description=""):
      assert callable(do) and callable(undo)
      self.do = do
      self.undo = undo
      self.description = description

   def __call__(self):
      self.do()
```

Объект `Command.Command` ожидает два вызываемых объекта: первый – команда «do» (выполнить), второй – команда «undo» (отменить) (функция `callable()` встроена в Python 3.3; в предыдущих версиях эквивалентную ей функцию можно написать следующим образом: `def callable(function): return isinstance(function, collections.Callable)`).

Чтобы выполнить объект `Command.Command`, достаточно просто вызвать его (поскольку мы реализовали специальный метод `__call__()`) или – что эквивалентно – вызвать его метод `do()`. Для отмены команды нужно вызвать ее метод `undo()`.

```
class Macro:

   def __init__(self, description=""):
      self.description = description
      self.__commands = []

   def add(self, command):
      if not isinstance(command, Command):
         raise TypeError("Expected object of type Command, got {}".
                  format(type(command).__name__))
      self.__commands.append(command)

   def __call__(self):
      for command in self.__commands:
         command()

   do = __call__

   def undo(self):
      for command in reversed(self.__commands):
         command.undo()
```

Класс `Command.Macro` служит для инкапсуляции последовательности команд, которые должны быть выполнены – или отменены – одной операцией[2]. Класс `Command.Macro` обладает тем же интерфейсом, что `Command.Command`: методы `do()` и `undo()` и возможность вызова напрямую. Кроме того, у макросов есть метод `add()`, позволяющий добавлять объекты `Command.Command`.

В случае макросов команды следует отменять в порядке, обратном добавлению. Пусть, например, мы создали макрос и добавили в него команды `A`, `B`, `C`. При выполнении макроса (то есть при вызове

[2] Хотя мы и говорим о единственной операции выполнения макроса, эта операция не атомарна в смысле параллельного выполнения, но ее можно сделать таковой, воспользовавшись подходящими блокировками.

его напрямую или вызове его метода `do()`) сначала будет выполнена команда `A`, затем `B`, затем `C`. Поэтому при вызове `undo()` мы должны сначала выполнить метод `undo()` команды `C`, затем `B` и наконец `A`.

В Python функции, связанные методы и другие вызываемые объекты – это полноправные объекты, которые можно передавать из одного места программы в другое и сохранять в структурах данных, например, списках и словарях. Поэтому Python – идеальный язык для реализации паттерна Команда. Да и сам паттерн, как мы видели, можно использовать с большой пользой – как для предоставления функциональности выполнить-отменить, так и для поддержки макросов и отложенного выполнения.

3.3. Паттерн Интерпретатор

Паттерн Интерпретатор формализует два типичных требования: предоставить средства, с помощью которых пользователь мог бы вводить в программу нестроковые значения, и дать пользователям возможность создавать программы.

На самом примитивном уровне приложение получает от пользователя – или от другой программы – строки, которые необходимо каким-то образом интерпретировать (и, быть может, выполнить). Допустим, к примеру, что мы получаем от пользователя строку, которая должна представлять целое число. Простой – и не самый разумный – способ получить это число таков: `i = eval(userCount)`. Это опасно, потому что, понадеявшись получить невинную строку вида `"1234"`, мы на самом деле можем столкнуться с таким монстром: `"os.system('rmdir /s /q C:\\\\')"`.

В общем случае, имея строку, которая должна представлять значение определенного типа, мы можем в Python получить это значение просто и безопасно.

```
try:
   count = int(userCount)
   when = datetime.datetime.strptime(userDate, "%Y/%m/%d").date()
except ValueError as err:
   print(err)
```

В данном случае мы поручаем Python безопасно разобрать две строки: одну, представляющую `int`, и другую, представляющую `datetime.date`.

Разумеется, иногда интерпретировать одиночные строки, преобразуя их в значения, недостаточно. Например, мы можем написать

приложение с функциями калькулятора или разрешить пользователям создавать фрагменты кода, которые будут подаваться приложению как данные. Примером широко распространенного воплощения таких требований является создание предметно-ориентированных языков (DSL). Подобные языки можно создавать на Python непосредственно – например, написав анализатор на основе рекурсивного спуска. Однако гораздо проще воспользоваться сторонней библиотекой грамматического анализа, к примеру, PLY (`www.dabeaz.com/ply`), PyParsing (`pyparsing.wikispaces.com`) или одной из многих других[3].

Если мы работаем в окружении, где пользователям приложения можно доверять, то можем дать им доступ к самому интерпретатору Python. Интегрированная среда разработки IDLE, включенная в дистрибутив Python, так и делает, хотя мудро выполняет пользовательский код в отдельном процессе, чтобы в случае его аварийного завершения сама IDLE не пострадала.

3.3.1. Вычисление выражения с помощью eval()

Встроенная функция `eval()` вычисляет строку, считая ее выражением (с доступом к переданному ей глобальному или локальному контексту), и возвращает результат. Этого достаточно для построения простого калькулятора `calculator.py`, который мы будем рассматривать в этом подразделе. Сначала посмотрим, как с ним работать.

```
$ ./calculator.py
Введите выражение (для выхода нажать Ctrl+D): 65
A=65
ANS=65
Введите выражение (для выхода нажать Ctrl+D): 72
A=65, B=72
ANS=72
Введите выражение (для выхода нажать Ctrl+D): hypotenuse(A, B)
имя 'hypotenuse' не определено
Введите выражение (для выхода нажать Ctrl+D): hypot(A, B)
A=65, B=72, C=97.0
ANS=97.0
Введите выражение (для выхода нажать Ctrl+D): ^D
```

[3] Введение в грамматический анализ, в том числе с помощью библиотек PLY и PyParsing, имеется в другой книге автора «Программирование на Python 3, второе издание».

Пользователь ввел длины двух катетов прямоугольного треугольника, а затем воспользовался функцией `math.hypot()` (сначала сделав ошибку) для вычисления гипотенузы. После того как выражение введено, программа `calculator.py` распечатывает созданные (и доступные пользователю) к этому моменту переменные и результат вычисления текущего выражения. (Текст, который вводит пользователь, выделен полужирным шрифтом, и подразумевается, что в конце каждой строки он нажимает клавишу `Enter`.)

Чтобы работать с калькулятором было удобнее, мы сохраняем результат каждого выражения в переменной – сначала `A`, потом `B` и так далее, пока не дойдем до `Z`, после чего снова возвращаемся к `A`. Кроме того, мы импортировали все функции и константы из модуля `math` (`hypot()`, `e`, `pi`, `sin()` и т. д.) в пространство имен калькулятора, чтобы пользователь мог обращаться к ним без указания имени модуля (то есть `cos()` вместо `math.cos()`).

Если пользователь введет строку, которую невозможно вычислить, то калькулятор напечатает сообщение об ошибке и повторно выведет приглашение, а весь текущий контекст останется неизменным.

```
def main():
   quit = "Ctrl+Z,Enter" if sys.platform.startswith("win") else "Ctrl+D"
   prompt = "Введите выражение (для выхода нажать {}): ".format(quit)
   current = types.SimpleNamespace(letter="A")
   globalContext = global_context()
   localContext = collections.OrderedDict()
   while True:
      try:
         expression = input(prompt)
         if expression:
            calculate(expression, globalContext, localContext, current)
      except EOFError:
         print()
         break
```

Мы обозначаем конец ввода данных пользователем с помощью константы `EOF` (конец файла). А это означает, что калькулятор можно использовать не только интерактивно, но и в конвейере оболочки, подавая на вход данные из файла, переадресованного на стандартный ввод.

Нам необходимо хранить имя текущей переменной (`A`, `B` и т. д.), чтоб ее можно было обновить после очередного вычисления. Однако просто передать ее в виде строки мы не можем, потому что строки копируются, так что изменить их невозможно. Плохое решение – завести глобальную переменную. Более правильное и встречающееся

куда чаще – создать список из одного элемента, например, `current = ["A"]`. Этот список можно передать как `current`, а для чтения или изменения строки обращаться к ней, как к `current[0]`.

В этом примере мы остановились на более современном подходе и создали крохотное пространство имен с единственным атрибутом (`letter`), значение которого равно `"A"`. Теперь мы можем беспрепятственно передавать пространство имен `current` из одного места в другое, а значение содержащегося в нем атрибута `letter` читать и изменять с помощью элегантной нотации `current.letter`.

Класс `types.SimpleNamespace` появился в версии Python 3.3. В предыдущих версиях эквивалентный эффект можно получить, написав `current = type("_", (), dict(letter="A"))()`. При этом создается новый класс с именем _ и единственным атрибутом `letter` с начальным значением `"A"`. Встроенная функция `type()` возвращает тип объекта, будучи вызвана с одним аргументом, или создает новый класс, если ей передать имя класса, кортеж базовых классов и словарь атрибутов. Если переданный кортеж пуст, то базовым классом будет `object`. Поскольку нам нужен не класс, а только экземпляр, то после вызова `type()` мы сразу же вызываем сам класс – отсюда и дополнительные скобки – и таким образом получаем его экземпляр, который присваиваем переменной `current`.

Python может дать доступ к текущему глобальному контексту с помощью встроенной функции `globals()`; при этом возвращается словарь, который можно модифицировать (например, добавлять в него новые элементы, как мы видели раньше на стр. 35). Python может предоставить и локальный контекст с помощью встроенной функции `locals()`, но модифицировать возвращенный ей словарь *нельзя*.

Мы хотим предоставить глобальный контекст, дополненный константами и функциями из модуля `math`, и первоначально пустой локальный контекст. И если глобальный контекст *обязан* быть словарем, то локальный может быть как словарем (`dict`), так и любым другим объектом-отображением. В данном случае мы решили представить локальный контекст объектом `collections.OrderedDict` – упорядоченным словарем.

Поскольку калькулятор можно использовать интерактивно, мы создали цикл обработки событий, который завершается при обнаружении `EOF`. Внутри цикла мы выводим приглашение пользователю (сообщая ему, как можно выйти из программы) и после ввода текста вызываем функцию `calculate()`, которая должна произвести вычисление и напечатать результаты.

```
import math

def global_context():

   globalContext = globals().copy()
   for name in dir(math):
      if not name.startswith("_"):
         globalContext[name] = getattr(math, name)
   return globalContext
```

Эта вспомогательная функция сначала создает (путем поверхностного копирования) локальный словарь, содержащий глобальные модули, функции и переменные программы. Затем она перебирает все открытые константы и функции в модуле `math` и для каждой добавляет неквалифицированное имя в словарь `globalContext`, присваивая в качестве значения реальную константу или функцию из модуля `math`, на которую это имя ссылается. Так, например, значением ключа `"factorial"`, добавленного в `globalContext`, будет ссылка на функцию `math.factorial()`. Именно таким способом мы даем пользователям калькулятора возможность работать с неквалифицированными именами.

Можно было бы поступить и проще: выполнить `from math import *`, а затем использовать `globals()` напрямую, обойдясь вовсе без словаря `globalContext`. Для модуля `math` такой подход, может, и годится, но выбранный нами путь дает более точный контроль, который может быть полезен для других модулей.

```
def calculate(expression, globalContext, localContext, current):
   try:
      result = eval(expression, globalContext, localContext)
      update(localContext, result, current)
      print(", ".join(["{}={}".format(variable, value)
            for variable, value in localContext.items()]))
      print("ANS={}".format(result))
   except Exception as err:
      print(err)
```

В этой функции мы просим Python вычислить строковое выражение, обращаясь к созданным нами словарям глобального и локального контекста. Если функция `eval()` завершается успешно, то мы обновляем локальный контекст, сохраняя в нем результат, и распечатываем переменные и результат. Если же возникает исключение, мы его печатаем. Поскольку локальный контекст представлен объектом `collections.OrderedDict`, метод `items()` возвращает элемент в порядке вставки даже без явной сортировки (если бы

мы воспользовались обычным словарем, то пришлось бы написать `sorted(localContext.items())`).

Хотя, вообще говоря, перехватывать все исключения, указывая тип `Exception`, не рекомендуется, в данном случае это разумно, потому что при вычислении введенного пользователем выражения может возникать любое исключение.

```
def update(localContext, result, current):
   localContext[current.letter] = result
   current.letter = chr(ord(current.letter) + 1)
   if current.letter > "Z": # мы поддерживаем только 26 переменных
      current.letter = "A"
```

Эта функция присваивает результат следующей по порядку переменной, перебирая их по кругу: `A … Z A … Z …`. Это означает, что после того как пользователь введет 26 выражений, результат последнего станет значением `Z`, результат следующего перезапишет значение `A` и т. д.

Функция `eval()` вычисляет *произвольное* выражение языка Python. Это таит в себе опасность, если выражение получено из ненадежного источника. Можно вместо этого использовать более ограничительную – и более безопасную – функцию из стандартной библиотеки: `ast.literal_eval()`.

3.3.2. Исполнение кода с помощью exec()

Для исполнения произвольного Python-кода можно использовать встроенную функцию `exec()`. В отличие от `eval()`, `exec()` не ограничивается одним выражением и всегда возвращает `None`. Контекст передается `exec()` так же, как `eval()` – с помощью словарей локальных и глобальных объектов. Получить результаты `exec()` можно с помощью переданного ей локального контекста.

В этом подразделе мы рассмотрим программу `genome1.py`. Она создает переменную `genome` (строку, содержащую случайную последовательность букв `A`, `C`, `G` и `T`) и выполняет восемь фрагментов пользовательского кода, передавая геном через контекст.

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

В этом фрагменте показано создание словаря `context`, содержащего данные, с которым будет работать пользовательский код, и последующее выполнение объекта `code`, содержащего пользовательский код, в данном контексте.

```
TRANSFORM, SUMMARIZE = ("TRANSFORM", "SUMMARIZE")
Code = collections.namedtuple("Code", "name code kind")
```

Мы ожидаем, что пользовательский код представлен в виде именованного кортежа `Code`, содержащего описательное имя, сам код (в виде строки) и тип – `TRANSFORM` или `SUMMARIZE`. В ходе выполнения пользовательский код должен создать либо объект `result`, либо объект `error`. Если вид кода – `TRANSFORM`, то ожидается, что `result` содержит новую строку генома, а если `SUMMARIZE`, то `result` должен быть числом. Естественно, мы постараемся, чтобы наша программа корректно обрабатывала ситуацию, когда пользовательский код не отвечает сформулированным выше требованиям.

```
def execute(code, context):
   try:
      exec(code.code, globals(), context)
      result = context.get("result")
      error = context.get("error")
      handle_result(code, result, error)
   except Exception as err:
      print("'{}' raised an exception: {}\n".format(code.name, err))
```

Эта функция выполняет `exec()` для пользовательского кода, передавая собственный глобальный контекст программы и предоставленный локальный контекст. Затем она пытается получить объекты `result` и `error`, один из которых должен был создать пользовательский код, и передает эти объекты функции `handle_result()`.

Как и в примере применения функции `eval()` в предыдущем подразделе, мы перехватываем все исключения (чего вообще-то следует избегать), поскольку пользовательский код может возбудить исключение абсолютно любого типа.

```
def handle_result(code, result, error):
   if error is not None:
      print("'{}' error: {}".format(code.name, error))
   elif result is None:
      print("'{}' produced no result".format(code.name))
   elif code.kind == TRANSFORM:
      genome = result
      try:
         print("'{}' produced a genome of length {}".format(code.name,
                len(genome)))
      except TypeError as err:
         print("'{}' error: expected a sequence result: {}".format(
                code.name, err))
   elif code.kind == SUMMARIZE:
```

```
        print("'{}' produced a result of {}".format(code.name, result))
    print()
```

Если объект `error` не равен `None`, то он печатается. В противном случае, если `result` равен `None`, то печатается сообщение «produced no result». Если результат `result` получен, и вид пользовательского кода равен `TRANSFORM`, то мы присваиваем `result` переменной `genome` и просто печатаем новую длину генома. Блок `try…except` призван защитить нашу программу от ошибок в пользовательском коде (например, возврата единственного значения вместо строки или иной последовательности в случае `TRANSFORM`). Если вид кода – `SUMMARIZE`, то мы просто печатаем итоговую строку с результатом.

В программе `genome1.py` есть восемь вариантов `Code`: первые два (которые мы скоро рассмотрим) порождают корректные результаты, в третьем имеется синтаксическая ошибка, четвертый сообщает об ошибке, пятый не делает ничего, шестой возвращает набор некорректного типа, седьмой вызывает `sys.exit()`, а до восьмого дело не доходит, т. к. седьмой вариант завершает программу. Вот что печатает программа.

```
$ ./genome1.py
'Count' produced a result of 12

'Replace' produced a genome of length 2394

'Exception Test' raised an exception: invalid syntax (<string>, line 4)

'Error Test' error: 'G[AC]{2}TT' not found

'No Result Test' produced no result

'Wrong Kind Test' error: expected a sequence result: object of
type 'int' has no len()
```

Из распечатки видно, что, поскольку пользовательский код исполняется тем же интерпретатором, что сама программа, то он может завершить программу – как нормально, так и аварийно (последняя строка не поместилась на странице и перенесена).

```
Code("Count",
"""
import re
matches = re.findall(target, genome)
if matches:
    result = len(matches)
else:
```

```
   error = "'{}' not found".format(target)
""", SUMMARIZE)
```

Это фрагмент `Code` с именем «Count». Он делает куда больше, чем возможно в одном выражении, которое способна обработать функция `eval()`. Строки `target` и `genome` взяты из объекта `context`, который передан `exec()` в качестве локального контекста, и в том же объекте `context` будут неявно сохраняться новые переменные (в частности, `result` и `error`).

```
Code("Replace",
"""
import re
result, count = re.subn(target, replace, genome)
if not count:
   error = "no '{}' replacements made".format(target)
""", TRANSFORM)
```

Фрагмент `Code` с именем «Replace» выполняет простое преобразование строки `genome`, заменяя неперекрывающиеся подстроки, сопоставляемые с регулярным выражением `target`, строкой `replace`.

Функция `re.subn()` (и метод `regex.subn()`) выполняет подстановки точно так же, как `re.sub()` (и `regex.sub()`). Но если функция (и метод) `sub()` возвращает строку, в которой все замены уже произведены, то функция (и метод) `subn()` возвращает не только строку, но и количество произведенных замен.

Хотя функции `execute()` и `handle_result()` из программы `genome1.py` понять нетрудно, в одном отношении эта программа хрупкая: если пользовательский код «падает» (или просто вызывает `sys.exit()`), то наша программа завершается. В следующем подразделе мы расскажем, как решить эту проблему.

3.3.3. Исполнение кода в подпроцессе

Один из способов выполнить пользовательский код, не ставя под угрозу наше приложение, – запустить отдельный процесс. В этом подразделе мы рассмотрим программы `genome2.py` и `genome3.py`, на примере которых покажем, как запустить интерпретатор Python в подпроцессе, подав ему выполняемую программу на стандартный вход, а результаты получить через стандартный выход.

Мы передаем программам `genome2.py` и `genome3.py` те же восемь фрагментов `Code`, что и программе `genome1.py`. Вот что печатает `genome2.py` (`genome3.py` печатает то же самое):

```
$ ./genome2.py
'Count' produced a result of 12

'Replace' produced a genome of length 2394

'Exception Test' has an error on line 3
   if genome[i] = "A":
                ^
SyntaxError: invalid syntax

'Error Test' error: 'G[AC]{2}TT' not found

'No Result Test' produced no result

'Wrong Kind Test' error: expected a sequence result: object of
type 'int' has no len()

'Termination Test' produced no result

'Length' produced a result of 2406
```

Обратите внимание, что хотя седьмой фрагмент вызывает `sys.exit()`, программа `genome2.py` продолжает работать: для этого фрагмента она просто печатает «produced no result» и переходит к выполнению фрагмента с именем «Length». (Программа `genome1.py` завершилась при вызове `sys.exit()`, поэтому последним напечатала сообщение «...error: expected a sequence...».) Стоит также отметить, что `genome2.py` гораздо лучше сообщает об ошибках (например, в сообщении о синтаксической ошибке во фрагменте «Exception Test»).

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

Создание контекста и выполнение пользовательского кода в этом контексте производятся точно так же, как в программе `genome1.py`.

```
def execute(code, context):
   module, offset = create_module(code.code, context)
   with subprocess.Popen([sys.executable, "-"], stdin=subprocess.PIPE,
            stdout=subprocess.PIPE, stderr=subprocess.PIPE) as process:
      communicate(process, code, module, offset)
```

В начале этой функции мы создаем строку кода (`module`), содержащую пользовательский код плюс вспомогательный код, который мы скоро рассмотрим. В переменной `offset` хранится количество строк, добавленных перед пользовательским кодом, – это даст нам возмож-

ность точно указывать номера строк в сообщениях об ошибках. Затем функция запускает подпроцесс, где исполняет новый экземпляр интерпретатора Python, имя которого находится в `sys.executable`, а аргумент - (дефис) означает, что интерпретатор ожидает поступления Python-кода со стандартного ввода `sys.stdin`[4]. За взаимодействие с процессом – в том числе отправку ему кода в переменной `module` – отвечает написанная нами функция `communicate()`.

```
def create_module(code, context):
   lines = ["import json", "result = error = None"]
   for key, value in context.items():
      lines.append("{} = {!r}".format(key, value))
   offset = len(lines) + 1
   outputLine = "\nprint(json.dumps((result, error)))"
   return "\n".join(lines) + "\n" + code + outputLine, offset
```

Эта функция создает список строк, которые образуют новый Python-модуль, подлежащий исполнению интерпретатором в подпроцессе. В первой строке мы импортируем модуль `json`, с помощью которого будем возвращать результат родительскому процессу (то есть программе `genome2.py`). Во второй строке инициализируются переменные `result` и `error`, которые обязательно должны существовать. Затем мы добавляем по одной строке для каждой контекстной переменной. И наконец, сохраняем переменные `result` и `error` (которые могли быть изменены в пользовательском коде) в JSON-строке, которая печатается на `sys.stdout` после выполнения пользовательского кода.

```
UTF8 = "utf-8"

def communicate(process, code, module, offset):
   stdout, stderr = process.communicate(module.encode(UTF8))
   if stderr:
      stderr = stderr.decode(UTF8).lstrip().replace(", in <module>", ":")
      stderr = re.sub(", line (\d+)",
              lambda match: str(int(match.group(1)) - offset), stderr)
      print(re.sub(r'File."[^"]+?"', "'{}' has an error on line "
              .format(code.name), stderr))
      return
   if stdout:
      result, error = json.loads(stdout.decode(UTF8))
      handle_result(code, result, error)
      return
   print("'{}' produced no result\n".format(code.name))
```

[4] Функция `subprocess.Popen()`, появившаяся в версии Python 3.2, добавила поддержку контекстных менеджеров (то есть предложения `with`).

Первым делом функция `communicate()` отправляет созданный ранее код модуля работающему в подпроцессе интерпретатору Python для исполнения. Затем она приостанавливается в ожидании результатов. После завершения интерпретатора функция считывает данные из `stdout` и `stderr` в одноименные локальные переменные. Отметим, что коммуникация происходит на уровне байтов – именно поэтому мы должны преобразовать строку `module` в последовательность байтов в кодировке UTF-8.

Если произошла какая-то ошибка (возникло исключение или что-то было выведено в `sys.stderr`), то мы подменяем номер строки в сообщении (который учитывает добавленные нами перед пользовательским кодом строки) истинным номером строки в пользовательском коде, а текст «File "<stdin>"» – именем объекта `Code`. После этого мы печатаем текст сообщения об ошибке.

Вызов `re.sub()` ищет – и запоминает – цифры номера строки, отвечающие регулярному выражению `(\d+)`, и заменяет их результатом вызова лямбда-функции, переданной в качестве второго аргумента (чаще в качестве второго аргумента передается строка, но в данном случае нам необходимо проделать кое-какие вычисления). Лямбда-функция преобразует последовательность цифр в целое число, вычитает смещение `offset` и возвращает новый номер строки, которым нужно подменить исходный. В результате номер строки в сообщении об ошибке отражает положение ошибки в пользовательском коде вне зависимости от того, сколько строчек мы добавили в начало при создании модуля, подлежащего интерпретации.

Если ошибок не было, но какие-то данные были помещены в стандартный вывод, то мы преобразуем прочитанные байты в строку (которая должна быть в формате JSON) и выделяем из нее объекты Python – в данном случае кортеж из двух элементов: `result` и `error`. Затем мы вызываем написанную нами функцию `handle_result()` (она во всех программах `genome1.py`, `genome2.py` и `genome3.py` одинакова и была показана на стр. 105).

Пользовательский код в программе `genome2.py` ничем не отличается от содержащегося в `genome1.py`, правда, в `genome2.py` мы добавили вспомогательный технический код до и после пользовательского. Использование формата JSON для возврата результатов удобно и безопасно, но ограничивает типы возвращаемых данных (например, тип `result`) типами `dict`, `list`, `str`, `int`, `float`, `bool` и `None`, причем `dict` и `list` могут содержать объекты только этих же типов.

Программа `genome3.py` отличается от `genome2.py` только тем, что возвращает результаты в формате pickle, позволяющем представить любые типы Python.

```
def create_module(code, context):
   lines = ["import pickle", "import sys", "result = error = None"]
   for key, value in context.items():
      lines.append("{} = {!r}".format(key, value))
   offset = len(lines) + 1
   outputLine = "\nsys.stdout.buffer.write(pickle.dumps((result, error)))"
   return "\n".join(lines) + "\n" + code + outputLine, offset
```

Эта функция очень похожа на вариант из программы `genome2.py`. Мелкое отличие заключается в импорте модуля `sys`, а крупное – в том, что если методы `loads()` и `dumps()` модуля `json` работают со строками, то эквивалентные им функции из модуля `pickle` – с байтами. Поэтому здесь мы должны записывать байты прямо в буфер `sys.stdout`, чтобы избежать непрошеной перекодировки.

```
def communicate(process, code, module, offset):
   stdout, stderr = process.communicate(module.encode(UTF8))
   ...
   if stdout:
      result, error = pickle.loads(stdout)
      handle_result(code, result, error)
      return
```

Метод `communicate()` в программе `genome3.py` такой же, как в `genome2.py`, за исключением строки, в которой вызывается метод `loads()`. В случае данных в формате JSON мы должны были преобразовать последовательность байтов в строку в кодировке UTF-8, здесь же мы работаем с байтами напрямую.

Применение `exec()` для выполнения произвольного Python-кода, полученного от пользователя или от другой программы, дает доступ ко всей мощи интерпретатора Python и его стандартной библиотеки. А выполняя пользовательский код в отдельном подпроцессе, мы можем защитить свою программу от краха или завершения пользовательского кода. Однако воспрепятствовать пользователю сделать что-то вредоносное так не получится. Для выполнения ненадежного кода потребуется какая-то песочница, например, предоставляемая интерпретатором PyPy (`pypy.org`).

В некоторых программах блокировка в ожидании завершения пользовательского кода приемлема, но есть опасность ждать «вечно», если в этом коде присутствует ошибка (например, бесконечный цикл).

Одно из возможных решений – создать подпроцесс в отдельном потоке, а в главном потоке включить таймер. Если таймер срабатывает, то мы можем насильственно завершить подпроцесс и сообщить о проблеме пользователю. Параллельное программирование мы будем рассматривать в следующей главе.

3.4. Паттерн Итератор

Паттерн Итератор позволяет последовательно обойти элементы коллекции или агрегированного объекта, не раскрывая деталей их внутреннего устройства. Он настолько полезен, что в Python встроена его поддержка, а, кроме того, предоставляются специальные методы, которые мы можем реализовать в собственных классах, чтобы органично поддержать итерирование.

Итерирование можно поддержать тремя способами: следуя протоколу последовательности, пользуясь вариантом встроенной функции `iter()` с двумя аргументами или следуя протоколу итератора. Примеры мы приведем в следующих подразделах.

3.4.1. Итераторы, следующие протоколу последовательности

Первый способ поддержать итераторы в собственном классе – удовлетворить требованиям протокола последовательности. Это означает, что мы должны реализовать специальный метод `__getitem__()`, который принимает целочисленный индекс, отсчитываемый от нуля, и возбуждает исключение `IndexError`, когда больше элементов не осталось.

```
for letter in AtoZ():
   print(letter, end="")
print()

for letter in iter(AtoZ()):
   print(letter, end="")
print()

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

В обоих фрагментах создается объект `AtoZ()`, который затем обходится. Сначала объект возвращает символ `"A"`, потом `"B"` и так далее вплоть до `"Z"`. Сделать объект итерируемым можно было бы по-раз-

ному, но в данном случае мы реализовали метод `__getitem__()`, который покажем чуть ниже.

Во втором цикле мы воспользовались встроенной функцией `iter()`, чтобы получить итератор для экземпляра класса `AtoZ`. Ясно, что в данном случае это необязательно, но, как мы скоро увидим (здесь и в другой части книги), у функции `iter()` есть области применения.

```
class AtoZ:

   def __getitem__(self, index):
      if 0 <= index < 26:
         return chr(index + ord("A"))
      raise IndexError()
```

Это весь класс `AtoZ`. Мы включили в него метод `__getitem__()`, удовлетворив тем самым требования протокола последовательности. При обходе объекта этого класса на 27-й итерации будет возбуждено исключение `IndexError`. Если это происходит в цикле `for`, то исключение отбрасывается, цикл благополучно завершается и выполнение продолжается с предложения, следующего за циклом.

3.4.2. Реализация итераторов с помощью функции iter() с двумя аргументами

Еще один способ поддержать итерирование – воспользоваться встроенной функцией `iter()`, передав ей не один, а два аргумента. В этом случае первый аргумент должен быть вызываемым объектом (например, функцией или связанным методом), а второй – значением-ограничителем. При этом вызываемый объект вызывается на каждой итерации – без аргументов – и итерирование завершается, когда он возбуждает исключение `StopIteration` либо возвращает значение-ограничитель.

```
for president in iter(Presidents("George Bush"), None):
   print(president, end=" * ")
print()

for president in iter(Presidents("George Bush"), "George W. Bush"):
   print(president, end=" * ")
print()
```

```
George Bush * Bill Clinton * George W. Bush * Barack Obama *
George Bush * Bill Clinton *
```

Вызов `Presidents()` создает экземпляр класса `Presidents`, а благодаря реализации специального метода `__call__()` такие экземпляры являются вызываемыми объектами. Таким образом, мы создали вызываемый объект `Presidents` (как и требует функция `iter()` с двумя аргументами) и задали ограничитель `None`. Ограничитель обязателен, даже если он равен `None`, чтобы Python знал, что речь идет о варианте `iter()` с двумя, а не с одним аргументом.

Конструктор `Presidents` создает вызываемый объект, который будет по очереди возвращать одного президента за другим, начав с Джорджа Вашингтона или с того, которого мы передадим ему в виде необязательного аргумента. В данном случае мы велели начать с Джорджа Буша. В первом цикле мы указали ограничитель `None`, что означает «иди до конца» – на момент написания этой книги до Барака Обамы. А во втором цикле мы указали в качестве ограничителя имя президента; это означает, что вызываемый объект будет возвращать президентов с первого до предшествующего ограничителю.

```
class Presidents:

    __names = ("George Washington", "John Adams", "Thomas Jefferson",
               ...
               "Bill Clinton", "George W. Bush", "Barack Obama")

    def __init__(self, first=None):
        self.index = (-1 if first is None else
                      Presidents.__names.index(first) - 1)

    def __call__(self):
        self.index += 1
        if self.index < len(Presidents.__names):
            return Presidents.__names[self.index]
        raise StopIteration()
```

В классе `Presidents` хранится статический (на уровне класса в целом) список `__names`, содержащий имена всех президентов США. В методе `__init__()` индексу присваивается начальное значение, на 1 меньше номера первого президента в списке или номера указанного президента.

Объекты любого класса, в котором реализован специальный метод `__call__()`, являются вызываемыми. И когда такой объект вызывается, на самом деле выполняется метод `__call__()`[5].

[5] В языках, где функции не являются полноправными объектами, вызываемые экземпляры называются *функторами*.

В данном случае метод `__call__()` либо возвращает имя следующего президента в списке, либо возбуждает исключение `StopIteration`. На первой итерации того цикла, в котором был задан ограничитель `None`, мы никогда до ограничителя не дойдем (потому что `__call__()` ни в каком случае не возвращает `None`), однако цикл все равно благополучно останавливается, так как, исчерпав всех президентов, мы возбудим исключение `StopIteration`. Во втором случае, когда встроенная функция `iter()` получит президента, совпадающего с ограничителем, она сама возбудит исключение `StopIteration`, чтобы завершить цикл.

3.4.3. Итераторы на базе протокола итераторов

Пожалуй, самый простой способ поддержать итераторы в своем классе – реализовать протокол итераторов. Для этого в классе должен быть реализован специальный метод `__iter__()`, который возвращает объект итератора. У этого объекта должен быть свой метод `__iter__()`, который возвращает сам итератор, и метод `__next__()`, который возвращает следующий элемент или возбуждает исключение `StopIteration`, если элементов не осталось. Предложения `for` и `in` в Python пользуются именно этим протоколом. Реализовать метод `__iter__()` проще всего, сделав его генератором – или заставив возвращать генератор – поскольку генераторы соблюдают протокол итераторов (подробнее о генераторах см. подраздел 3.1.2).

В этом разделе мы создадим простой класс мультимножества. Это класс коллекции, похожий на `set`, но допускающий повторяющиеся элементы. Пример мультимножества показан на рис. 3.3. Естественно, мы сделаем мультимножество итерируемым – и даже тремя способами. Весь код находится в файле `Bag1.py`, если явно не оговорено противное.

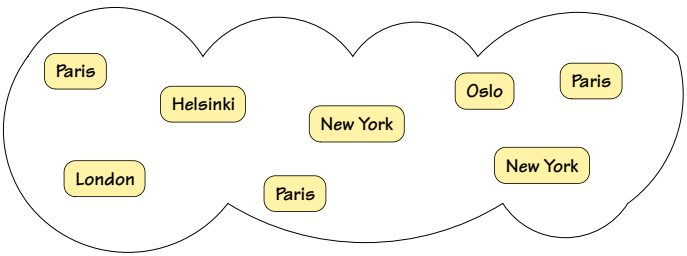

Рис. 3.3. Мультимножество – это неупорядоченная коллекция значений, в которой могут встречаться повторяющиеся элементы

```
class Bag:

   def __init__(self, items=None):
      self.__bag = {}
      if items is not None:
           for item in items:
              self.add(item)
```

Данные мультимножества хранятся в закрытом словаре `self.__bag`. Ключами словаря могут быть произвольные хэшируемые объекты (это элементы мультимножества), а их значениями являются счетчики (сколько раз данный элемент встречается в мультимножестве). При желании пользователь может добавить в только что созданное мультимножество элементы.

```
   def add(self, item):
      self.__bag[item] = self.__bag.get(item, 0) + 1
```

Поскольку `self.__bag` – не `collections.defaultdict`, то увеличивать счетчик можно, только если элемент уже существует, в противном случае мы получили бы исключение `KeyError`. Для чтения счетчика существующего элемента мы вызываем метод `dict.get()`, а если элемента не существует, то возвращаем 0. Затем счетчик элемента увеличивается на 1, причем, если необходимо, элемент сначала создается.

```
   def __delitem__(self, item):
      if self.__bag.get(item) is not None:
         self.__bag[item] -= 1
         if self.__bag[item] <= 0:
            del self.__bag[item]
      else:
         raise KeyError(str(item))
```

При попытке удалить элемент, отсутствующий в мультимножестве, мы возбуждаем исключение `KeyError`, передавая в нем этот элемент в строковой форме. Если же элемент есть в мультимножестве, то мы сначала уменьшаем его счетчик. Если счетчик обращается в нуль, то элемент удаляется.

Мы не стали реализовывать специальные методы `__getitem__()` и `__setitem__()`, потому что для мультимножеств они не имеют смысла (так как мультимножества не упорядочены). Вместо этого для добавления элементов мы пишем *`bag`*`.add()`, для удаления – `del` *`bag[item]`*, а для проверки того, сколько раз элемент встречается в мультимножестве, – *`bag`*`.count(`*`item`*`)`.

```
def count(self, item):
   return self.__bag.get(item, 0)
```

Этот метод просто возвращает число вхождений данного элемента в мультимножество или 0, если элемента там нет. С тем же успехом можно было бы возбуждать исключение `KeyError` при попытке получить число вхождений отсутствующего элемента. Для этого всего-то и нужно было бы написать в теле метода `return self.__bag[item]`.

```
def __len__(self):
   return sum(count for count in self.__bag.values())
```

Тут есть хитрость, потому что мы должны учитывать, сколько раз повторяется каждый элемент. Для этого мы обходим все значения в мультимножестве (то есть счетчики элементов) и суммируем их с помощью встроенной функции `sum()`.

```
def __contains__(self, item):
   return item in self.__bag
```

Этот метод возвращает `True`, если указанный элемент входит в мультимножество хотя бы один раз (поскольку если элемент вообще присутствует в мультимножестве, то его счетчик не меньше 1). В противном случае возвращается `False`.

Итак, мы видели все методы мультимножества, кроме поддержки итераторов. Сначала рассмотрим метод `Bag.__iter__()` из файла `Bag1.py`.

```
def __iter__(self): # Без необходимости создается список элементов!
   items = []
   for item, count in self.__bag.items():
      for _ in range(count):
         items.append(item)
   return iter(items)
```

Это лишь первая попытка. Метод строит список элементов – каждый элемент входит в него столько раз, сколько указывает его счетчик, – а затем возвращает итератор этого списка. Если мультимножество большое, то и список будет очень велик, что само по себе неэффективно. Поэтому рассмотрим два более удачных подхода.

```
def __iter__(self):
   for item, count in self.__bag.items():
      for _ in range(count):
         yield item
```

Этот код взят из модуля `Bag2.py`, и это единственное отличие от класса `Bag` из `Bag1.py`.

Здесь мы обходим элементы мультимножества, получая каждый элемент вместе с его счетчиком, и `count` раз уступаем один и тот же элемент. Создание генератора сопряжено с крохотными фиксированными накладными расходами, не зависящими от количества элементов. И уж, конечно, не надо создавать отдельный список, так что этот метод гораздо эффективнее версии из `Bag1.py`.

```
def __iter__(self):
    return (item for item, count in self.__bag.items()
            for _ in range(count))
```

Это версия метода `Bag.__iter__()` из модуля `Bag3.py`. По существу, она совпадает с версией из `Bag2.py`, только вместо того чтобы превращать метод в генератор, мы возвращаем генераторное выражение.

Хотя приведенные в этой книге реализации мультимножества отлично работают, имейте в виду, что в стандартной библиотеке имеется собственная реализация: `collections.Counter`.

3.5. Паттерн Посредник

Паттерн Посредник предоставляет средства для создания объекта – посредника – который инкапсулирует взаимодействия между другими объектами. Это позволяет осуществлять взаимодействия между объектами, которые ничего не знают друг о друге. Например, в случае нажатия кнопки соответствующий ей объект должен только известить посредника, а уж тот уведомит все объекты, которых интересует нажатие этой кнопки. Посредник для формы, содержащей виджеты текстовых полей и кнопок, и два его метода, показаны на рис. 3.4.

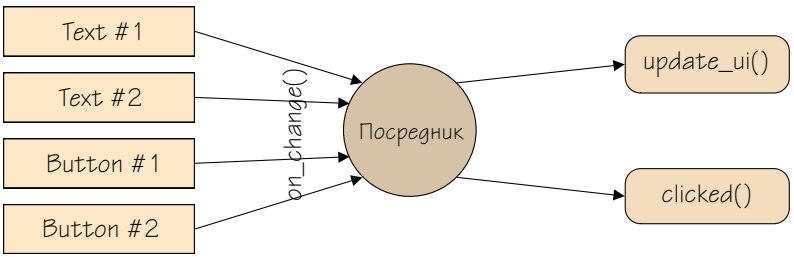

Рис. 3.4. Посредник виджетов формы

Понятно, что этот паттерн находит широкое применение в программировании графических интерфейсов. На самом деле, все библиотеки ГИП для Python (например, Tkinter, PyQt/PySide, PyGObject, wxPython) предлагают тот или иной эквивалентный механизм. Примеры на Tkinter мы увидим в главе 7.

В двух подразделах этого раздела мы рассмотрим два подхода к реализации посредника. Первый вполне традиционный, во втором применяются сопрограммы. В обоих случаях используются классы `Form`, `Button` и `Text` (реализации которых будут показаны), моделирующие фиктивную библиотеку пользовательского интерфейса.

3.5.1. Традиционный Посредник

В этом подразделе мы создадим традиционный посредник – класс, координирующий взаимодействия, – в данном случае для формы. Весь приведенный ниже код взят из файла `mediator1.py`.

```
class Form:

    def __init__(self):
        self.create_widgets()
        self.create_mediator()
```

Как и большинство функций и методов в этой книге, этот метод подвергнут безжалостному рефакторингу – до такой степени, что всю свою работу делегирует другим методам.

```
    def create_widgets(self):
        self.nameText = Text()
        self.emailText = Text()
        self.okButton = Button("OK")
        self.cancelButton = Button("Cancel")
```

В этой форме есть два текстовых поля: имя и почтовый адрес пользователя и две кнопки: `OK` и `Cancel`. Разумеется, в настоящий пользовательский интерфейс мы включили бы и метки, а потом еще и красиво скомпоновали бы виджеты, но сейчас наша цель – продемонстрировать паттерн Посредник, так что ничего этого мы делать не будем. Классы `Text` и `Button` будут показаны ниже.

```
    def create_mediator(self):
        self.mediator = Mediator(((self.nameText, self.update_ui),
                (self.emailText, self.update_ui),
                (self.okButton, self.clicked),
                (self.cancelButton, self.clicked)))
        self.update_ui()
```

Мы создаем единственный объект-посредник для всей формы. Этот объект принимает одну или несколько пар *(виджет, вызываемый объект)*, которые описывают поддерживаемые посредником связи. В данном случае все вызываемые объекты – связанные методы (см. врезку «Связанные и несвязанные методы» на стр. 77). Здесь мы говорим, что если текст в любом из текстовых полей изменяется, то следует вызвать метод `Form.update_ui()`; а если нажата любая кнопка, то нужно вызывать метод `Form.clicked()`. Создав посредник, мы вызываем метод `update_ui()`, который инициализирует форму.

```
def update_ui(self, widget=None):
   self.okButton.enabled = (bool(self.nameText.text) and
                            bool(self.emailText.text))
```

Этот метод активирует кнопку `OK`, если оба текстовых поля не пусты; в противном случае кнопка становится неактивной. Ясно, что этот метод следует вызывать всякий раз, как изменяется текст в каком-нибудь поле.

```
def clicked(self, widget):
   if widget == self.okButton:
      print("OK")
   elif widget == self.cancelButton:
      print("Cancel")
```

Этот метод вызывается при нажатии любой кнопки. В настоящем приложении он, наверное, не просто печатал бы название кнопки, а делал что-нибудь более интересное.

```
class Mediator:

   def __init__(self, widgetCallablePairs):
      self.callablesForWidget = collections.defaultdict(list)
      for widget, caller in widgetCallablePairs:
         self.callablesForWidget[widget].append(caller)
         widget.mediator = self
```

Это первый из двух методов класса `Mediator`. Мы хотим создать словарь, в котором ключами будут виджеты, а значениями – списки, содержащие один или несколько вызываемых объектов. Это можно сделать с помощью словаря по умолчанию. Если производится обращение к элементу словаря по умолчанию, а этот элемент отсутствует, то он создается и добавляется в словарь, причем значение создает вызываемый объект, заданный при создании словаря. В данном случае словарю был передан объект `list`, который, будучи вызван, со-

здает новый пустой список. Таким образом, в первый раз, когда мы ищем в словаре виджет, в словарь вставляется новый элемент с этим виджетом в качестве ключа и пустым списком в качестве значения. После этого мы сразу же добавляем в список вызывающий объект. При последующих обращениях к тому же виджету вызывающий объект добавляется в конец существующего списка элемента. Мы также записываем в атрибут `mediator` виджета (если необходимо, создаем этот атрибут) ссылку на данный посредник (`self`).

Метод добавляет связанные методы в том порядке, в каком они перечислены в парах; если бы порядок нас не интересовал, то можно было бы указать `set` вместо `list` при создании словаря и использовать `set.add()` вместо `list.append()` для добавления связанных методов.

```
    def on_change(self, widget):
        callables = self.callablesForWidget.get(widget)
        if callables is not None:
            for caller in callables:
                caller(widget)
        else:
            raise AttributeError("No on_change() method registered for {}"
                 .format(widget))
```

Всякий раз как изменяется состояние опосредуемого объекта, то есть виджета, переданного посреднику, ожидается, что будет вызван метод посредника `on_change()`. Этот метод затем последовательно вызывает все связанные методы, ассоциированные с данным виджетом.

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```

Это вспомогательный класс, которому могут наследовать опосредуемые классы. Он хранит ссылку на объект посредника и при вызове своего метода `on_change()` вызывает метод `on_change()` посредника, передавая ему данный виджет (то есть `self` – виджет, чье состояние изменилось).

Поскольку этот метод базового класса не модифицируется в подклассах, то можно было бы заменить базовый класс декоратором класса, как мы видели раньше (см. подраздел 2.4.2.2).

```
class Button(Mediated):

   def __init__(self, text=""):
      super().__init__()
      self.enabled = True
      self.text = text

   def click(self):
      if self.enabled:
         self.on_change()
```

Класс `Button` наследует классу `Mediated`. Это наделяет кнопку атрибутом `self.mediator` и методом `on_change()`, который она должна вызывать при изменении состояния, например, после щелчка мышью.

Так, в этом примере вызов `Button.click()` приводит к вызову метода `Button.on_change()` (унаследованного от `Mediated`), в результате чего будет вызван метод `on_change()` посредника, который затем вызовет один или несколько методов, ассоциированных с данной кнопкой, – в нашем случае это метод `Form.clicked()`, которому кнопка передается в аргументе `widget`.

```
class Text(Mediated):

   def __init__(self, text=""):
      super().__init__()
      self.__text = text

   @property
   def text(self):
      return self.__text

   @text.setter
   def text(self, text):
      if self.text != text:
         self.__text = text
         self.on_change()
```

Структурно класс `Text` аналогичен классу `Button` и тоже наследует `Mediated`.

Каким бы ни был виджет (кнопка, текстовое поле и т. д.), коль скоро мы сделаем его подклассом `Mediated` и будем вызывать метод `on_change()` при любом изменении состояния, заботу о взаимодействиях можно оставить посреднику. Разумеется, при создании объекта `Mediator` мы должны зарегистрировать виджеты и вызываемые для них методы. Это означает, что все виджеты в форме слабо связаны, так что удается избежать прямых – и потенциально хрупких – связей.

3.5.2. Посредник на основе сопрограмм

Посредник можно рассматривать как конвейер, который получает сообщения (вызывается `on_change()`) и передает их заинтересованным объектам. Как мы уже видели (подраздел 3.1.2), для реализации такого рода механизма можно использовать сопрограммы. Приведенный ниже код взят из файла `mediator2.py`, а не показанный идентичен тому, что мы видели в предыдущем подразделе.

В этом подразделе мы воспользуемся совершенно другим подходом, нежели в предыдущем. Там мы составляли пары из виджета и ассоциированных с ним методов, и всякий раз как виджет уведомлял о своем изменении, посредник вызывал соответствующие методы. Здесь каждому виджету сопоставляется посредник, который на самом деле является конвейером сопрограмм. При любом изменении состояния виджет отправляет себя в конвейер, а компоненты конвейера (сопрограммы) решают, хотят ли они выполнить какое-нибудь действие в ответ на изменение полученного виджета.

```
def create_mediator(self):
   self.mediator = self._update_ui_mediator(self._clicked_mediator())
   for widget in (self.nameText, self.emailText, self.okButton,
                  self.cancelButton):
      widget.mediator = self.mediator
   self.mediator.send(None)
```

Для версии с сопрограммами нам не нужен отдельный класс посредника. Его место занимает конвейер сопрограмм; в данном случае – с двумя компонентами: `self._update_ui_mediator()` и `self._clicked_mediator()` (оба – методы класса `Form`).

Построив конвейер, мы записываем ссылку на него в атрибут `mediator` каждого виджета. В конце мы отправляем в конвейер `None`. Поскольку нет такого виджета `None`, то действия, ассоциированные с виджетами, не выполняются, зато выполняются действия уровня самой формы (например, активация или деактивация кнопки `OK` в методе `_update_ui_mediator()`).

```
@coroutine
def _update_ui_mediator(self, successor=None):
   while True:
      widget = (yield)
      self.okButton.enabled = (bool(self.nameText.text) and
                               bool(self.emailText.text))
      if successor is not None:
         successor.send(widget)
```

Эта сопрограмма – часть конвейера (декоратор `@coroutine` был показан и обсуждался выше, на стр. 92).

Виджет, уведомляющий об изменении, передается в конвейер, и выражение `yield` возвращает его в переменной `widget`. Что до активации и деактивации кнопки `OK`, так это делается вне зависимости от того, какой виджет изменился (в конце концов, если `widget` равно `None`, то никакой виджет вообще не изменялся, а мы просто инициализируем форму).

```
@coroutine
def _clicked_mediator(self, successor=None):
   while True:
      widget = (yield)
      if widget == self.okButton:
         print("OK")
      elif widget == self.cancelButton:
         print("Cancel")
      elif successor is not None:
         successor.send(widget)
```

Эта конвейерная сопрограмма интересуется только нажатиями на кнопки `OK` и `Cancel`. Если изменившийся виджет совпадает с одной из них, сопрограмма его обрабатывает, иначе пропускает следующей сопрограмме, если таковая имеется.

```
class Mediated:

   def __init__(self):
      self.mediator = None

   def on_change(self):
      if self.mediator is not None:
         self.mediator.send(self)
```

Классы `Button` и `Text` такие же, как в `mediator1.py`, но в классе `Mediated` есть одно небольшое изменении: при вызове метода `on_change()` он отправляет изменившийся виджет (`self`) в конвейер посредника.

В предыдущем подразделе мы отмечали, что класс `Mediated` можно было бы заменить декоратором класса. В примерах к книге имеется файл `mediator2d.py`, в котором так и сделано (см. подраздел 2.4.2.2).

Паттерн Посредник можно также модифицировать для обеспечения мультиплексирования, то есть организации коммуникации многие-ко-многим между объектами. (См. также паттерн Наблюдатель (раздел 3.7) и паттерн Состояние (раздел 3.8).)

3.6. Паттерн Хранитель

Паттерн Хранитель служит для сохранения и восстановления состояния объекта, не нарушая инкапсуляцию.

В языке Python имеется готовая поддержка этого паттерна; с помощью модуля `pickle` мы можем сериализовать и десериализовать произвольные объекты Python (с несколькими ограничениями, например, нельзя сериализовать файловый объект). На самом деле, Python умеет сериализовывать `None`, `bool`, `bytearray`, `byte`, `complex`, `float`, `int` и `str`, а также словари, списки и кортежи, содержащие только объекты, допускающие сериализацию (в том числе коллекции), функции верхнего уровня, классы верхнего уровня и экземпляры пользовательских классов верхнего уровня, в которых атрибут `__dict__` сериализуемый, – то есть объекты большинства пользовательских классов. Того же эффекта можно достичь с помощью модуля `json`, хотя последний поддерживает только базовые типы Python вкупе со словарями и списками (примеры использования `json` и `pickle` мы видели в подразделе 3.3.3).

Даже в тех редких случаях, когда мы наталкиваемся на ограничение сериализации, всегда можно добавить собственную поддержку этого механизма, например, реализовав специальные методы `__getstate__()` и `__setstate__()` и, возможно, метод `__getnewargs__()`. Аналогично, если мы захотим использовать формат JSON для своих классов, то сможем расширить кодировщик и декодировщик из модуля `json`.

Можно было бы также придумать собственный формат и протоколы, но смысла в этом немного, т. к. Python и так предоставляют развитую поддержку этого паттерна.

Десериализация по существу сводится к исполнению произвольного Python-кода, поэтому не рекомендуется десериализовать файлы, полученные из ненадежных источников, в том числе на физических носителях и по сети. В таких случаях JSON безопаснее. Или следует использовать контрольные суммы и шифрование, дабы удостовериться, что сериализованный файлом никто не подменил.

3.7. Паттерн Наблюдатель

Паттерн Наблюдатель поддерживает отношения зависимости многие-ко-многим между объектами, например, когда изменяется состояние одного объекта, уведомляются все связанные с ним объекты.

В наши дни, наверное, самое распространенное воплощение этого паттерна и его вариантов – парадигма модель-представление-контроллер (MVC). В этой парадигме модель описывает данные, одно или несколько представлений эти данные визуализируют, а один или несколько контроллеров осуществляют посредничество между вводом (например, взаимодействием с пользователем) и моделью. И любые изменения в модели автоматически отражаются в ассоциированных с ней представлениях.

У подхода MVC есть одно популярное упрощение – использовать только модели и представления, но поручить представлениям как визуализацию данных, так и передачу модели входных данных; иными словами, представления совмещены с контроллерами. В терминах паттерна Наблюдатель это означает, что представления – наблюдатели модели, а модель – наблюдаемый субъект.

В этом разделе мы создадим модель, которая представляет диапазон значений с минимумом и максимумом (например, полоса прокрутки, виджет ползунок или монитор температуры). И создадим два отдельных наблюдателя (представления) модели: один будет выводить значение модели при каждом его изменении (как вариант индикатора продвижения на HTML), а другой – хранить историю изменений (значения и временные метки). Вот пример прогона программы `observer.py`.

```
$ ./observer.py > /tmp/observer.html
0 2013-04-09 14:12:01.043437
7 2013-04-09 14:12:01.043527
23 2013-04-09 14:12:01.043587
37 2013-04-09 14:12:01.043647
```

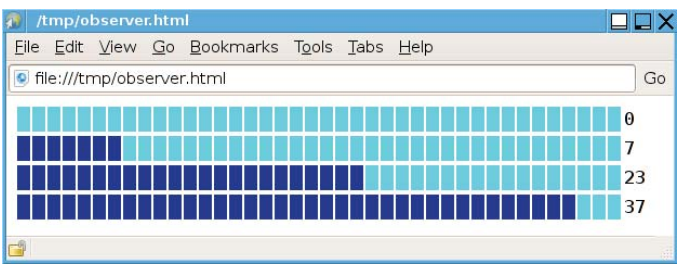

Рис. 3.5. Пример HTML-разметки, которую наблюдатель выводит при изменении модели

Исторические данные посылаются на `sys.stderr`, а HTML-код – на `sys.stdout`, который мы перенаправили в HTML-файл. Полу-

чившаяся HTML-страница показана на рис. 3.5. Программа выводит четыре HTML-таблицы с одной строкой: первая – когда наблюдается пустая модель, остальные – после каждого изменения модели. На рис. 3.6 показан пример архитектуры модель–представление.

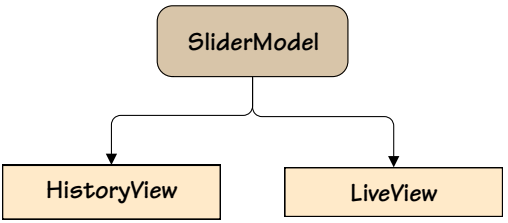

Рис. 3.6. Модель и два представления

В примере для этого раздела, `observer.py`, используется базовый класс `Observed`, который предоставляет средства для добавления, удаления и уведомления наблюдателей. Класс `SliderModel`, представляющий диапазон значений с минимумом и максимумом, наследует `Observed`, поэтому может стать объектом наблюдения. Кроме того, у нас есть два представления, наблюдающих за моделью: `HistoryView` и `LiveView`. Разумеется, мы рассмотрим все эти классы, но сначала взглянем на функцию `main()` программы, чтобы понять, как они используются и как был получен результат, показанный на рис. 3.5 выше.

```
def main():
   historyView = HistoryView()
   liveView = LiveView()
   model = SliderModel(0, 0, 40) # минимум, значение, максимум
   model.observers_add(historyView, liveView) # liveView
                                              # порождает вывод
   for value in (7, 23, 37):
      model.value = value                     # liveView
                                              # порождает вывод
   for value, timestamp in historyView.data:
      print("{:3} {}".format(value, datetime.datetime.fromtimestamp(
              timestamp)), file=sys.stderr)
```

Сначала мы создаем оба представления, затем – модель, в которой минимум равен 0, текущее значение – 0, а максимум – 40. Затем мы регистрируем оба представления как наблюдателей модели. Как только `LiveView` сделан наблюдателем, он сразу же выводит свой первый результат, а когда сделан наблюдателем `HistoryView`, он выводит первое значение и временную метку. После этого мы трижды обнов-

ляем значение модели и при каждом обновлении `LiveView` выводим новую HTML-таблицу с одной строкой, а `HistoryView` – значение и временную метку.

В конце мы печатаем всю историю на `sys.stderr` (то есть на консоль). Функция `datetime.datetime.fromtimestamp()` принимает временную метку (количество секунд от начала эпохи, возвращаемое функцией `time.time()`) и возвращает эквивалентный объект `datetime.datetime`. Метод `str.format()` умеет выводить объекты типа `datetime.datetime` в формате ISO-8601.

```
class Observed:

   def __init__(self):
      self.__observers = set()

   def observers_add(self, observer, *observers):
      for observer in itertools.chain((observer,), observers):
         self.__observers.add(observer)
         observer.update(self)

   def observer_discard(self, observer):
      self.__observers.discard(observer)

   def observers_notify(self):
      for observer in self.__observers:
         observer.update(self)
```

Этот класс предназначен на роль базового для моделей и любых других классов, которые хотят поддержать наблюдение. В классе `Observed` хранится множество наблюдающих объектов. При добавлении каждого такого объекта вызывается его метод `update()`, чтобы он мог запомнить текущее состояние модели. Далее ожидается, что при всяком изменении своего состояния модель будет вызывать унаследованный метод `observers_notify()`, который вызовет метод `update()` каждого наблюдателя (представления), чтобы тот мог представить новое состояние модели.

В методе `observers_add()` есть одна тонкость. Мы хотим иметь возможность добавлять одного или более наблюдателей, но если бы написали просто `*observers`, то можно было бы добавить 0 или более наблюдателей. Поэтому мы требуем, чтобы хотя бы один наблюдатель был (`observer`), а сверх него готовы принять еще 0 или более (`*observers`). Это можно было бы сделать с помощью конкатенации кортежей (например, `for observer in (observer,) + observers:`), но мы воспользовались более эффективной функцией

`itertools.chain()`. Раньше (стр. 59) уже отмечалось, что эта функция принимает произвольное количество итерируемых объектов и возвращает единственный итерируемый объект, который ведет себя как их конкатенация.

```
class SliderModel(Observed):

   def __init__(self, minimum, value, maximum):
      super().__init__()
      # Уже должны существовать перед использованием установщиков свойств
      self.__minimum = self.__value = self.__maximum = None
      self.minimum = minimum
      self.value = value
      self.maximum = maximum

   @property
   def value(self):
      return self.__value

   @value.setter
      def value(self, value):
      if self.__value != value:
         self.__value = value
         self.observers_notify()
   ...
```

Этот класс модели создан специально для рассматриваемого примера, но, конечно, модель могла бы быть любой. Благодаря наследованию классу `Observed` этот класс получает закрытый набор наблюдателей (первоначально пустой) и методы `observers_add()`, `observer_discard()` и `observers_notify()`. При всяком изменении своего состояния – например, при изменении значения – модель должна вызвать метод `observers_notify()`, чтобы наблюдатели могли отреагировать.

В классе есть еще свойства `minimum` и `maximum`, которые здесь опущены; устроены они точно так же, как свойство `value`, и, разумеется, их установщики тоже вызывают метод `observers_notify()`.

```
class HistoryView:

   def __init__(self):
      self.data = []

   def update(self, model):
      self.data.append((model.value, time.time()))
```

Это представление является наблюдателем модели, потому что предоставляет метод `update()`, который принимает наблюдаемую

модель в качестве единственного аргумента, помимо `self`. При каждом вызове метод `update()` добавляет кортеж из двух элементов (*значение*, *временная метка*) в список `self.data`, сохраняя тем самым историю изменений модели.

```
class LiveView:

   def __init__(self, length=40):
      self.length = length
```

Это еще одно представление, наблюдающее за моделью. Параметр `length` – это количество ячеек для представления значения модели в HTML-таблице с одной строкой.

```
   def update(self, model):
      tippingPoint = round(model.value * self.length /
              (model.maximum - model.minimum))
      td = '<td style="background-color: {}"> </td>'
      html = ['<table style="font-family: monospace" border="0"><tr>']
      html.extend(td.format("darkblue") * tippingPoint)
      html.extend(td.format("cyan") * (self.length - tippingPoint))
      html.append("<td>{}</td></tr></table>".format(model.value))
      print("".join(html))
```

Этот метод вызывается при первом наблюдении и при каждом последующем обновлении. Для представления модели он выводит однострочную HTML-таблицу с `self.length` ячейками, закрашивая зелено-голубым цветом пустые ячейки и темно-синим – заполненные. Сколько должно быть ячеек каждого вида, определяется путем вычисления точки перехода между заполненными ячейками (если таковые есть) и пустыми.

Паттерн Наблюдатель широко используется в программировании ГИП, но имеет применения и в контексте других событийно-ориентированных архитектур, например, в программах моделирования и серверах. В качестве примеров можно назвать триггеры в базах данных, систему сигнализации в Django, механизм сигналов и слотов в библиотеке построения графических интерфейсов Qt и многочисленные применения технологии WebSockets.

3.8. Паттерн Состояние

Паттерн Состояние предназначен для создания объектов, поведение которых изменяется при изменении состояния; это означает, что у объекта есть режимы работы.

Для иллюстрации этого паттерна проектирования мы создадим класс мультиплексора с двумя состояниями и методами, поведение которых зависит от того, в каком состоянии находится мультиплексор. Если мультиплексор активен, то он может принимать «соединения» – пары (*имя события*, *обратный вызов*) – где обратный вызов – это произвольный вызываемый объект Python (`lambda`, функция, связанный метод и т. д.). После того как соединение установлено, в ответ на каждое полученное мультиплексором событие вызывается ассоциированный с ним метод (при условии, что мультиплексор находится в активном состоянии). Если же мультиплексор неактивен, то при вызове его методов не происходит вообще ничего.

Чтобы продемонстрировать мультиплексор в действии, мы напишем функции обратного вызова, которые будут подсчитывать полученные события, и присоединим их к активному мультиплексору. Затем мы пошлем ему несколько случайных событий и в конце распечатаем подсчитанные обратными вызовами счетчики. Весь код находится в файле `multiplexer1.py`, а ниже показан результат прогона этой программы:

```
$ ./multiplexer1.py
After 100 active events: cars=150 vans=42 trucks=14 total=206
After 100 dormant events: cars=150 vans=42 trucks=14 total=206
After 100 active events: cars=303 vans=83 trucks=30 total=416
```

После отправки ста случайных событий активному мультиплексору мы изменяем его состояние на неактивное (dormant), а затем посылаем еще сто событий – все они должны быть проигнорированы. Затем мы снова переводим мультиплексор в активное состояние и посылаем следующие сто событий; на них соответствующие обратные вызовы должны реагировать.

Для начала мы посмотрим, как устроен мультиплексор, как устанавливаются соединения и как посылаются события. Затем обратимся к функциям обратного вызова и классу событий. И напоследок рассмотрим сам мультиплексор.

```
totalCounter = Counter()
carCounter = Counter("cars")
commercialCounter = Counter("vans", "trucks")

multiplexer = Multiplexer()
for eventName, callback in (("cars", carCounter),
        ("vans", commercialCounter), ("trucks", commercialCounter)):
multiplexer.connect(eventName, callback)
multiplexer.connect(eventName, totalCounter)
```

Мы начали с создания нескольких счетчиков. Экземпляры этих классов вызываемые, поэтому их можно использовать всюду, где нужна функция (например, в качестве обратного вызова). Эти объекты поддерживают по одному независимому счетчику для каждого переданного имени или единственный анонимный счетчик, если имя не задано (как в случае `totalCounter`).

Затем мы создаем мультиплексор (по умолчанию активный). После этого мы соединяем функции обратного вызова с событиями. Нас интересуют три имени событий: «cars», «vans» и «trucks». Функция `carCounter()` соединена с событием «cars», функция `commercialCounter()` – с событиями «vans» и «trucks», а функция `totalCounter()` – со всеми тремя событиями.

```
for event in generate_random_events(100):
   multiplexer.send(event)
print("After 100 active events: cars={} vans={} trucks={} total={}"
       .format(carCounter.cars, commercialCounter.vans,
               commercialCounter.trucks, totalCounter.count))
```

В этом фрагменте мы генерируем сто случайных событий и по одному посылаем их мультиплексору. Если, например, послано событие «cars», то мультиплексор вызовет функции `carCounter()` и `totalCounter()`, передавая им событие в качестве единственного аргумента. Аналогично, если было передано событие «vans» или «trucks», то будут вызваны функции `commercialCounter()` и `totalCounter()`.

```
class Counter:

   def __init__(self, *names):
      self.anonymous = not bool(names)
      if self.anonymous:
         self.count = 0
      else:
         for name in names:
            if not name.isidentifier():
               raise ValueError("names must be valid identifiers")
            setattr(self, name, 0)
```

Если ни одного имени не передано, то создается анонимный счетчик, значение которого хранится в `self.count`. В противном случае создаются независимые счетчики для каждого имени с помощью встроенной функции `setattr()`. Например, у объекта `carCounter` будет атрибут `self.cars`, а у объекта `commercialCounter` – атрибуты `self.vans` и `self.trucks`.

```
    def __call__(self, event):
        if self.anonymous:
            self.count += event.count
        else:
            count = getattr(self, event.name)
            setattr(self, event.name, count + event.count)
```

При вызове объекта `Counter` на самом деле вызывается этот специальный метод. Если счетчик анонимный (как, например, `totalCounter`), то `self.count` увеличивается на единицу. В противном случае мы пытаемся найти атрибут, соответствующий имени события. Например, если событие называется `"trucks"`, то мы запишем в `count` значение `self.trucks`. Затем мы изменяем значение атрибута, прибавляя к старому счетчику величину, взятую из события.

Мы не передали при вызове встроенной функции `getattr()` значение по умолчанию, поэтому если указанного атрибута (например, `"truck"`) не существует, то метод, как и положено, возбудит исключение `AttributeError`. Заодно это гарантирует, что мы не создадим по ошибке атрибут с неправильным именем, поскольку в таких случаях до вызова `setattr()` дело вообще не дойдет.

```
class Event:

    def __init__(self, name, count=1):
        if not name.isidentifier():
            raise ValueError("names must be valid identifiers")
        self.name = name
        self.count = count
```

Это весь класс `Event`. Он очень прост, потому что нужен лишь как часть инфраструктуры для демонстрации паттерна Состояние, воплощенного в классе `Multiplexer`. Кстати говоря, класс `Multiplexer` является также примером паттерна Наблюдатель (см. раздел 3.7).

3.8.1. Чувствительные к состоянию методы

К обработке состояния, хранящегося внутри класса, есть два основных подхода. Первый – использование методов, чувствительных к состоянию, – мы рассмотрим в этом подразделе. Второй – использование определяемых состоянием методов – тема следующего подраздела.

```
class Multiplexer:

   ACTIVE, DORMANT = ("ACTIVE", "DORMANT")

   def __init__(self):
      self.callbacksForEvent = collections.defaultdict(list)
      self.state = Multiplexer.ACTIVE
```

У класса `Multiplexer` два состояния (или режима работы): `ACTIVE` и `DORMANT`. Если экземпляр `Multiplexer` находится в состоянии `ACTIVE`, то его чувствительные к состоянию методы делают что-то полезное, иначе не делают ничего. Сразу после создания объект `Multiplexer` находится в состоянии `ACTIVE`.

Ключами словаря `self.callbacksForEvent` являются имена событий, а значениями – списки вызываемых объектов.

```
   def connect(self, eventName, callback):
      if self.state == Multiplexer.ACTIVE:
         self.callbacksForEvent[eventName].append(callback)
```

Этот метод служит для создания ассоциации между именованным событием и обратным вызовом. Если указанного имени события еще нет в словаре, то, поскольку `self.callbacksForEvent` – словарь по умолчанию, будет автоматически создан и возвращен элемент с таким ключом и пустым списком в качестве значения. Если же имя события уже есть в словаре, то будет возвращен ассоциированный с ним список. Таким образом, мы в любом случае получаем список, в который можно дописать новый обратный вызов (словари по умолчанию обсуждались на стр. 120).

```
   def disconnect(self, eventName, callback=None):
      if self.state == Multiplexer.ACTIVE:
         if callback is None:
            del self.callbacksForEvent[eventName]
         else:
            self.callbacksForEvent[eventName].remove(callback)
```

Если при вызове этого метода обратный вызов не указан, то мы считаем, что пользователь хочет разорвать все обратные вызовы, ассоциированные с указанным именем события. В противном случае из списка обратных вызовов для данного события удаляется только указанный.

```
   def send(self, event):
      if self.state == Multiplexer.ACTIVE:
         for callback in self.callbacksForEvent.get(event.name, ()):
            callback(event)
```

Если событие отправлено мультиплексору и мультиплексор активен, то этот метод перебирает все обратные вызовы, ассоциированные с этим событием (их может и не быть), и вызывает каждый, передавая событие в качестве аргумента.

3.8.2. Определяемые состоянием методы

Программа `multiplexer2.py` отличается от `multiplexer1.py` только тем, что в классе `Multiplexer` используются определяемые состоянием, а не чувствительные к состоянию методы. В классе `Multiplexer` те же два состояния и тот же метод `__init__()`, что и прежде. Однако атрибут `self.state` теперь является свойством.

```
@property
def state(self):
   return (Multiplexer.ACTIVE if self.send == self.__active_send
           else Multiplexer.DORMANT)
```

В этой версии мультиплексора состояние как таковое не хранится. Вместо этого оно вычисляется путем проверки того, совпадает ли один из открытых методов с активным или пассивным закрытым методом.

```
@state.setter
def state(self, state):
   if state == Multiplexer.ACTIVE:
      self.connect = self.__active_connect
      self.disconnect = self.__active_disconnect
   self
      self.connect = lambda *args: None
      self.disconnect = lambda *args: None
      self.send = lambda *args: None
```

При изменении состояния установщик свойства `state` переключает набор методов мультиплексора в соответствии с новым состоянием. Так, если новое состояние равно `DORMANT`, то открытым методам присваиваются методы, являющиеся лямбда-выражениями.

```
def __active_connect(self, eventName, callback):
   self.callbacksForEvent[eventName].append(callback)
```

Это закрытый «активный» метод: в каждый момент времени соответствующему открытому методу присваивается либо он, либо анонимный ничего не делающий метод, реализованный в виде лямбда-выражения. Мы не приводим закрытые методы `disconnect` и `send`, потому что они устроены точно так же. Важно отметить, что ни один

из них не проверяет состояние экземпляра (поскольку вызывается только в соответствующем состоянии), поэтому они несколько проще и на какие-то мгновения быстрее.

Естественно, было бы нетрудно написать версию `Multiplexer` на основе сопрограмм, но поскольку несколько таких примеров мы уже видели, то не станем показывать еще один (впрочем, в программе `multiplexer3.py` продемонстрирован возможный подход к мультиплексированию с помощью сопрограмм).

Хотя мы применили паттерн Состояние для реализации мультиплексора, объекты с состоянием (их еще называют модальными) встречаются в самых разных контекстах.

3.9. Паттерн Стратегия

Паттерн Стратегия позволяет инкапсулировать набор взаимозаменяемых алгоритмов, из которых пользователь выбирает тот, что ему нужен.

Так, в этом разделе мы напишем два разных алгоритма организации списка с произвольным количеством элементов в виде таблицы с указанным числом строк. Один алгоритм выводит фрагмент HTML-разметки – на рис. 3.7 показаны результаты для таблиц с двумя, тремя и четырьмя строками. Другой порождает результат в виде простого текста, показанный ниже.

```
$ ./tabulator3.py
...
+------------------+------------------+------------------+
| Nikolai Andrianov | Matt Biondi      | Bjшrn Dжhlie     |
| Birgit Fischer   | Sawao Kato       | Larisa Latynina  |
| Carl Lewis       | Michael Phelps   | Mark Spitz       |
| Jenny Thompson   |                  |                  |
+------------------+------------------+------------------+
+------------------+------------------+
| Nikolai Andrianov | Matt Biondi      |
| Bjшrn Dжhlie     | Birgit Fischer   |
| Sawao Kato       | Larisa Latynina  |
| Carl Lewis       | Michael Phelps   |
| Mark Spitz       | Jenny Thompson   |
+------------------+------------------+
```

Параметризовать алгоритм можно разными способами. Очевидный подход – создать класс `Layout`, который принимает экземпляр `Tabulator`, который строит ту или иную таблицу. Такой подход применен в программе `tabulator1.py` (не показана). Его можно улуч-

шить, используя табуляторы без состояния, имеющие только статические методы, и передавать алгоритму класс, а не экземпляр табулятора. Так сделано в программе `tabulator2.py` (тоже не показана).

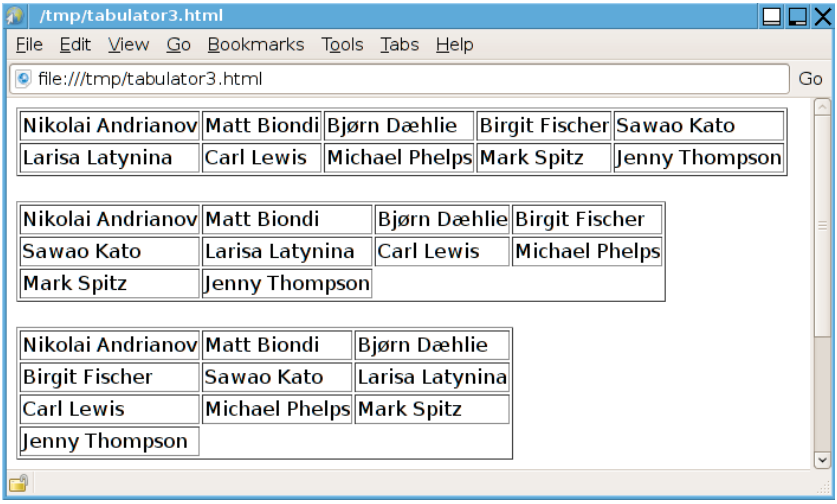

Рис. 3.7. Фрагмент HTML-таблицы, созданный табулятором

В этом разделе мы покажем более простой, но при этом еще более удачный прием: класс `Layout`, который принимает функцию табуляции, реализующую нужный алгоритм.

```
WINNERS = ("Nikolai Andrianov", "Matt Biondi", "Bjшrn Dжhlie",
          "Birgit Fischer", "Sawao Kato", "Larisa Latynina", "Carl Lewis",
          "Michael Phelps", "Mark Spitz", "Jenny Thompson")

def main():
   htmlLayout = Layout(html_tabulator)
   for rows in range(2, 6):
      print(htmlLayout.tabulate(rows, WINNERS))
   textLayout = Layout(text_tabulator)
   for rows in range(2, 6):
      print(textLayout.tabulate(rows, WINNERS))
```

В этой функции создаются два объекта `Layout`, параметризованные различными функциями-табуляторами. Для каждого формата печатается таблица с двумя, тремя, четырьмя и пятью строками.

```
class Layout:

   def __init__(self, tabulator):
```

```
        self.tabulator = tabulator

    def tabulate(self, rows, items):
        return self.tabulator(rows, items)
```

Этот класс поддерживает только один алгоритм: табуляция. Функция, реализующая этот алгоритм, ожидает получить счетчик строк и последовательность элементов, а возвращает результаты в виде таблицы.

На самом деле, этот класс можно было бы еще упростить: вот версия из файла `tabulator4.py`.

```
class Layout:

    def __init__(self, tabulator):
        self.tabulate = tabulator
```

Здесь мы сделали атрибут `self.tabulate` вызываемым объектом (переданной функцией табуляции). Вызов из функции `main()` работает точно так же, как для классов `Layout` из программ `tabulator3.py` и `tabulator4.py`.

Хотя сами алгоритмы табуляции с точки зрения этого паттерна проектирования несущественны, для полноты мы все же рассмотрим один из них.

```
    def html_tabulator(rows, items):
        columns, remainder = divmod(len(items), rows)
        if remainder:
            columns += 1
        column = 0
        table = ['<table border="1">\n']
        for item in items:
            if column == 0:
                table.append("<tr>")
                table.append("<td>{}</td>".format(escape(str(item))))
                column += 1
                if column == columns:
                    table.append("</tr>\n")
                column %= columns
        if table[-1][-1] != "\n":
            table.append("</tr>\n")
        table.append("</table>\n")
        return "".join(table)
```

В обеих функциях табуляции мы должны вычислить, сколько столбцов необходимо для размещения всех элементов в таблице с заданным числом строк. Зная это число (`columns`), мы можем обойти

все элементы, следя за текущим номером столбца (`column`) в текущей строке.

Функция `text_tabulator()` (не показана) немного длиннее, но по существу устроена аналогично.

В более серьезной программе алгоритмы могли бы радикально отличаться – как в плане кода, так и характеристик производительности – чтобы пользователь мог выбрать наиболее подходящий для конкретных условий компромисс. Подстановка алгоритма в виде вызываемого объекта – лямбды, функций, связанных методов – не вызывает трудностей, потому что в Python вызываемые объекты – полноправные элементы языка, то есть их можно передавать и сохранять в коллекциях, как любые другие объекты.

3.10. Паттерн Шаблонный метод

Паттерн Шаблонный метод позволяет определить шаги алгоритма, оставив реализацию некоторых шагов подклассам.

В этом разделе мы создадим класс `AbstractWordCounter`, в котором будет два метода. Первый, `can_count(filename)`, должен возвращать булево значение, показывающее, может ли класс подсчитать слова в заданном файле (определяется путем анализа расширения имени файла). Второй, `count(filename)`, должен возвращать количество слов. Мы напишем также два подкласса: для подсчета слов в простых текстовых файлах и в HTML-файлах. Сначала посмотрим на эти классы в действии (код взят из файла `wordcount1.py`):

```
def count_words(filename):
   for wordCounter in (PlainTextWordCounter, HtmlWordCounter):
      if wordCounter.can_count(filename):
         return wordCounter.count(filename)
```

Все методы во всех классах сделаны статическими. Это означает, что никакое состояние в экземплярах не запоминается (поскольку и экземпляров-то нет) и что мы можем работать непосредственно с объектами классов, а не с экземплярами (можно было бы сделать методы нестатическими и использовать экземпляры, если бы нам нужно было хранить состояние).

Здесь мы перебираем оба подсчитывающих слова подкласса и, если один из них умеет считать слова в данном файле, то мы поручаем ему это сделать и возвращаем счетчик. Если ни один не может это сделать, то мы (неявно) возвращаем `None`, сообщая, что не сумели справиться с подсчетом.

```
class AbstractWordCounter:

   @staticmethod
   def can_count(filename):
      raise NotImplementedError()

   @staticmethod
   def count(filename):
      raise NotImplementedError()
```

```
class AbstractWordCounter(
       metaclass=abc.ABCMeta):

   @staticmethod
   @abc.abstractmethod
   def can_count(filename):
      pass

   @staticmethod
   @abc.abstractmethod
   def count(filename):
      pass
```

Этот чисто абстрактный класс описывает интерфейс подсчета слов, а методы этого интерфейса должны быть реализованы в подклассах. В левом фрагменте, взятом из файла `wordcount1.py`, принят традиционный подход, а в правом, взятом из файла `wordcount2.py`, – более современный, с применением модуля `abc` (**a**bstract **b**ase **c**lass).

```
class PlainTextWordCounter(AbstractWordCounter):

   @staticmethod
   def can_count(filename):
      return filename.lower().endswith(".txt")

   @staticmethod
   def count(filename):
   if not PlainTextWordCounter.can_count(filename):
      return 0
   regex = re.compile(r"\w+")
   total = 0
   with open(filename, encoding="utf-8") as file:
      for line in file:
         for _ in regex.finditer(line):
            total += 1
   return total
```

В этом подклассе реализован интерфейс подсчета слов с крайне упрощенным представлением о том, что такое слово, и в предположении, что все `txt`-файлы хранятся в кодировке UTF-8 (или в 7-разрядной кодировке ASCII, которая является подмножеством UTF-8).

```
class HtmlWordCounter(AbstractWordCounter):

   @staticmethod
   def can_count(filename):
```

```
        return filename.lower().endswith((".htm", ".html"))

    @staticmethod
    def count(filename):
        if not HtmlWordCounter.can_count(filename):
            return 0
        parser = HtmlWordCounter.__HtmlParser()
        with open(filename, encoding="utf-8") as file:
            parser.feed(file.read())
        return parser.count
```

В этом подклассе интерфейс подсчета слов реализован для HTML-файлов. Здесь используется наш собственный закрытый анализатор HTML (который является подклассом `html.parser.HTMLParser`, погруженным в класс `HtmlWordCounter`, как мы скоро увидим). Имея закрытый анализатор HTML, для подсчета слов в HTML-файле нам остается только создать экземпляр анализатора и передать ему файл для разбора. По завершении разбора мы вернем счетчик слов, любезно сохраненный для нас анализатором.

Для полноты картины рассмотрим внутренний класс `HtmlWordCounter.__HtmlParser`, который собственно и занимается подсчетом. HTML-анализатор из стандартной библиотеки Python работает по принципу SAX-анализатора (Simple API for XML), то есть читает текст и вызывает методы, распознав то или иное событие (например, «начальный тег», «конечный тег» и т. д.). Поэтому, чтобы воспользоваться анализатором, мы должны создать его подкласс, в котором реализованы методы, обрабатывающие интересные нам события.

```
class __HtmlParser(html.parser.HTMLParser):

    def __init__(self):
        super().__init__()
        self.regex = re.compile(r"\w+")
        self.inText = True
        self.text = []
        self.count = 0
```

Мы сделали внутренний подкласс `html.parser.HTMLParser` закрытым и включили в него четыре атрибута. В `self.regex` хранится простое определение «слова» (последовательность из одной или более букв, цифр и подчерков). В `self.inText` хранится флаг, показывающий, является ли анализируемый сейчас текст видимым пользователю (в отличие от текста внутри тегов `<script>` или `<style>`). В `self.text` хранится один или несколько кусков, составляющих текущее значение, а в `self.count` – счетчик слов.

```
def handle_starttag(self, tag, attrs):
   if tag in {"script", "style"}:
      self.inText = False
```

Имя и сигнатура этого метода (как и всех методов `handle_…()`) определены базовым классом. По умолчанию методы-обработчики не делают ничего, поэтому мы должны переопределить те, которые нам интересны.

Мы не хотим учитывать слова внутри скриптов и таблиц стилей, так что, встречая соответствующие начальные теги, выключаем сбор текста.

```
def handle_endtag(self, tag):
   if tag in {"script", "style"}:
      self.inText = True
   else:
      for _ in self.regex.finditer(" ".join(self.text)):
        self.count += 1
      self.text = []
```

Дойдя до тега, завершающего скрипт или таблицу стилей, мы снова включаем сбор текста. Во всех остальных случаях мы обходим собранный текст и подсчитываем слова в нем. После этого мы отбрасываем собранный текст, очищая список.

```
def handle_data(self, text):
   if self.inText:
      text = text.rstrip()
      if text:
         self.text.append(text)
```

Если мы получили текст, не находясь внутри скрипта или таблицы стилей, то подбираем его.

Благодаря поддержке закрытых вложенных классов в Python и библиотечному классу `html.parser.HTMLParser` мы можем производить достаточно изощренный разбор, скрывая детали от пользователей класса `HtmlWordCounter`.

Паттерн Шаблонный метод в некоторых отношениях похож на рассмотренный в разделе 2.2 паттерн Мост.

3.11. Паттерн Посетитель

Паттерн Посетитель используется, когда нужно применить некую функцию к каждому элементу коллекции или объекту-агрегату. Это

не то же самое, что типичное использование паттерна Итератор (раздел 3.4) – где мы обходим коллекцию или агрегат и вызываем некий метод для каждого элемента, – поскольку в случае «посетителя» вызывается внешняя функция, а не метод.

В Python имеется встроенная поддержка этого паттерна. Например, конструкция *newList* = map(*function*, *oldSequence*) означает, что *function()* вызывается для каждого элемента oldSequence, в результате чего порождается *newList*. То же самое можно сделать, воспользовавшись списковым включением: *newList* = [*function*(*item*) for *item* in *oldSequence*].

Если нам нужно применить функцию к каждому элементу коллекции или объекта-агрегата, то можно обойти его в цикле for: for *item* in *collection*: *function*(*item*). Если типы элементов различны, то можно использовать предложения if и встроенную функцию isinstance(), чтобы выполнять тот или иной код внутри *function()* в зависимости от типа элемента.

Для некоторых поведенческих паттернов в Python имеется прямая поддержка; остальные нетрудно реализовать самостоятельно. Паттерны Цепочка ответственности, Посредник и Наблюдатель можно реализовать традиционным способом или с помощью сопрограмм, и все они являются вариациями на тему разрыва связи между взаимодействующими объектами. Паттерн Команда можно использовать для отложенного вычисления и реализации механизма выполнение-отмена. Поскольку Python – интерпретируемый язык (на уровне байт-кода), то паттерн Интерпретатор можно реализовать с помощью самого Python и даже изолировать интерпретируемый код в отдельном процессе. Поддержка паттерна Итератор (и – неявно – паттерна Посетитель) встроена в Python. Паттерн Хранитель неплохо поддержан в стандартной библиотеке Python (например, с помощью модулей pickle и json). У паттернов Состояние, Стратегия и Шаблонный метод прямой поддержки нет, но все они легко реализуются.

Паттерны проектирования предлагают полезные способы рассуждения о коде, организации и реализации кода. Некоторые паттерны применимы только в объектно-ориентированной парадигме, тогда как другие равным образом годятся и для процедурного стиля программирования. С момента публикации оригинальной книги по паттернам проектирования эта тема была – и остается – предметом активных исследований. Прекрасной отправной точкой для получения

дополнительных сведений является сайт образовательной некоммерческой группы Hillside Group (`hillside.net`).

В следующей главе мы рассмотрим другую парадигму – параллельное программирование – и попытаемся повысить производительность за счет современных многоядерных процессоров. Но прежде мы займемся первым большим примером – разработаем пакет обработки изображений, которым будем неоднократно пользоваться для разных целей на протяжении этой книги.

3.12. Пример: пакет обработки изображений

В стандартной библиотеке Python нет ни одного модуля для обработки изображений. Однако можно создавать, загружать и сохранять изображения с помощью класса `tk.PhotoImage`, входящего в Tkinter (в файле `barchart2.py` имеется пример, показывающий, как это делается). К сожалению, Tkinter умеет читать и записывать только не слишком популярные графические форматы GIF, PPM и PGM, хотя после включения Tcl/Tk 8.6 в Python будет поддерживаться еще и популярный формат PNG. Но даже в этом случае класс `tk.PhotoImage` можно будет использовать только в одном потоке (в главном потоке ГИП), так что захоти мы обрабатывать несколько изображений одновременно, он окажется бесполезен.

Конечно, можно было бы взять стороннюю графическую библиотеку, например Pillow (`github.com/python-imaging/Pillow`) или воспользоваться другой библиотекой построения ГИП[6]. Но мы решили в этом примере реализовать собственный графический пакет, а заодно использовать его для других целей впоследствии.

Мы хотим, чтобы наш пакет мог эффективно хранить данные изображения и работать с Python без каких-либо дополнений. Для этого мы будем представлять изображение в виде линейного массива цветов. Каждый цвет (то есть каждый пиксель) представляется 32-разрядным целым без знака, четыре байта которого служат для представления красной, зеленой, синей компоненты и альфа-канала (прозрачности); такой формат иногда называют ARGB. Поскольку мы используем одномерный массив, то пиксель с координатами x, y хранится в элементе с индексом $(y \times width) + x$. Это показано на

[6] Для построения двумерных графиков можно было бы взять стороннюю библиотеку `matplotlib` (`matplotlib.org`).

рис. 3.8, где подсвечен пиксель в изображении размером 8 × 8 с координатами (5, 1); его индекс в массиве равен 13 ((1 × 8) + 5).

В стандартной библиотеке Python имеется модуль `array` для работы с одномерными типизированными массивами, и для наших целей он идеально подходит. Однако сторонний модуль `numpy` предлагает высоко оптимизированный код работы с массивами (произвольной размерности), и было бы хорошо воспользоваться им, если он имеется. Поэтому мы спроектируем пакет `Image`, так чтобы он использовал `numpy`, если возможно, и `array` в качестве резервного варианта. Это означает, что пакет `Image` будет работать в любом случае, но не сможет задействовать все возможности `numpy`, так как объекты `array.array` и `numpy.ndarray` должны быть взаимозаменяемы.

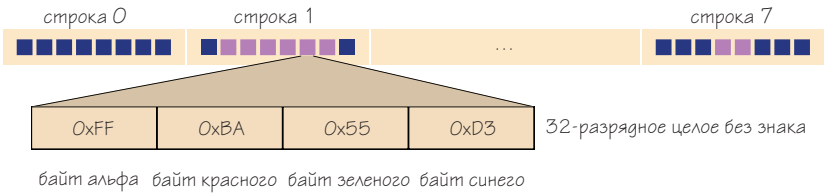

Рис. 3.8. Массив значений цветов для изображения 8 × 8

Мы хотим создавать и модифицировать произвольные изображения; но еще мы хотим уметь загружать существующие изображения и сохранять созданные или модифицированные изображения. Поскольку загрузка и сохранение зависят от формата, то мы спроектировали пакет `Image`, так что в нем есть единый модуль для обобщенной обработки изображений и по одному модулю на каждый графический формат для сохранения и загрузки. Кроме того, мы сделаем так, что пакет сможет автоматически подхватывать вновь добавленные модули обработки изображений – даже после развертывания – при условии, что они согласованы с требованиями его интерфейса.

Пакет `Image` состоит из четырех модулей. Модуль `Image/__init__.py` предоставляет общую функциональность, а три других – код загрузки и сохранения, зависящий от формата. Это `Image/Xbm.py` для монохромного формата XBM (`.xbm`), `Image/Xpm.py` – для цветных растровых изображений XPM (`.xpm`) и `Image/Png.py` –для формата PNG (`.png`). Формат PNG очень сложен, и уже существует Python-модуль для его поддержки – PyPNG (`github.com/drj11/pypng`) – поэтому наш модуль `Png.py` будет просто тонкой оберткой вокруг него (благодаря паттерну Адаптер из раздела 2.1), если, конечно, он доступен.

Мы начнем с общего модуля обработки изображений (`Image/__init__.py`). Затем рассмотрим модуль `Image/Xpm.py`, опуская низкоуровневые детали. И наконец, приведем полностью модуль-обертку `Image/Png.py`.

3.12.1. Общий модуль обработки изображений

Модуль `Image` предоставляет класс `Image` и ряд вспомогательных функций и констант, полезных для обработки изображений.

```
try:
   import numpy
except ImportError:
   numpy = None
   import array
```

Один из ключевых вопросов – как представлять данные изображения: в виде `array.array` или `numpy.ndarray`. Поэтому после обычных предложений импорта мы пытаемся импортировать `numpy`. Если это не получается, мы переходим на резервный вариант – импортируем стандартный модуль `array`, который содержит необходимую функциональность, и записываем в переменную `numpy` значение `None` для тех немногих мест, где различие между `array` и `numpy` имеет значение.

Мы хотим, чтобы пользователь мог получить доступ к нашему модулю, просто написав `import Image`. И этого должно быть достаточно, для того чтобы автоматически подгрузились модули с конкретными функциями загрузки и сохранения. Таким образом, чтобы создать и сохранить красный квадрат размером 64×64, пользователю будет достаточно написать такой код:

```
import Image
image = Image.Image.create(64, 64, Image.color_for_name("red"))
image.save("red_64x64.xpm")
```

Мы хотим, чтобы этот код работал, хотя пользователь явно не импортировал модуль `Image/Xpm.py`. И разумеется, мы хотим, чтобы он работал и для любых других модулей форматов, которые окажутся в каталоге `Image`, даже если они помещены туда после первоначального развертывания пакета `Image`.

Чтобы поддержать все эти «хотелки», мы включили в файл `Image/__init__.py` код, который автоматически пытается загрузить модули для всех форматов.

```
_Modules = []
for name in os.listdir(os.path.dirname(__file__)):
   if not name.startswith("_") and name.endswith(".py"):
      name = "." + os.path.splitext(name)[0]
      try:
         module = importlib.import_module(name, "Image")
         _Modules.append(module)
      except ImportError as err:
         warnings.warn("failed to load Image module: {}".format(err))
del name, module
```

Этот код помещает в закрытый список `_Modules` все модули, найденные в каталоге `Image`, за исключением `__init__.py` (и всех прочих модулей, имена которых начинаются знаком подчеркивания).

Мы перебираем все файлы в каталоге `Image` (где бы он ни находился в файловой системе). Для каждого подходящего файла с расширением `.py` мы строим имя модуля, исходя из имени файла. Нужно не забыть добавить в начало имени модуля `.` (точку), потому что мы собираемся импортировать модуль относительно пакета `Image`. При использовании такого относительного импорта необходимо указать имя пакета во втором аргументе функции `importlib.import_module()`. Если импорт завершился успешно, то мы добавляем соответствующий объект Python-модуля в список модулей; а как они используются, мы скоро увидим.

Чтобы не засорять пространство имен `Image`, мы удалили переменные `name` и `module`, поскольку они больше не понадобятся.

Показанный здесь подход на основе подключаемых модулей, прост, понятен и в большинстве случаев отлично работает. Однако у него есть одно ограничение: он не будет работать, если пакет `Image` помещен в zip-файл. (Напомним, что Python умеет импортировать модули из zip-файлов: мы должны лишь поместить такой файл в один из каталогов, перечисленных в списке `sys.path`, а затем импортировать его так, будто это обычный модуль; см. `docs.python.org/dev/library/zipimport.html`.) Решить эту проблему можно, воспользовавшись стандартной функцией `pkgutil.walk_packages()` (вместо `os.listdir()`, с соответствующими изменениями в коде), т. к. эта функция может работать как с обычными пакетами, так и с пакетами в zip-файлах; она также справляется с пакетами, поставляемыми в виде написанных на C расширений и предкомпилированного байт-кода (`.pyc` и `.pyo`).

```
class Image:

   def __init__(self, width=None, height=None, filename=None,
```

```
            background=None, pixels=None):
    assert (width is not None and (height is not None or
            pixels is not None) or (filename is not None))
    if filename is not None: # из файла
       self.load(filename)
    elif pixels is not None: # из данных
       self.width = width
       self.height = len(pixels) // width
       self.filename = filename
       self.meta = {}
       self.pixels = pixels
    else: # пустое
       self.width = width
       self.height = height
       self.filename = filename
       self.meta = {}
       self.pixels = create_array(width, height, background)
```

У метода изображения `__init__()` довольно сложная сигнатура, но это неважно, потому что для создания изображений мы предлагаем пользователям гораздо более простые вспомогательные методы класса (например, `Image.Image.create()`, который мы уже видели).

```
@classmethod
def from_file(Class, filename):
   return Class(filename=filename)

@classmethod
def create(Class, width, height, background=None):
   return Class(width=width, height=height, background=background)

@classmethod
def from_data(Class, width, pixels):
   return Class(width=width, pixels=pixels)
```

Это три простых фабричных метода для создания изображений. Их можно вызывать от имени самого класса (например, `image = Image.Image.create(200, 400)`), и они будут правильно работать также для подклассов `Image`.

Метод `from_file()` создает изображение, читая его из указанного файла. Метод `create()` создает пустое изображение с заданным цветом фона (или с прозрачным фоном, если цвет не указан). Метод `from_data()` создает изображение заданной ширины с пикселями (то есть цветами), взятыми из одномерного массива (типа `array.array` или `numpy.ndarray`).

```
def create_array(width, height, background=None):
   if numpy is not None:
```

```
            if background is None:
               return numpy.zeros(width * height, dtype=numpy.uint32)
            else:
               iterable = (background for _ in range(width * height))
               return numpy.fromiter(iterable, numpy.uint32)
         else:
            typecode = "I" if array.array("I").itemsize >= 4 else "L"
            background = (background if background is not None else
                          ColorForName["transparent"])
         return array.array(typecode, [background] * width * height)
```

Эта функция создает одномерный массив 32-разрядных целых без знака (см. рис. 3.8). Если переменная `numpy` присутствует и фон прозрачный, то мы можем воспользоваться фабричной функцией `numpy.zeros()` для создания массива, в котором все элементы равны нулю (`0x00000000`). Любой пиксель с нулевой компонентой альфа полностью прозрачен. Если цвет фона задан, то мы создаем генераторное выражение, которое порождает `width` × `height` значений (одинаковых и равных `background`) и передает этот итерируемый объект фабричной функции `numpy.fromiter()`.

Если `numpy` отсутствует, то мы должны создать объект `array.array`. В отличие от `numpy`, этот модуль не позволяет задать точный размер хранящихся в нем целых чисел, поэтому мы делаем лучшее из возможного. Мы указываем спецификатор типа `"I"` (целое без знака длиной не менее двух байтов), если этот тип на самом деле занимает четыре или более байтов; в противном случае указываем спецификатор типа `"L"` (целое без знака длиной не менее четырех байтов). Тем самым мы используем целое минимального размера, способное хранить четыре байта, – даже на 64-разрядных машинах, где целое без знака обычно занимает восемь байтов. Затем мы создаем массив элементов указанного типа и записываем в него `width` × `height` значений, содержащих цвет фона. (Словарь по умолчанию `ColorForName` мы обсудим ниже на стр. 155).

```
class Error(Exception): pass
```

Этот класс дает нам тип исключения `Image.Error`. Можно было бы воспользоваться одним из встроенных типов исключений (например, `ValueError`), но таким образом пользователям класса `Image` будет проще перехватывать исключения, относящиеся к изображениям, не опасаясь перехватить что-то еще.

```
   def load(self, filename):
      module = Image._choose_module("can_load", filename)
```

```
        if module is not None:
            self.width = self.height = None
            self.meta = {}
            module.load(self, filename)
            self.filename = filename
        else:
            raise Error("no Image module can load files of type {}".format(
                        os.path.splitext(filename)[1]))
```

Модуль `Image.__init__.py` ничего не знает о форматах графических файлов. Но форматные модули об этом как раз знают, и ранее мы их загрузили в список `_Modules`. Форматные модули можно считать вариациями на тему паттерна Шаблонный метод (раздел 3.10) или паттерна Стратегия (раздел 3.9).

Здесь мы пытаемся найти модуль, способный загрузить файл с указанным именем. Если такой найдется, то мы инициализируем некоторые переменные экземпляра изображения и просим модуль загрузить файл. Если его метод `load()` завершается успешно, то он заполняет массив `self.pixels` значениями цветов и соответственно устанавливает `self.width` и `self.height`. В противном случае возбуждается исключение. (Мы увидим примеры метода `load()` для разных форматов в подразделах 3.12.2 и 3.12.3.)

```
    @staticmethod
    def _choose_module(actionName, filename):
        bestRating = 0
        bestModule = None
        for module in _Modules:
            action = getattr(module, actionName, None)
            if action is not None:
                rating = action(filename)
                if rating > bestRating:
                    bestRating = rating
                    bestModule = module
        return bestModule
```

Этот статический метод нужен для поиска в закрытом списке `_Modules` модуля, который умеет выполнять действие (`actionName`) над файлом (`filename`). Он перебирает все загруженные модули и для каждого пытается найти функцию с именем `actionName` (например, `can_load()` или `can_save()`) с помощью встроенной функции `getattr()`. Найдя функцию, метод вызывает ее с указанным именем файла.

Ожидается, что функция действия вернет 0, если вообще не может выполнить действие, 100 – если может выполнить его в полном объ-

еме, и промежуточное целое число, если может выполнить действие, но не полностью. Например, модуль `Image/Xbm.py` возвращает 100 для файлов с расширением `.xbm`, поскольку он поддерживает этот формат в полном объеме, а для всех остальных расширений возвращает 0. С другой стороны, модуль `Image/Xpm.py` возвращает лишь 80 для `xpm`-файлов, потому что поддерживает спецификацию XPM не целиком (хотя отлично работал на всех `xpm`-файлах, на которых тестировался).

В конце возвращается модуль с наивысшим рейтингом или `None`, если не найдено ни одного подходящего модуля.

```
def save(self, filename=None):
   filename = filename if filename is not None else self.filename
   if not filename:
      raise Error("can't save without a filename")
   module = Image._choose_module("can_save", filename)
   if module is not None:
      module.save(self, filename)
      self.filename = filename
   else:
      raise Error("no Image module can save files of type {}".format(
                  os.path.splitext(filename)[1]))
```

Этот метод очень похож на метод `load()` тем, что пытается найти модуль, способный сохранить файл с указанным именем (то есть в формате, определяемом расширением), и выполняет сохранение.

```
def pixel(self, x, y):
   return self.pixels[(y * self.width) + x]
```

Метод `pixel()` возвращает цвет в заданной позиции в виде значения ARGB (32-разрядного целого без знака).

```
def set_pixel(self, x, y, color):
   self.pixels[(y * self.width) + x] = color
```

Метод `set_pixel()` записывает в указанную позицию заданное значение ARGB, если координаты `x` находятся в допустимом диапазоне; в противном случае он возбуждает исключение `IndexError`.

Модуль `Image` предоставляет основные методы рисования, в том числе `line()`, `ellipse()` и `rectangle()`. Мы приведем лишь один метод в качестве типичного представителя.

```
def line(self, x0, y0, x1, y1, color):
   Δx = abs(x1 - x0)
   Δy = abs(y1 - y0)
```

```
    xInc = 1 if x0 < x1 else -1
    yInc = 1 if y0 < y1 else -1
    δ = Δx - Δy

    while True:
       self.set_pixel(x0, y0, color)
       if x0 == x1 and y0 == y1:
          break
       δ2 = 2 * δ
       if δ2 > -Δy:
          δ -= Δy
          x0 += xInc
       if δ2 < Δx:
          δ += Δx
          y0 += yInc
```

В этом методе алгоритм построения прямой Брезенхэма (для которого нужна только целочисленная арифметика) используется для рисования отрезка прямой, соединяющего точки (`x0`, `y0`) и (`x1`, `y1`)[7]. Благодаря поддержке Unicode в Python 3 мы можем использовать естественные в этом контексте имена переменных; например, *Δx* и *Δy* для представления приращений координат *x* и *y*, и *δ* и *δ2* – для погрешностей.

```
def scale(self, ratio):
   assert 0 < ratio < 1
   rows = round(self.height * ratio)
   columns = round(self.width * ratio)
   pixels = create_array(columns, rows)
   yStep = self.height / rows
   xStep = self.width / columns
   index = 0
   for row in range(rows):
      y0 = round(row * yStep)
      y1 = round(y0 + yStep)
      for column in range(columns):
         x0 = round(column * xStep)
         x1 = round(x0 + xStep)
         pixels[index] = self._mean(x0, y0, x1, y1)
         index += 1
   return self.from_data(columns, pixels)
```

Этот метод создает и возвращает новое изображение, являющееся уменьшенной версией данного. Коэффициент подобия может находиться в диапазоне (0.0, 1.0); если он равен 0,75, то получается

[7] Этот алгоритм описан на странице en.wikipedia.org/wiki/Bresenham's_line_algorithm.

изображение, ширина и высота которого составляют ¾ от исходных, а если 0,5 – то ¼ от исходных. Каждый пиксель (то есть цвет) результирующего изображения получается путем усреднения цветов в прямоугольнике исходного изображения, который данный пиксель представляет.

Координаты *x*, *y* в изображении целые, но чтобы избежать неточности, мы должны использовать арифметику с плавающей точкой (то есть оператор `/`, а не `//`) при работе с данными пикселей. Поэтому мы применяем встроенную функцию `round()`, когда нужно получить целые числа. В самом конце мы вызываем вспомогательный фабричный метод класса `Image.Image.from_data()`, который создает новое изображение, зная вычисленное число столбцов и созданный и заполненный нами массив пикселей.

```
def _mean(self, x0, y0, x1, y1):
   αTotal, redTotal, greenTotal, blueTotal, count = 0, 0, 0, 0, 0
   for y in range(y0, y1):
      if y >= self.height:
         break
      offset = y * self.width
      for x in range(x0, x1):
         if x >= self.width:
            break
         α, r, g, b = self.argb_for_color(self.pixels[offset + x])
         αTotal += α
         redTotal += r
         greenTotal += g
         blueTotal += b
         count += 1
   α = round(αTotal / count)
   r = round(redTotal / count)
   g = round(greenTotal / count)
   b = round(blueTotal / count)
   return self.color_for_argb(α, r, g, b)
```

Этот закрытый метод вычисляет суммы четырех компонентов всех пикселей в прямоугольнике, заданном координатами противоположных углов `x0`, `y0`, `x1`, `y1`. Затем каждая сумма делится на число пикселей, участвовавших в порождении цвета, то есть мы получаем среднее арифметическое. Весь процесс показан на рис. 3.9.

```
MAX_ARGB = 0xFFFFFFFF
MAX_COMPONENT = 0xFF
```

Минимальное 32-разрядное значение ARGB равно `0x0` (то есть `0x00000000` – прозрачный, строго говоря, прозрачный черный цвет).

Эти две константы в модуле `Image` определяют максимальное значение ARGB (полностью непрозрачный белый) и максимальное значение любой компоненты цвета (255).

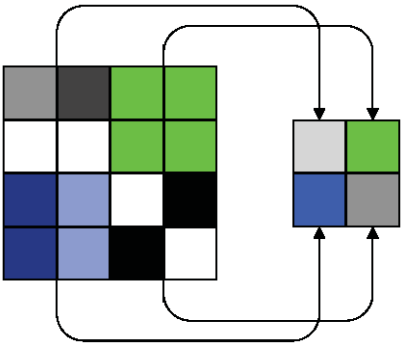

Рис. 3.9. Масштабирование изображения размером 4 × 4 с коэффициентом 0,5

```
@staticmethod
def argb_for_color(color):
   if numpy is not None:
      if isinstance(color, numpy.uint32):
         color = int(color)
   if isinstance(color, str):
      color = color_for_name(color)
   elif not isinstance(color, int) or not (0 <= color <= MAX_ARGB):
      raise Error("invalid color {}".format(color))
   α = (color >> 24) & MAX_COMPONENT
   r = (color >> 16) & MAX_COMPONENT
   g = (color >> 8) & MAX_COMPONENT
   b = (color & MAX_COMPONENT)
   return α, r, g, b
```

Этот статический метод (и функция модуля) возвращает четыре компоненты заданного цвета (каждая в диапазоне от 0 до 255). Цвет может быть передан в виде `int, numpy.uint32` или строкового названия. Далее выделяются отдельные компоненты – байты целого числа, представляющего цвет, для чего применяются операции поразрядного сдвига (`>>`) и поразрядного И (`&`).

```
@staticmethod
def color_for_name(name):
   if name is None:
      return ColorForName["transparent"]
   if name.startswith("#"):
```

```
            name = name[1:]
            if len(name) == 3: # добавим непрозрачный альфа-канал
               name = "F" + name # теперь 4 16-ричных цифры
            if len(name) == 6: # добавим непрозрачный альфа-канал
               name = "FF" + name # теперь все 8 16-ричных цифр
            if len(name) == 4: # было #FFF или #FFFF
               components = []
               for h in name:
                  components.extend([h, h])
               name = "".join(components) # теперь все 8 16-ричных цифр
            return int(name, 16)
         return ColorForName[name.lower()]
```

Этот статический метод (и функция модуля) возвращает 32-разрядное значение ARGB, соответствующее строковому названию цвета. Если вместо цвета передано `None`, то метод возвращает прозрачный цвет. Если строка начинается знаком `#`, предполагается, что это цвет в стиле HTML вида `"#HHH"`, `"#HHHH"`, `"#HHHHHH"` или `"#HHHHHHHH"`, где `H` – шестнадцатиричная цифра. Если цифр хватает только для RGB-значения, мы добавляем в начало две цифры `"F"`, чтобы получить непрозрачный альфа-канал. В противном случае возвращается цвет из словаря `ColorForName`; эта операция всегда завершается успешно, потому что `ColorForName` – словарь по умолчанию.

```
ColorForName = collections.defaultdict(lambda: 0xFF000000, {
   "transparent": 0x00000000, "aliceblue": 0xFFF0F8FF,
   ...
   "yellow4": 0xFF8B8B00, "yellowgreen": 0xFF9ACD32})
```

`ColorForName` – это словарь `collections.defaultdict`, который возвращает 32-разрядное целое без знака, соответствующее заданному именованному цвету. Если же в словаре нет указанного имени, то возвращается значение `0xFF000000`, соответствующее непрозрачному черному цвету. Пользователи модуля `Image` вправе использовать этот словарь непосредственно, но функция `color_for_name()` удобнее и гибче. Имена цветов взяты из файла `rgb.txt`, поставляемого вместе с X11, и добавлен прозрачный цвет (transparent).

Функция `collections.defaultdict()` принимает в качестве первого аргумента фабричную функцию, а за ней могут следовать любые аргументы, принимаемые простым словарем. Фабричная функция служит для того, чтобы породить значение элемента, который автоматически создается при попытке обратиться к несуществующему ключу. Здесь мы воспользовались лямбда-функцией, которая всегда возвращает одно и то же значение (непрозрачный черный). Хотя передавать

ключевые аргументы можно (например, `transparent=0x00000000`), в данном случае цветов больше 255, а это максимальное число аргументов в Python, поэтому мы инициализируем словарь по умолчанию с помощью обычного словаря, применяя синтаксис `{key: value}`, на который это ограничение не распространяется.

```
argb_for_color = Image.argb_for_color
rgb_for_color = Image.rgb_for_color
color_for_argb = Image.color_for_argb
color_for_rgb = Image.color_for_rgb
color_for_name = Image.color_for_name
```

После класса `Image` мы написали несколько вспомогательных функций, основанных на статических методах класса. Это означает, например, что, сделав `import Image`, пользователь может вызывать функцию `Image.color_for_name()`, а, если у него имеется экземпляр класса `Image.Image`, то также и *`image`*`.color_for_name()`.

На этом мы завершаем обзор ядра модуля `Image` (в файле `Image/__init__.py`). Мы опустили несколько не столь интересных констант, кое-какие методы `Image.Image` (`rectangle()`, `ellipse()` и `subsample()`), свойство `size` (которое просто возвращает кортеж из двух элементов: ширина и высота) и различные статические методы для манипулирования цветами. Этого модуля достаточно для создания, загрузки, модификации и сохранения графических файлов в форматах XBM и XPM, а, если установлен модуль PyPNG, то и в формате PNG.

Теперь рассмотрим два модуля для конкретных графических форматов, от которых модуль `Image` зависит. Но мы не станем останавливаться на модуле `Image/Xbm.py`, потому что, если не считать низкоуровневых деталей работы с форматом XBM, он не научит нас ничему такому, чего нет в модуле `Image/Xpm.py`, к изучению которого мы и приступаем.

3.12.2. Обзор модуля Xpm

Все форматные модули должны предоставлять четыре функции. Две из них – `can_load()` и `can_save()`. Та и другая должна возвращать 0, если не может выполнить операцию, 100 – если может, и промежуточное значение – если может, но не в полном объеме. Ожидается также, что модуль предоставляет функции `load()` и `save()`, которые вправе предполагать, что будут вызваны только для файла, для которого `can_load()` или `can_save()` вернула ненулевое значение.

```
def can_load(filename):
   return 80 if os.path.splitext(filename)[1].lower() == ".xpm" else 0

def can_save(filename):
   return can_load(filename)
```

Модуль `Image/Xpm.py` реализует большую часть спецификации XPM, оставив за бортом только некоторые редко используемые возможности. Поэтому он возвращает значение 80 (могу, но не идеально) как для загрузки, так и для сохранения[8]. Это означает, что если будет добавлен новый модуль обработки XPM – скажем, `Image/Xpm2.py` – который возвращает значение больше 80, то он и будет использоваться вместо этого модуля (мы уже обсуждали этот вопрос при рассмотрении метода `Image._choose_module()` на стр. 150).

```
(_WANT_XPM, _WANT_NAME, _WANT_VALUES, _WANT_COLOR, _WANT_PIXELS, _DONE) =
     ("WANT_XPM", "WANT_NAME", "WANT_VALUES", "WANT_COLOR",
      "WANT_PIXELS", "DONE")
_CODES = "".join((chr(x) for x in range(32, 127) if chr(x) not in '\\"'))
```

XPM – это простой текстовый формат (в 7-разрядной кодировке ASCII), который необходимо разобрать для извлечения данных. В формате есть кое-какие метаданные (ширина, высота, количество цветов и т. д.), таблица цветов и данные пикселей, причем каждый пиксель представлен ссылкой на таблицу цветов. Детали завели бы нас слишком далеко в сторону от Python, тем не менее, мы воспользуемся простым, написанным вручную анализатором, а константы выше представляют его состояния.

```
def load(image, filename):
   colors = cpp = count = None
   state = _WANT_XPM
   palette = {}
   index = 0
   with open(filename, "rt", encoding="ascii") as file:
      for lino, line in enumerate(file, start=1):
         line = line.strip()
         ...
```

Это начало функции модуля `load()`. Параметр `image` – объект типа `Image.Image`, и внутри функции (не показанной) непосредственно ус-

[8] Вместо того чтобы проверять тип файла по расширению имени, можно было бы прочесть первые несколько байтов – «сигнатуру» файла. Например, XPM-файлы начинаются байтами `0x2F 0x2A 0x20 0x58 0x50 0x4D 0x20 0x2A 0x2F` (`"/* XPM */"`), а PNG-файлы – байтами `0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A` (`"·PNG····"`).

танавливаются атрибуты изображения `pixels`, `width` и `height`. Массив пикселей создается с помощью функции `Image.create_array()`, поэтому модуль `Xpm.py` ничего не знает о том, что это за массив – `array.array` или `numpy.ndarray` – и ему это безразлично, лишь бы массив был одномерным и имел длину `width × height`. Правда, это значит, что обращаться к пикселям массива можно только с помощью методов, общих для обоих типов.

```
def save(image, filename):
   name = Image.sanitized_name(filename)
   palette, cpp = _palette_and_cpp(image.pixels)
   with open(filename, "w+t", encoding="ascii") as file:
      _write_header(image, file, name, cpp, len(palette))
      _write_palette(file, palette)
      _write_pixels(image, file, palette)
```

В форматах XBM и XPM присутствует имя, основанное на имени файла, но являющееся допустимым идентификатором в смысле языка C. Это имя возвращает функция `Image.sanitized_name()`. Почти вся работа по сохранению перепоручается закрытым вспомогательным функциям, которые не представляют особого интереса и потому не показаны.

```
def sanitized_name(name):
   name = re.sub(r"\W+", "", os.path.basename(os.path.
splitext(name)[0]))
   if not name or name[0].isdigit():
      name = "z" + name
   return name
```

Функция `Image.sanitized_name()` принимает имя файла и образует из него имя, которое включает только латинские буквы без диакритических знаков, цифры и знаки подчеркивания, а начинается с буквы или знака подчеркивания. В регулярном выражении часть `\W+` сопоставляется с одним или несколькими символами, не являющимися частью слова (которые не могут встречаться в допустимых идентификаторах C).

Для добавления в модель `Image` поддержки любого другого формата нужно создать подходящий модуль в каталоге `Image`, реализовав в нем четыре обязательные функции: `can_load()`, `can_save()`, `load()` и `save()`, причем первые две должны возвращать правильные целые числа в зависимости от переданного имени файла. Очень популярен формат PNG, но он довольно сложен. К счастью, мы можем с минимальными усилиями адаптировать уже имеющийся модуль PyPNG, чем и займемся в следующем подразделе.

3.12.3. Модуль-обертка PNG

Модуль PyPNG (`github.com/drj11/pypng`) отлично поддерживает графический формат PNG. Однако его интерфейс отличается от того, что модуль `Image` требует от форматных модулей. Поэтому мы сейчас создадим модуль `Image/Png.py`, в котором воспользуемся паттерном Адаптер (см. раздел 2.1), чтобы добавить поддержку формата PNG в модуль `Image`. В отличие от предыдущего подраздела, где была показана лишь малая часть кода, модуль `Image/Png.py` будет приведен целиком.

```
try:
   import png
except ImportError:
   png = None
```

Для начала мы пытаемся импортировать модуль PyPNG, который называется `png`. В случае ошибки мы создаем переменную `png` и записываем в нее значение `None`, которое можно будет проверить позже.

```
def can_load(filename):
   return (80 if png is not None and
           os.path.splitext(filename)[1].lower() == ".png" else 0)

def can_save(filename):
   return can_load(filename)
```

Если модуль `png` успешно импортирован, то мы возвращаем значение 80 (не вполне идеально), показывая, что поддерживаем формат PNG. Значение 80, а не 100 выбрано для того, чтобы какой-нибудь другой модуль мог заместить этот. Как и в случае XPM, мы возвращаем одинаковый рейтинг для сохранения и загрузки, хотя никто не мешает указать разные.

```
def load(image, filename):
   reader = png.Reader(filename=filename)
   image.width, image.height, pixels, _ = reader.asRGBA8()
   image.pixels = Image.create_array(image.width, image.height)
   index = 0
   for row in pixels:
      for r, g, b, α in zip(row[::4], row[1::4], row[2::4], row[3::4]):
         image.pixels[index] = Image.color_for_argb(α, r, g, b)
         index += 1
```

Сначала мы создаем объект `png.Reader`, указывая то имя файла, которое сами получили; в результате PNG-файл будет загружен в

экземпляр `reader`. Затем мы извлекаем ширину, высоту и пиксели изображения, а метаданные пропускаем.

В модуле PyPNG используется формат RGBA, тогда как в нашем модуле `Image` – формат ARGB, эту разницу необходимо учесть. Делается это путем извлечения пикселей с помощью метода `png.Reader.asRGBA8()`, который возвращает двумерный массив строк, содержащих значения компонент цвета. Например, если в начале первой строки изображения находится непрозрачный красный пиксель, за которым следует непрозрачный синий, то список значений для первой строки будет начинаться так: `0xFF`, `0x00`, `0x00`, `0xFF`, `0x00`, `0x00`, `0xFF`, `0xFF`.

Получив пиксели в формате RGBA, мы создаем новый массив нужного размера, заполняя его прозрачными пикселями. После этого мы обходим все строки компонент цвета и вырезаем каждую компоненту. Красные компоненты находятся в позициях строки с индексами 0, 4, 8, 12, ..., зеленые – в позициях 1, 5, 9, 13, ..., синие – в позициях 2, 6, 10, 14, ..., альфа – в позициях 3, 7, 11, 15, ... Затем мы пользуемся встроенной функцией `zip()` для получения 4-кортежей, содержащих компоненты цвета. Первый 4-кортеж содержит элементы первой строки в позициях (0, 1, 2, 3), второй – элементы в позициях (4, 5, 6, 7) и т. д. Для каждого кортежа мы создаем значение цвета в формате ARGB и вставляем его в наш одномерный массив пикселей, представляющий изображение.

```
def save(image, filename):
   with open(filename, "wb") as file:
      writer = png.Writer(width=image.width, height=image.height,
                          alpha=True)
      writer.write_array(file, list(_rgba_for_pixels(image.pixels)))
```

Функция `save()` делегирует большую часть работы модулю `png`. Сначала она создает объект `png.Writer` с правильными метаданными, а затем записывает в него все пиксели. Поскольку в модуле `Image` используется формат ARGB, а в модуле `png` – формат RGBA, то мы воспользовались закрытой вспомогательной функцией для преобразования одного в другой.

```
def _rgba_for_pixels(pixels):
   for color in pixels:
      a, r, g, b = Image.argb_for_color(color)
      for component in (r, g, b, a):
         yield component
```

Эта функция обходит переданный ей массив (то есть *image*.pixels) и выделяет цветовые компоненты каждого пикселя. Затем она уступает каждую компоненту (в порядке RGBA) вызывающей стороне.

И это весь код модуля-обертки, потому что трудная работа делается модулем PyPNG png.

Модуль Image предлагает полезный интерфейс для рисования (set_pixel(), line(), rectangle(), ellipse()) и поддерживает загрузку и сохранение файлов в форматах XBM, XPM и (если установлен модуль PyPNG) PNG. В нем имеются также методы subsample() (для быстрого грубого масштабирования) и scale() (для плавного масштабирования) и ряд вспомогательных функций и статических методов для манипулирования цветами.

Модуль Image можно использовать в параллельных программах – например, чтобы создавать, загружать, модифицировать и сохранять изображения в нескольких потоках или процессах. Поэтому он удобнее, чем, скажем, Tkinter, который умеет обрабатывать изображения только в главном потоке пользовательского интерфейса. К сожалению, масштабирование производится довольно медленно. Один из способов ускорить масштабирование – при условии, что имеется многоядерный процессор и несколько изображений, которые можно масштабировать одновременно, – распараллелить программу, как мы покажем в следующей главе. Однако масштабирование – счетная задача, поэтому можно рассчитывать только на ускорение, пропорциональное количеству ядер; так, на 4-ядерной машине нельзя получить больше чем 4-кратное ускорение. О том, как добиться большего с помощью Cython, мы расскажем в главе 5.

ГЛАВА 4. Высокоуровневый параллелизм в Python

Интерес к параллельному программированию резко усилился в начале нового тысячелетия. Этому немало способствовали язык Java, который ввел параллелизм в обиход, повсеместное распространение многоядерных машин и наличие поддержки параллельного программирования в большинстве современных языков.

Писать и сопровождать параллельные программы труднее (иногда *намного* труднее), чем последовательные. Более того, бывает так, что производительность параллельных программ хуже (иногда *намного* хуже), чем эквивалентных последовательных. И тем не менее, если делать все правильно, то можно написать параллельную программу, которая будет настолько быстрее последовательной, что все дополнительные усилия окупятся с лихвой.

Большинство современных языков (включая C++ и Java) поддерживают параллелизм на уровне языка и предоставляют дополнительную высокоуровневую функциональность в стандартных библиотеках. Параллелизм можно реализовывать по-разному, самое важное различие состоит в том, как осуществляется доступ к разделяемым данным: прямо (например, с помощью разделяемой памяти) или косвенно (например, с помощью межпроцессных коммуникаций – IPC). Под многопоточным параллелизмом понимается одновременное выполнение нескольких потоков в одном процессе операционной системы. Обычно потоки обращаются к разделяемым данным путем сериализации доступа к общей памяти, причем отвечает за сериализацию программист, который должен применить тот или иной механизм блокировки. Параллелизм на уровне процессов (мультипроцессирование) имеет место, когда есть отдельные независимо исполняющиеся процессы. Параллельные процессы обычно обращаются к разделяемым данным с помощью IPC, хотя могут пользоваться разделяемой

памятью, если язык или библиотека ее поддерживают. Еще один вид параллелизма основан на «параллельном ожидании», а не на параллельном выполнении; этот подход применяется в реализациях асинхронного ввода-вывода.

В Python имеется низкоуровневая поддержка асинхронного ввода-вывода (модули `asyncore` и `asynchat`). Высокоуровневая поддержка предоставляется сторонней библиотекой Twisted (`twistedmatrix.com`). Включение поддержки высокоуровневого асинхронного ввода-вывода – в том числе циклов обработки событий – в стандартную библиотеку запланировано в версии Python 3.4 (`www.python.org/dev/peps/pep-3156`).

Если говорить о более традиционных многопоточном и многопроцессном параллелизме, то Python поддерживает и то, и другое. Поддержка потоков в Python реализована в привычном русле, но уровень поддержки мультипроцессирования гораздо выше, чем во многих других языках и библиотеках. К тому же, в многопроцессном и многопоточном параллелизме используются одни и те же абстракции, так что перейти от одного подхода к другому очень просто, по крайней мере, если не используется разделяемая память.

Из-за глобальной блокировки интерпретатора (GIL) сам интерпретатор Python может в каждый момент времени исполняться только одним процессорным ядром[1]. Код, написанный на C, может захватывать и освобождать GIL, поэтому такого ограничения для него не существует, а значительная часть Python – и его стандартной библиотеки – написана на C. Но все равно это означает, что, реализуя параллелизм с помощью потоков, мы не получим ускорения, на которое рассчитывали.

Вообще говоря, в задачах с большим объемом вычислений (счетных задачах) многопоточность легко может привести к падению производительности по сравнению с последовательной программой. Одно из решений – писать код на Cython (раздел 5.2). В варианте Python, с дополнительными синтаксическими конструкциями, транслирующимися на чистый C, это может дать ускорение в 100 раз – гораздо больше, чем можно достичь с помощью любого вида параллелизма, где повышение производительности пропорционально количеству процессорных ядер. Однако если вообще имеет смысл заниматься распараллеливанием, то для счетных задач лучше вовсе избегать GIL,

[1] Это ограничение не распространяется на Jython и некоторые другие интерпретаторы Python. Ни в одном примере из этой книги не делается предположений о наличии или отсутствии GIL.

применяя модуль `multiprocessing`. В этом случае используются не потоки в одном процессе (конкурирующие за GIL), а отдельные процессы, в каждом из которых есть свой экземпляр интерпретатора Python, вследствие чего конкуренция отсутствует.

Для задач, ограниченных скоростью ввода-вывода (например, сетевых), распараллеливание может принести огромный эффект. В этих случаях сетевые задержки зачастую настолько доминируют, что безразлично, какой параллелизм использовать – многопоточный или многопроцессный.

Мы рекомендуем всегда, когда есть возможность, начинать с написания последовательной программы. Писать такие программы проще и быстрее, чем параллельные, да и тестировать их легче. После того как последовательная программа отлажена, может оказаться, что ее быстродействия вполне достаточно. А если нет, то мы сможем воспользоваться ей для сравнения с параллельной версией как с точки зрения результатов (то есть корректности), так и с точки зрения производительности. Если же говорить о типе параллелизма, то мы рекомендуем многопроцессный для счетных задач и многопроцессный или многопоточный для задач, ограниченных скоростью ввода-вывода. Но важен не только тип параллелизма, но и его уровень.

В этой книге мы определим три уровня параллелизма.

- **Параллелизм низкого уровня.** Это параллелизм, в котором используются атомарные операции. Он рассчитан скорее на авторов библиотек, чем на разработчиков приложений, поскольку сделать ошибку очень легко, а отлаживать ее крайне сложно. Python не поддерживает такой вид параллелизма, хотя сами реализации параллелизма в Python обычно построены на основе низкоуровневых операций.
- **Параллелизм среднего уровня.** Это параллелизм, в котором атомарные операции явно не используются, зато используются явные блокировки. Такой уровень параллелизма поддерживают многие языки. В Python поддержку на этом уровне обеспечивают такие классы, как `threading.Semaphore`, `threading.Lock` и `multiprocessing.Lock`. На этом уровне обычно пишут разработчики приложений, поскольку зачастую ничего другого просто нет.
- **Параллелизм высокого уровня.** Это параллелизм, в котором нет ни атомарных операций, ни явных блокировок (они, конечно, есть, но спрятаны глубоко внутри, так что нас не интересуют). Некоторые современные языки начинают поддержи-

вать высокоуровневый параллелизм. В Python для этой цели имеется модуль `concurrent.futures` (Python 3.2), а также классы очередей `queue.Queue` и `multiprocessing`.

Писать на среднем уровне довольно просто, но легко допустить ошибку. Есть подходы, особенно уязвимые для тонких, с трудом отлавливаемых ошибок, а равно и очевидных любому крахов и зависаний, причем без какой-то легко прослеживаемой закономерности.

Ключевая проблема – разделение данных. Изменяемые общие данные необходимо защищать блокировками, которые гарантируют, что доступ производится строго последовательно (то есть в каждый момент времени к разделяемым данным может обращаться только один поток или процесс). Когда несколько потоков или процессов пытаются обратиться к одному и тому же элементу разделяемых данных, все они, кроме одного, блокируются (то есть простаивают). Это означает, что пока блокировка действует, в нашем приложении работает только один поток или процесс (как если бы она была последовательной), а остальные ждут. Поэтому следует ставить блокировки как можно реже и на как можно меньший срок. Самое простое решение – вообще не делать изменяемые данные разделяемыми. Тогда явные блокировки не понадобятся, и большая часть проблем параллелизма отпадет сама собой.

Конечно, иногда несколько параллельных потоков или процессов нуждаются в доступе к одним и тем же данным, но проблему можно решить и без явных блокировок. Одно из возможных решений – использовать структуру данных, поддерживающую параллельный доступ. В модуле `queue` есть несколько потокобезопасных очередей, а для многопроцессного параллелизма имеются классы `multiprocessing.JoinableQueue` и `multiprocessing.Queue`. Такие очереди можно использовать для организации единственного источника задач для всех параллельных потоков или процессов или единственного места сбора результатов, оставляя заботы о синхронизации доступа самой структуре данных.

Если имеются данные, для параллельной обработки которых очереди с поддержкой многопоточного доступа недостаточно, то лучший способ решить задачу без блокировки – передавать неизменяемые данные (например, числа или строки) или изменяемые, но тогда только читать их. Если необходимо использовать изменяемые данные, то безопаснее всего сделать их глубокую копию. Глубокое копирование позволяет избежать проблем, присущих блокировкам, правда, ценой повышенного потребления памяти и процессорного времени. Мож-

но вместо этого работать с типами данных, которые поддерживают параллельный доступ, например, `multiprocessing.Value` для одного изменяемого значения или `multiprocessing.Array` для массива изменяемых значений – при условии, что они созданы объектом `multiprocessing.Manager`, с которым мы познакомимся ниже.

В первых двух разделах этой главы мы исследуем параллелизм на примере двух приложений: одного, ограниченного быстродействием процессора, и другого, ограниченного скоростью вывода-вывода. В обоих случаях мы будем пользоваться высокоуровневыми средствами Python для распараллеливания – давно зарекомендовавшими себя потокобезопасными очередями и новым (Python 3.2) модулем `concurrent.futures`. В третьем разделе мы рассмотрим большой пример, показывающий, как с помощью параллельной обработки организовать отзывчивый графический интерфейс пользователя (ГИП), который сообщает о ходе продвижения операции и допускает ее отмену.

4.1. Распараллеливание задач с большим объемом вычислений

В примере класса `Image` из главы 3 (раздел 3.12) мы продемонстрировали код плавного масштабирования и отметили, что он работает довольно медленно. Но представим, что требуется масштабировать сразу много изображений и хочется сделать это максимально быстро, воспользовавшись преимуществами нескольких ядер.

Масштабирование изображений – счетная задача, поэтому можно ожидать, что мультипроцессирование даст наилучшую производительность. Эту гипотезу подкрепляют результаты хронометража в табл. 4.1[2]. (В примере из главы 5 мы рассмотрим мультипроцессирование в сочетании с Cython, что даст куда более внушительное ускорение, см. раздел 5.3).

Таблица 4.1. Сравнение скорости масштабирования изображений

Программа	Степень параллелизма	Секунд	Ускорение
`imagescale-s.py`	Нет	784	Эталон
`imagescale-c.py`	4 сопрограммы	781	1,00×

[2] Хронометраж проводился на слабо загруженной машине с 4-ядерным процессором AMD64 3 Ггц в ходе обработки 56 изображений размером от 1 до 12 МиБ и совокупным размером 316 МиБ. Совокупный размер масштабированных изображений составил 67 МиБ.

Таблица 4.1. (окончание)

Программа	Степень параллелизма	Секунд	Ускорение
`imagescale-t.py`	Пул из 4 потоков	1339	0,59×
`imagescale-q-m.py`	4 процесса с очередью	206	3,81×
`imagescale-m.py`	Пул из 4 процессов	201	3,90×

Результат для программы `imagescale-t.py` с четырьмя потоками не оставляет сомнений в том, что использование многопоточности для решения счетных задач только уменьшает производительность по сравнению с последовательной программой. Объясняется это тем, что Python задействует для обработки только одно ядро, так что в дополнение к масштабированию интерпретатор еще несет весьма заметные расходы на контекстное переключение между четырьмя потоками. Сравните с многопроцессными версиями, которые распределяют ту же работу по всем процессорным ядрам. Разница между многопроцессной очередью и пулом процессов невелика, и обе версии дают такое ускорение, которого мы и ожидали (прямо пропорционально количеству ядер)[3].

Все программы масштабирования изображений принимают в командной строке аргументы, которые разбираются с помощью модуля `argparse`. В состав аргументов входит максимальный размер масштабированного изображения, признак плавности масштабирования (мы хронометрировали именно такой режим), а также исходный и конечный каталоги. Если размер исходного изображения меньше указанного, то оно не масштабируется, а просто копируется; все изображения, для которых проводился хронометраж, масштабировались. В случае параллельных версий можно задать также степень параллелизма (сколько использовать потоков или процессов); это в основном для отладки и хронометража. Для счетных программ обычно задается столько потоков или процессов, сколько есть ядер. Для программ, ограниченных скоростью ввода-вывода, мы задаем число, кратное количеству ядер (2×, 3×, 4× или больше), – в зависимости от полосы пропускания сети. Для полноты картины ниже приведена функция `handle_commandline()`, применяемая в параллельных программах масштабирования.

[3] В Windows запуск новых процессов обходится гораздо дороже, чем в большинстве других операционных систем. По счастью, для организации очередей и пулов в Python используются постоянные пулы процессов, чтобы исключить накладные расходы на создание все новых и новых процессов.

```
def handle_commandline():
   parser = argparse.ArgumentParser()
   parser.add_argument("-c", "--concurrency", type=int,
          default=multiprocessing.cpu_count(),
          help="specify the concurrency (for debugging and "
             "timing) [default: %(default)d]")
   parser.add_argument("-s", "--size", default=400, type=int,
          help="make a scaled image that fits the given dimension "
             "[default: %(default)d]")
   parser.add_argument("-S", "--smooth", action="store_true",
          help="use smooth scaling (slow but good for text)")
   parser.add_argument("source",
          help="the directory containing the original .xpm images")
   parser.add_argument("target",
          help="the directory for the scaled .xpm images")
   args = parser.parse_args()
   source = os.path.abspath(args.source)
   target = os.path.abspath(args.target)
   if source == target:
      args.error("source and target must be different")
   if not os.path.exists(args.target):
      os.makedirs(target)
   return args.size, args.smooth, source, target, args.concurrency
```

Обычно мы не позволяем пользователям задавать уровень параллелизма, но для отладки, хронометража и тестирования это полезно, поэтому мы его включили. Функция `multiprocessing.cpu_count()` возвращает количество имеющихся процессорных ядер (например, 2 для машины с двухъядерным процессором и 8 для машины с двумя 4-ядерными процессорами).

В модуле `argparse` принят декларативный подход к созданию анализатора командной строки. Создав анализатор, мы разбираем командную строку и выделяем из нее аргументы. Производится ряд простых проверок (например, мы не разрешаем записывать масштабированные изображения поверх исходных), после чего создается конечный каталог, если его еще не существует. Функция `os.makedirs()` аналогична функции `os.mkdir()`, только умеет создавать все промежуточные каталоги на пути к конечному.

Прежде чем вплотную заняться кодом, отметим несколько важных правил, применимых к любому Python-файлу, в котором используется модуль `multiprocessing`.

- Файл должен быть допускающим импорт модулем. Например, `my-mod.py` – допустимое имя для *программы* на Python, но не для модуля (т. к. `import my-mod` приводит к синтаксической ошибке); имена *`my_mod.py`* и *`MyMod.py`* допустимы.

- В файле должна быть функция, являющаяся точкой входа (например, `main()`), и заканчиваться он должен обращением к точке входа, например: `if __name__ == "__main__": main()`.
- В Windows Python-файл и интерпретатор Python (`python.exe` или `pythonw.exe`) должны находиться на одном и том же диске (например, `C:`).

Далее мы рассмотрим две многопроцессных версии программы масштабирования изображений: `imagescale-q-m.py` и `imagescale-m.py`. Обе программы сообщают о ходе продвижения (печатают имя текущего обрабатываемого файла) и поддерживают отмену (например, в результате нажатия пользователем `Ctrl+C`).

4.1.1. Очереди и многопроцессная обработка

Программа `imagescale-q-m.py` создает очередь подлежащих выполнению задач (то есть ожидающих масштабирования изображений) и очередь результатов.

```
Result = collections.namedtuple("Result", "copied scaled name")
Summary = collections.namedtuple("Summary", "todo copied scaled canceled")
```

Именованный кортеж `Result` служит для хранения одного результата, а именно счетчиков скопированных и масштабированных изображений (всегда 1,0 или 0,1) и имени масштабированного изображения. Именованный кортеж `Summary` используется для хранения сводки результатов.

```
def main():
    size, smooth, source, target, concurrency = handle_commandline()
    Qtrac.report("starting...")
    summary = scale(size, smooth, source, target, concurrency)
    summarize(summary, concurrency)
```

Функция `main()` одинакова для всех программ масштабирования. Сначала она разбирает командную строку с помощью рассмотренной выше функции `handle_commandline()`, которая возвращает максимальный размер масштабированного изображения, булево значение, показывающее, надо ли масштабировать плавно, исходный каталог, откуда читаются файлы, и конечный каталог, куда записываются результаты. Для параллельных версий возвращается также, сколько по-

токов или процессов использовать (по умолчанию это число равно количеству ядер).

Программа сообщает пользователю, что начала работать, а затем вызывает функцию `scale()`, которая все и делает. Когда `scale()` вернет сводку результатов, мы напечатаем их с помощью функции `summarize()`.

```
def report(message="", error=False):
   if len(message) >= 70 and not error:
      message = message[:67] + "..."
   sys.stdout.write("\r{:70}{}".format(message, "\n" if error else ""))
   sys.stdout.flush()
```

Для удобства эта функция помещена в модуль `Qtrac.py`, поскольку используется во всех консольных параллельных программах из этой главы. Она выводит в текущую строку консоли указанное сообщение (обрезая его при необходимости до 70 символов) и сбрасывает стандартный вывод, чтобы строка напечаталась немедленно. Если сообщение содержит сведения об ошибке, то выводится знак новой строки, чтобы это сообщение не было затерто следующим; обрезка в этом случае также не производится.

```
def scale(size, smooth, source, target, concurrency):
   canceled = False
   jobs = multiprocessing.JoinableQueue()
   results = multiprocessing.Queue()
   create_processes(size, smooth, jobs, results, concurrency)
   todo = add_jobs(source, target, jobs)
   try:
      jobs.join()
   except KeyboardInterrupt: # Может не работать в Windows
      Qtrac.report("canceling...")
      canceled = True
   copied = scaled = 0
   while not results.empty(): # Безопасно, т. к. все задачи уже завершены
      result = results.get_nowait()
      copied += result.copied
      scaled += result.scaled
   return Summary(todo, copied, scaled, canceled)
```

Это основная функция многопроцессной программы масштабирования с использованием очереди, ее работа иллюстрируется на рис. 4.1. Сначала функция создает очередь работ с ожиданием. Поскольку это очередь с ожиданием (joinable), то программа будет ждать ее опустошения. Затем создается очередь результатов без ожидания. После этого создаются процессы, которые будут заниматься содержа-

тельной работой; все они готовы к работе, но заблокированы, потому что в очереди еще нет ни одной задачи. Наконец, мы вызываем функцию `add_jobs()`, чтобы заполнить очередь задач.

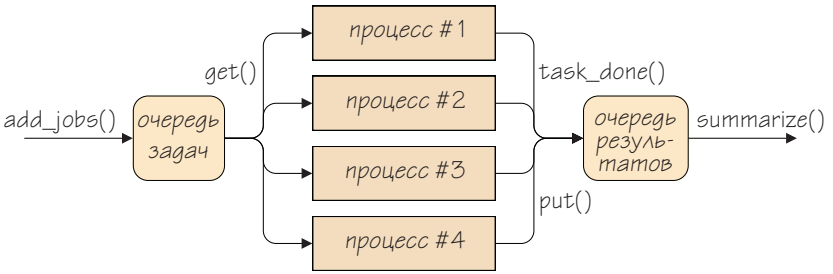

Рис. 4.1. Параллельная обработка задач и результатов с помощью очередей

Поместив все задачи в очередь, мы вызываем метод `multiprocessing.JoinableQueue.join()`, который ждет, когда очередь опустеет. Он вызывается внутри блока `try ... except`, чтобы в случае, если пользователь решит снять программу (например, нажав `Ctrl+C` в Unix), можно было аккуратно обработать отмену.

После того как все задачи выполнены (или программа снята), мы обходим очередь результатов. Обычно использование метода `empty()` для параллельной очереди ненадежно, но здесь все работает нормально, потому что рабочие процессы уже завершены, и очередь больше не обновляется. По той же причине мы можем использовать для получения результатов неблокирующий метод `multiprocessing.Queue.get_nowait()` вместо блокирующего метода `multiprocessing.Queue.get()`.

Получив все результаты, мы возвращаем именованный кортеж `Summary`, содержащий детальные данные. При нормальном завершении значение `todo` (осталось) будет равно 0, а флаг `canceled` равен `False`, но если программа была принудительно снята, то `todo`, скорее всего, будет отлично от 0, а `canceled` равен `True`.

Хотя эта функция называется `scale()`, на самом деле это общая функция «выполнить работу параллельно», которая подготавливает задачи для процессов и собирает результаты. Ее легко можно адаптировать к другим ситуациям.

```
def create_processes(size, smooth, jobs, results, concurrency):
   for _ in range(concurrency):
      process = multiprocessing.Process(target=worker, args=(size,
```

```
                smooth, jobs, results))
        process.daemon = True
        process.start()
```

Эта функция создает процессы, которые будут выполнять работу. Каждому процессу передается одна и та же функция `worker()` (поскольку все они делают одно и то же) и данные о предстоящей им работе, в том числе разделяемые очереди задач и результатов. Естественно, нам нет нужды думать о блокировке этих очередей, потому что каждая очередь сама заботится о синхронизации доступа. Создав процесс, мы делаем его демоном: когда главный процесс завершается, он завершает и все свои дочерние процессы-демоны (тогда как процессы, не являющиеся демонами, продолжают работать и в Unix становятся так называемыми «зомби»).

После того как процесс создан и сделан демоном, мы говорим ему, что можно начинать выполнение переданной функции. Начав выполнение, процесс сразу же блокируется, потому что в очереди еще нет задач. Но это не страшно, так как блокируется отдельный процесс, а главному это не мешает работать. Таким образом, создание процессов завершается быстро, и функция возвращает управление. После этого вызывающая программа добавляет в очередь задачи, которые заблокированные процессы смогут обработать.

```
def worker(size, smooth, jobs, results):
   while True:
      try:
         sourceImage, targetImage = jobs.get()
         try:
            result = scale_one(size, smooth, sourceImage, targetImage)
            Qtrac.report("{} {}".format("copied" if result.copied else
                 "scaled", os.path.basename(result.name)))
            results.put(result)
         except Image.Error as err:
            Qtrac.report(str(err), True)
      finally:
         jobs.task_done()
```

Можно было бы создать подкласс `multiprocessing.Process` (или `threading.Thread`), отвечающий за выполнение параллельных задач. Но мы поступили немного проще – создали функцию, которая передается экземпляру `multiprocessing.Process` в качестве аргумента `target` (то же самое можно сделать и с классом `threading.Thread`).

Исполнитель `worker` входит в бесконечный цикл, на каждой итерации которого пытается получить из разделяемой очереди задачу.

В данном случае бесконечный цикл безопасен, потому что процесс является демоном и, следовательно, будет завершен при выходе из программы. Метод `multiprocessing.Queue.get()` блокируется в ожидании задачи, а, получив ее, возвращает кортеж из двух элементов: имя исходного и конечного файла с изображением.

Получив задачу, мы масштабируем (или копируем) изображение, вызывая функцию `scale_one()`, после чего сообщаем, что получилось. Результирующий объект `result` (типа `Result`) мы помещаем в разделяемую очередь результатов.

При использовании очереди с ожиданием *крайне важно* после обработки для каждой задачи вызывать метод `multiprocessing.JoinableQueue.task_done()`. Именно так метод `multiprocessing.JoinableQueue.join()` узнает, когда ожидание заканчивается (то есть в очереди не осталось задач).

```
def add_jobs(source, target, jobs):
   for todo, name in enumerate(os.listdir(source), start=1):
      sourceImage = os.path.join(source, name)
      targetImage = os.path.join(target, name)
      jobs.put((sourceImage, targetImage))
   return todo
```

Сразу после задания и запуска все процессы заблокированы в ожидании задач в разделяемой очереди.

Для каждого подлежащего обработке изображения эта функция создает две строки: `sourceImage` содержит полный путь к исходному изображению, а `targetImage` – полный путь к конечному. Пара путей добавляется в разделяемую очередь в виде 2-кортежа. В конце функция возвращает общее количество подлежащих обработке задач.

После помещения в очередь первой же задачи один из заблокированных процессов заберет ее и начнет обрабатывать. То же происходит со второй задачей, с третьей и так далее, пока у каждого ожидающего процесса не появится по задаче. После этого задачи в очереди продолжат накапливаться, пока процессы-исполнители делают свое дело. Как только какой-то исполнитель освободится, он заберет из очереди следующую задачу. В конечном итоге все задачи будут разобраны и обработаны, и исполнители окажутся заблокированы в ожидании новой работы. А завершатся они при выходе из программы.

```
def scale_one(size, smooth, sourceImage, targetImage):
   oldImage = Image.from_file(sourceImage)
   if oldImage.width <= size and oldImage.height <= size:
```

```
            oldImage.save(targetImage)
            return Result(1, 0, targetImage)
        else:
            if smooth:
                scale = min(size / oldImage.width, size / oldImage.height)
                newImage = oldImage.scale(scale)
            else:
                stride = int(math.ceil(max(oldImage.width / size,
                                           oldImage.height / size)))
                newImage = oldImage.subsample(stride)
            newImage.save(targetImage)
            return Result(0, 1, targetImage)
```

Именно в этой функции происходит собственно масштабирование (или копирование). Она использует модуль `cyImage` (раздел 5.3), либо – если он отсутствует – модуль `Image` (раздел 3.12). Если изображение уже меньше указанного размера, то оно просто копируется в конечный каталог, а в результирующем объекте `Result` сообщается, что скопировано 1 изображение, масштабировано 0 и указывается имя конечного файла. В противном случае производится либо плавное масштабирование изображение, либо выборка подмножества пикселей, и сохраняется результат. В таком случае в возвращенном объекте `Result` сообщается, что скопировано 0 изображений, масштабировано 1 и опять-таки указывается имя конечного файла.

```
def summarize(summary, concurrency):
    message = "copied {} scaled {} ".format(summary.copied, summary.scaled)
    difference = summary.todo - (summary.copied + summary.scaled)
    if difference:
        message += "skipped {} ".format(difference)
    message += "using {} processes".format(concurrency)
    if summary.canceled:
        message += " [canceled]"
    Qtrac.report(message)
    print()
```

После обработки всех изображений (то есть когда очередь задач опустела) создается объект `Summary` (в функции `scale()`), который передается этой функции. Ниже показаны результаты типичного прогона программы, вторая строка сформирована этой функцией.

```
$ ./imagescale-m.py -S /tmp/images /tmp/scaled
copied 0 scaled 56 using 4 processes
```

Для хронометража в Linux просто поставьте перед именем команды `time`. В Windows нет встроенной команды для этой цели, но реше-

ния существуют[4]. (Хронометраж *внутри* программы, занимающейся многопроцессной обработкой, не годится. В наших экспериментах мы таким образом получали время работы главного процесса, не включая рабочие процессы. Отметим, что в Python 3.3 есть модуль `time`, который содержит несколько функций точного измерения времени.)

Пятисекундная разница во времени работы `imagescale-q-m.py` и `imagescalem.py` несущественна; в следующем прогоне легко могло бы получиться наоборот. По существу, обе версии эквивалентны.

4.1.2. Будущие объекты и многопроцессная обработка

В версии Python 3.2 появился модуль `concurrent.futures`, предлагающий изящный высокоуровневый способ организовать параллелизм в Python с помощью нескольких потоков или процессов. В этом подразделе мы рассмотрим три функции из программы `imagescale-m.py` (все остальное не отличается от программы `imagescale-q-m.py` из предыдущего подраздела). В программе `imagescale-m.py` используются *будущие объекты*. Согласно документации, `concurrent.futures.Future` – объект, который «инкапсулирует асинхронное выполнение вызываемого объекта» (см. `docs.python.org/dev/library/concurrent.futures.html#future-objects`). Для создания будущих объектов служит метод `concurrent.futures.Executor.submit()`. Будущий объект может сообщить о своем состоянии (отменен, выполняется, завершен) и вернуть результат или исключение.

Класс `concurrent.futures.Executor` невозможно использовать напрямую, потому что он абстрактный. Необходимо использовать один из двух его конкретных подклассов. В классе `concurrent.futures.ProcessPoolExecutor()` реализован параллелизм на уровне процессов. Из-за применения пула процессов вместе с этим классом можно использовать только такие объекты `Future`, которые исполняют или возвращают сериализуемые в формате pickle объекты, в том числе, конечно, невложенные функции. Это ограничение не распространяется на класс `concurrent.futures.ThreadPoolExecutor`, который реализует параллелизм на уровне потоков.

Концептуально, использовать пулы процессов или потоков проще, чем очереди, как видно из рис. 4.2.

---

[4] См., например, `stackoverflow.com/questions/673523/how-to-measure-execution-time-of-commandin-windows-command-line`.

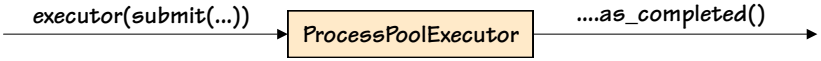

Рис. 4.2. Параллельная обработка задач и результатов с помощью исполнителя с пулом

```
def scale(size, smooth, source, target, concurrency):
   futures = set()
   with concurrent.futures.ProcessPoolExecutor(
           max_workers=concurrency) as executor:
      for sourceImage, targetImage in get_jobs(source, target):
         future = executor.submit(scale_one, size, smooth, sourceImage,
                targetImage)
         futures.add(future)
      summary = wait_for(futures)
      if summary.canceled:
         executor.shutdown()
      return summary
```

Эта функция имеет такую же сигнатуру и делает то же самое, что одноименная функция в программе `imagescale-q-m.py`, только вот работает она совершенно по-другому. В начале мы создаем пустое множество будущих объектов, затем – исполнитель, работающий с пулом процессов. За кулисами при этом создается несколько рабочих процессов. Сколько именно, определяется с помощью разных эвристик, но мы решили не полагаться на них, а задали количество процессов сами – для удобства отладки и хронометража.

Имея исполнитель, мы перебираем задачи, которые возвращает функция `get_jobs()`, и передаем каждую из них пулу. Метод `concurrent.futures.ProcessPoolExecutor.submit()` принимает рабочую функцию и необязательные аргументы, а возвращает объект типа `Future`. Мы помещаем все будущие объекты в созданное ранее множество. Пул начинает работать, как только появляется первый будущий объект. Создав все будущие объекты, мы вызываем функцию `wait_for()`, передавая ей множество этих объектов. Функция блокирует выполнение, пока все будущие объекты не будут завершены (или пользователь не снимет программу). Если пользователь прерывает программу, то мы останавливаем исполнитель `ProcessPoolExecutor` вручную.

```
def get_jobs(source, target):
   for name in os.listdir(source):
      yield os.path.join(source, name), os.path.join(target, name)
```

Эта функция занимается по существу тем же, чем функция `add_jobs()` из предыдущего подраздела, только она не добавляет задачи в очередь, а поставляет их по запросу.

```
def wait_for(futures):
   canceled = False
   copied = scaled = 0
   try:
      for future in concurrent.futures.as_completed(futures):
         err = future.exception()
         if err is None:
            result = future.result()
            copied += result.copied
            scaled += result.scaled
            Qtrac.report("{} {}".format("copied" if result.copied else
                   "scaled", os.path.basename(result.name)))
         elif isinstance(err, Image.Error):
            Qtrac.report(str(err), True)
         else:
            raise err # Что-то непредвиденное
   except KeyboardInterrupt:
      Qtrac.report("canceling...")
      canceled = True
      for future in futures:
         future.cancel()
   return Summary(len(futures), copied, scaled, canceled)
```

После того как все будущие объекты созданы, мы вызываем эту функцию, которая ждет, пока они завершатся. Функция `concurrent.futures.as_completed()` блокирует программу, пока будущий объект не завершится (или не будет отменен), после чего возвращает этот объект. Если рабочий вызываемый объект, который исполнялся будущим объектом, возбудит исключение, то метод `Future.exception()` вернет это исключение; в противном случае он возвращает `None`. Если исключений не было, то мы извлекаем из будущего объекта результат и сообщаем пользователю о ходе работы. Если возникло ожидаемое исключение (например, в модуле `Image`), то мы также сообщаем пользователю. Если же исключение неожиданное, то мы повторно возбуждаем его, так как это означает, что либо в программе имеется логическая ошибка, либо пользователь снял программу, нажав `Ctrl+C`.

Если пользователь нажал `Ctrl+C`, то мы обходим все будущие объекты и отменяем каждый из них. В конце печатается итог проделанной работы.

Работать с `concurrent.futures` легче и надежнее, чем с очередями, однако оба подхода гораздо проще и лучше, чем явное использование

блокировок в многопоточной программе. К тому же, совсем нетрудно перейти от многопроцессной программы к многопоточной – достаточно лишь заменить `concurrent.futures.ProcessPoolExecutor` на `concurrent.futures.ThreadPoolExecutor`. В многопоточной программе, если нам нужен доступ к разделяемым данным, то следует либо только читать их (используя неизменяемые типы либо или выполняя глубокое копирование), либо применять блокировки (чтобы сериализовать доступ для чтения и записи), либо пользоваться потокобезопасными типами (например, `queue.Queue`). Аналогично для доступа к разделяемым данным в многопроцессной программе мы должны либо пользоваться неизменяемыми типами или глубоким копированием, либо – если хотим читать и записывать данные – работать с управляемыми типами `multiprocessing.Value` или `multiprocessing.Array` или очередями `multiprocessing.Queue`. В идеале хорошо бы вообще избегать разделяемых данных. Если это никак невозможно, то нужно постараться обращаться к ним только для чтения (применяя неизменяемые типы или глубокое копирование) либо пользоваться синхронизированными очередями, не требующими явных блокировок. Тогда код будет легко понять и сопровождать.

4.2. Распараллеливание задач, ограниченных скоростью ввода-вывода

Часто возникает необходимость загрузить группу файлов или веб-страниц из Интернета. Из-за сетевых задержек обычно есть возможность одновременно выполнять сразу несколько операций загрузки, а, значит, закончить гораздо быстрее, чем при загрузке файлов последовательно.

В этом разделе мы рассмотрим программы `whatsnew-q.py` и `whatsnew-t.py`. Они загружают RSS-ленты: небольшие XML-документы со сводкой технических новостей. Ленты берутся с разных сайтов, а программа собирает их в единую HTML-страницу, содержащую ссылки на все новости. На рис. 4.3 показана часть такой сгенерированной страницы «что нового». В табл. 4.2 приведены данные хронометража различных версий программы[5]. И хотя складывается впе-

[5] Данные хронометража получены на слабо загруженной машине с 4-ядерным процессором AMD64 3 Ггц в ходе загрузки почти с 200 сайтов по широкополосному соединению из дома.

чатление, что ускорение пропорционально количеству ядер, на самом деле это просто совпадение; все ядра были недогружены, и большую часть времени программа проводила в ожидании завершения сетевого ввода-вывода.

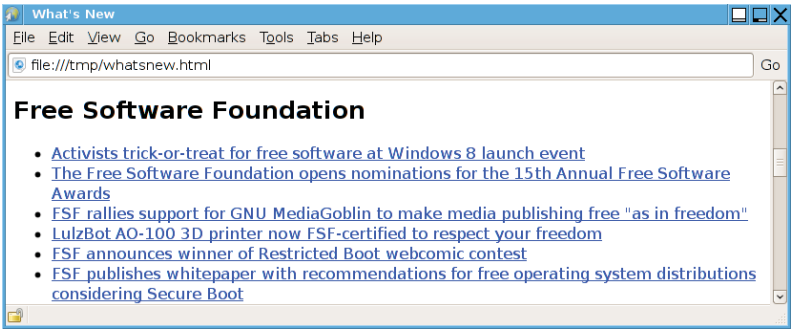

Рис. 4.3. Некоторые ссылки на технические новости из RSS-лент

В таблице приведены также данные хронометража версий программы `gigapixel` (в книге не показана). Каждая из них обращается к сайту `www.gigapan.org` и читает примерно 500 файлов в формате JSON общим размером 1,9 МиБ, которые содержат метаданные гигапиксельных изображений. Код этих программ аналогичен версиям программы `whatsnew`, но `gigapixel` достигает гораздо более высокой производительности. Разница объясняется тем, что `gigapixel` обращается к одному сайту с большой полосой пропускания, тогда как программы `whatsnew` заходят на разные сайты с сильно различающимися сетевыми характеристиками.

Поскольку сетевые задержки изменяются в широких пределах, то и ускорение заметно варьирует, так что параллельная версия может отличаться от последовательной то всего в 2 раза, а то в 10 и даже больше, в зависимости от того, к каким сайтам обращается, сколько данных загружает и какова пропускная способность сетевого соединения. Поэтому различия между многопроцессными и многопоточными версиями несущественны и легко могут оказаться прямо противоположными при следующем прогоне.

Таблица 4.2. Сравнение скоростей скачивания

Программа	Степень параллелизма	Секунд	Ускорение
`whatsnew.py`	*Нет*	172	*Эталон*
`whatsnew-c.py`	16 сопрограмм	180	0,96×

Таблица 4.2. (окончание)

Программа	Степень параллелизма	Секунд	Ускорение
whatsnew-q-m.py	16 процессов с очередью	45	3,82×
whatsnew-m.py	16 процессов с пулом процессов	50	3,44×
whatsnew-q.py	16 потоков с очередью	50	3,44×
whatsnew-t.py	16 потоков с пулом потоков	48	3,58×
gigapixel.py	*Нет*	238	*Эталон*
gigapixel-q-m.py	16 процессов с очередью	35	6,80×
gigapixel-m.py	16 процессов с пулом процессов	42	5,67×
gigapixel-q.py	16 потоков с очередью	37	6,43×
gigapixel-t.py	16 потоков с пулом потоков	37	6,43×

Из табл. 4.2 следует один важный вывод: за счет распараллеливания скачивание можно сильно ускорить, хотя фактический коэффициент ускорения зависит от обстоятельств и меняется от прогона к прогону.

4.2.1. Очереди и многопоточность

Начнем с рассмотрения программы `whatsnew-q.py`, в которой используется несколько потоков и две потокобезопасных очереди. В первой очереди хранятся задачи, представленные URL-адресами подлежащих скачиванию документов. Во второй очереди находятся результаты – 2-кортежи, содержащие `True` и HTML-фрагмент, вставляемый в конструируемую HTML-страницу, или `False` и сообщение об ошибке.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

Функция `main()` координирует всю работу. Сначала она выделяет из командной строки параметры `limit` (какое максимальное число новостей читать с указанного URL-адреса) и `concurrency` (степень параллелизма, полезную для отладки и хронометража). Затем про-

грамма сообщает пользователю, что она начала работать, и получает полное имя файла, содержащего URL-адреса и соответствующие им однострочные заголовки.

Далее эта функция создает две потокобезопасных очереди и рабочие потоки. После того как все рабочие потоки запущены (в этот момент они, разумеется, блокированы, потому что никакой работы еще нет), мы добавляем все задачи в первую очередь. Наконец, в функции `process()` мы дожидаемся завершения всех задач и выводим результаты. Общая структура параллелизма показана на рис. 4.4.

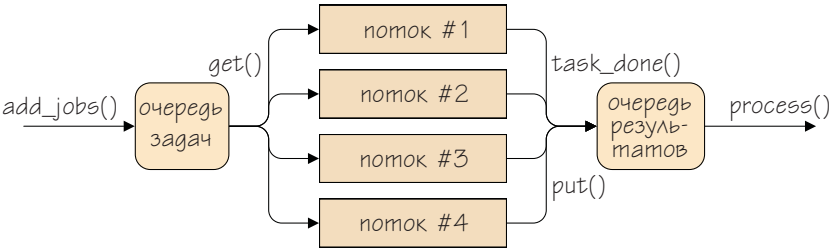

Рис. 4.4. Параллельная обработка задач и результатов с помощью очередей

Кстати, если бы задач было очень много или добавление задачи отнимало бы много времени, то было бы разумнее ставить задачи в очередь в отдельном потоке (или процессе).

```
def handle_commandline():
   parser = argparse.ArgumentParser()
   parser.add_argument("-l", "--limit", type=int, default=0,
            help="the maximum items per feed [default: unlimited]")
   parser.add_argument("-c", "--concurrency", type=int,
            default=multiprocessing.cpu_count() * 4,
            help="specify the concurrency (for debugging and "
               "timing) [default: %(default)d]")
   args = parser.parse_args()
   return args.limit, args.concurrency
```

Поскольку программы `whatsnew` ограничены скоростью вывода-вывода, то мы зададим для них степень параллелизма по умолчанию, кратную количеству ядер, – в данном случае 4×[6].

```
def create_threads(limit, jobs, results, concurrency):
   for _ in range(concurrency):
```

[6] Этот коэффициент выбран, потому что в наших тестах показал наибольшую производительность. Мы рекомендуем вам поэкспериментировать, т. к. все конфигурации разные.

```
    thread = threading.Thread(target=worker, args=(limit, jobs,
             results))
    thread.daemon = True
    thread.start()
```

Эта функция создает `concurrency` рабочих потоков, передавая каждой исполняемую функцию и аргументы, с которыми эту функцию вызывать.

Как и в предыдущем разделе, мы делаем каждый поток демоном, чтобы он завершался при выходе из программы. Сразу после запуска рабочие потоки оказываются заблокированными, т. к. задач еще нет, но главный поток программы по-прежнему работает.

```
def worker(limit, jobs, results):
   while True:
      try:
         feed = jobs.get()
         ok, result = Feed.read(feed, limit)
         if not ok:
            Qtrac.report(result, True)
         elif result is not None:
            Qtrac.report("read {}".format(result[0][4:-6]))
            results.put(result)
      finally:
         jobs.task_done()
```

В каждом рабочем потоке исполняется бесконечный цикл. Это безопасно, так как, будучи демонами, потоки автоматически завершатся по окончании работы программы.

Эта функция ждет получения задачи из очереди задач. Получив задачу, она обращается к функции `Feed.read()` из написанного нами модуля `Feed.py`, чтобы прочитать данные с указанного URL-адреса. Все версии программы `whatsnew` ожидают, что модуль `Feed.py` предоставит итератор для файла задач и объект-читатель для каждой RSS-ленты. Если во время чтения произойдет ошибка, то `ok` будет равно `False`, и мы печатаем результат (содержащий сообщение об ошибке). В противном случае переменная `result` содержит список HTML-строк, и мы печатаем первый элемент из него (вырезая HTML-теги) и добавляем весь список в очередь результатов.

Поскольку мы создали очередь с ожиданием, то каждому вызову `queue.Queue.get()` должен соответствовать вызов `queue.Queue.task_done()`. Мы гарантируем это с помощью блока `try…finally`[7].

[7] Отметим, что потокобезопасная очередь с ожиданием реализована в классе `queue.Queue`, тогда как его многопроцессный эквивалент называется `multiprocessing.JoinableQueue`, а не `multiprocessing.Queue`.

```
def read(feed, limit, timeout=10):
   try:
      with urllib.request.urlopen(feed.url, None, timeout) as file:
         data = file.read()
      body = _parse(data, limit)
      if body:
         body = ["<h2>{}</h2>\n".format(escape(feed.title))] + body
         return True, body
      return True, None
   except (ValueError, urllib.error.HTTPError, urllib.error.URLError,
            etree.ParseError, socket.timeout) as err:
      return False, "Error: {}: {}".format(feed.url, err)
```

Функция `Feed.read()` читает данные из переданного ей URL-адреса (`feed`) и пытается их разобрать. Если это получается, то функция возвращает `True` и список HTML-фрагментов (заголовок и одну или более ссылок); в противном случае она возвращает `False` и `None` или сообщение об ошибке.

```
def _parse(data, limit):
   output = []
   feed = feedparser.parse(data) # Atom + RSS
   for entry in feed["entries"]:
      title = entry.get("title")
      link = entry.get("link")
      if title:
         if link:
            output.append('<li><a href="{}">{}</a></li>'.format(
                  link, escape(title)))
         else:
            output.append('<li>{}</li>'.format(escape(title)))
      if limit and len(output) == limit:
         break
   if output:
      return ["<ul>"] + output + ["</ul>"]
```

Модуль `Feed.py` содержит две версии закрытой функции `_parse()`. В той, что приведена здесь, используется сторонний модуль `feedparser` (`pypi.python.org/pypi/feedparser`), который умеет разбирать новостные ленты в форматах Atom и RSS. Другая версия (не показана) используется, если модуль `feedparser` недоступен, она умеет разбирать только формат RSS.

Функция `feedparser.parse()` делает всю трудную работу по разбору новостной ленты. Нам остается только обойти созданные ей записи, извлечь из них заголовок и ссылку для каждой новости и построить HTML-список для их представления.

```
def add_jobs(filename, jobs):
   for todo, feed in enumerate(Feed.iter(filename), start=1):
      jobs.put(feed)
   return todo
```

Новости возвращаются функцией `Feed.iter()` в виде 2-кортежа (*`title`*, *`url`*), который добавляется в очередь задач. В конце мы возвращаем общее количество помещенных в очередь задач.

В данном случае можно было бы просто вернуть *`jobs`*`.qsize()`, а не подсчитывать задачи самостоятельно. Но если бы мы вызывали `add_jobs()` в отдельном потоке, то использовать `queue.Queue.qsize()` было бы ненадежно, потому что задачи выбираются из очереди одновременно с добавлением.

```
Feed = collections.namedtuple("Feed", "title url")

def iter(filename):
   name = None
   with open(filename, "rt", encoding="utf-8") as file:
      for line in file:
         line = line.rstrip()
         if not line or line.startswith("#"):
            continue
         if name is None:
            name = line
         else:
            yield Feed(name, line)
            name = None
```

Это функция `Feed.iter()` из модуля `Feed.py`. Предполагается, что `whatsnew.dat` – простой текстовый файл в кодировке UTF-8, который содержит по две строки на каждую новость: в первой – заголовок (например, `The Guardian - Technology`), а во второй – URL-адрес (например, `http://feeds.pinboard.in/rss/u:guardiantech/`). Пустые строки и комментарии (строки, начинающиеся символом `#`) игнорируются.

```
def process(todo, jobs, results, concurrency):
   canceled = False
   try:
      jobs.join() # Ждать окончания всей работы
   except KeyboardInterrupt: # В Windows может не работать
      Qtrac.report("canceling...")
      canceled = True
   if canceled:
      done = results.qsize()
   else:
```

```
        done, filename = output(results)
        Qtrac.report("read {}/{} feeds using {} threads{}".format(done, todo,
                concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```

После того как все потоки созданы и все задачи помещены в очередь, вызывается эта функция. Она обращается к функции `queue.Queue.join()`, которая ждет опустошения очереди (то есть завершения всех задач) или снятия программы пользователем. Если пользователь не прервет программу, то будет вызвана функция `output()`, которая запишет HTML-файл, содержащий весь список ссылок, а затем напечатает сводку. И наконец, вызывается функция `open()` из модуля `webbrowser`, которая открывает этот файл в браузере (см. рис. 4.3).

```
def output(results):
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")
        file.write("<body><h1>What's New</h1>\n")
        while not results.empty(): # Safe because all jobs have finished
            result = results.get_nowait()
            done += 1
            for item in result:
                file.write(item)
        file.write("</body></html>\n")
    return done, filename
```

Эта функция вызывается по завершении всех задач, и ей передается очередь результатов. Каждый результат представляет собой список HTML-фрагментов (заголовок, за которым следует одна или несколько ссылок). Функция создает файл `whatsnew.html` и записывает в него все заголовки и ссылки новостей. В конце она возвращает общее количество результатов (то есть успешно завершившихся задач) и имя созданного файла. Эта информация используется функцией `process()` для печати сводки и открытия HTML-файла в браузере.

4.2.2. Будущие объекты и многопоточность

В версии Python, начиная с 3.2, мы можем воспользоваться модулем `concurrent.futures`, чтобы сделать все то же самое, не прибегая к очередям (или явным блокировкам). В этом подразделе мы рассмот-

рим программу whatsnew-t.py, которая работает с этим модулем, опустив, однако, функции, не отличающиеся от тех, что были показаны выше (например, handle_commandline() и функции из модуля Feed.py).

```
def main():
   limit, concurrency = handle_commandline()
   Qtrac.report("starting...")
   filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
   futures = set()
   with concurrent.futures.ThreadPoolExecutor(
            max_workers=concurrency) as executor:
      for feed in Feed.iter(filename):
         future = executor.submit(Feed.read, feed, limit)
         futures.add(future)
      done, filename, canceled = process(futures)
      if canceled:
         executor.shutdown()
   Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
          len(futures), concurrency, " [canceled]" if canceled else ""))
   print()
   if not canceled:
      webbrowser.open(filename)
```

Эта функция создает пустое множество будущих объектов, а затем исполнитель, работающий с пулом потоков. Он устроен так же, как рассмотренный ранее исполнитель, только место процессов занимают потоки. В контексте этого исполнителя мы обходим файл данных и для каждой ленты создаем новый будущий объект (с помощью метода concurrent.futures.ThreadPoolExecutor.submit()), который будет вызывать функцию Feed.read() для URL-адреса feed, возвращая не более limit ссылок. Созданный будущий объект добавляется в множество.

Создав все будущие объекты, мы вызываем функцию process(), которая ждет завершения всех будущих объектов (или снятия программы пользователем). После этого печатается сводка результатов и, если пользователь не прерывал программы, созданная HTML-страница открывается в браузере.

```
def process(futures):
   canceled = False
   done = 0
   filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
   with open(filename, "wt", encoding="utf-8") as file:
      file.write("<!doctype html>\n")
      file.write("<html><head><title>What's New</title></head>\n")
```

```
      file.write("<body><h1>What's New</h1>\n")
      canceled, results = wait_for(futures)
      if not canceled:
         for result in (result for ok, result in results if ok and
                  result is not None):
            done += 1
            for item in result:
               file.write(item)
      else:
         done = sum(1 for ok, result in results if ok and result is not
                    None)
      file.write("</body></html>\n")
   return done, filename, canceled
```

Эта функция записывает в HTML-файл заголовок, а затем вызывает функцию `wait_for()`, которая ждет завершения работы. Если пользователь не прервал программу, то эта функция обходит все результаты (каждый из них является либо парой `True`, `list`, либо парой `False`, `str`, либо парой `False`, `None`) и для тех из них, которые содержат списки (состоящие из заголовка и одной или более ссылок), записывает их элементы в HTML-файл.

Если программа была прервана, то мы просто подсчитываем, сколько лент успешно обработано. В любом случае возвращается количество прочитанных лент, имя HTML-файла и признак прерывания пользователем.

```
def wait_for(futures):
   canceled = False
   results = []
   try:
      for future in concurrent.futures.as_completed(futures):
         err = future.exception()
         if err is None:
            ok, result = future.result()
            if not ok:
               Qtrac.report(result, True)
            elif result is not None:
               Qtrac.report("read {}".format(result[0][4:-6]))
            results.append((ok, result))
         else:
            raise err # Что-то непредвиденное
   except KeyboardInterrupt:
      Qtrac.report("canceling...")
      canceled = True
      for future in futures:
         future.cancel()
   return canceled, results
```

Эта функция обходит все будущие объекты, дожидаясь, пока каждый будет завершен или отменен. Получив будущий объект, функция сообщает об ошибке или об успешно прочитанной ленте и в любом случае добавляет булево значение и результат (список строк или строку сообщения об ошибке) в список результатов.

Если пользователь прервал программу (нажав `Ctrl+C`), то мы отменяем все будущие объекты. В конце мы возвращаем признак прерывания программы и список результатов.

Модуль `concurrent.futures` одинаково удобен при работе как с несколькими потоками, так и с несколькими процессами. Если говорить о производительности, то понятно, что при подходящих условиях – когда задача ограничена скоростью ввода-вывода, а не быстродействием процессора – и при должной осторожности многопоточность способна дать ожидаемое повышение производительности.

4.3. Пример: приложение с параллельным ГИП

Писать параллельные приложения с графическим интерфейсом пользователя (ГИП) нелегко, особенно если использовать Tkinter – стандартную библиотеку построения ГИП, входящую в дистрибутив Python. Краткое введение в программирование ГИП с помощью Tkinter приведено в главе 7; читателям, не знакомым с Tkinter, рекомендуется сначала прочитать эту главу, а затем вернуться сюда.

Очевидный подход к распараллеливанию – воспользоваться многопоточностью – на практике иногда приводит к медленно работающему, а то и зависающему интерфейсу в случае, когда объем обработки велик; в конце концов, любой графический интерфейс потребляет много процессорных ресурсов. Альтернативное решение – использовать несколько процессов – все равно дает очень «тормозной» интерфейс.

В этом разделе мы рассмотрим приложение ImageScale (находится в каталоге `imagescale`), представленное на рис. 4.5. В нем применена тактика, сочетающая параллельную обработку с отзывчивым ГИП, который сообщает о ходе продвижения и поддерживает отмену.

Как видно из рис. 4.6, в приложении комбинируются многопоточность и многопроцессность. Потоков выполнения два – главный поток ГИП и рабочий поток, который делегирует свою работу пулу процессов. При такой архитектуре интерфейс всегда отвечает быст-

ро, потому что получает большую часть процессорного времени ядра, разделяемого двумя потоками, тогда как рабочий поток (который сам почти ничего не делает) довольствуется остальным. А рабочие процессы исполняются другими ядрами (в случае многоядерной машины), так что с ГИП вообще не конкурируют.

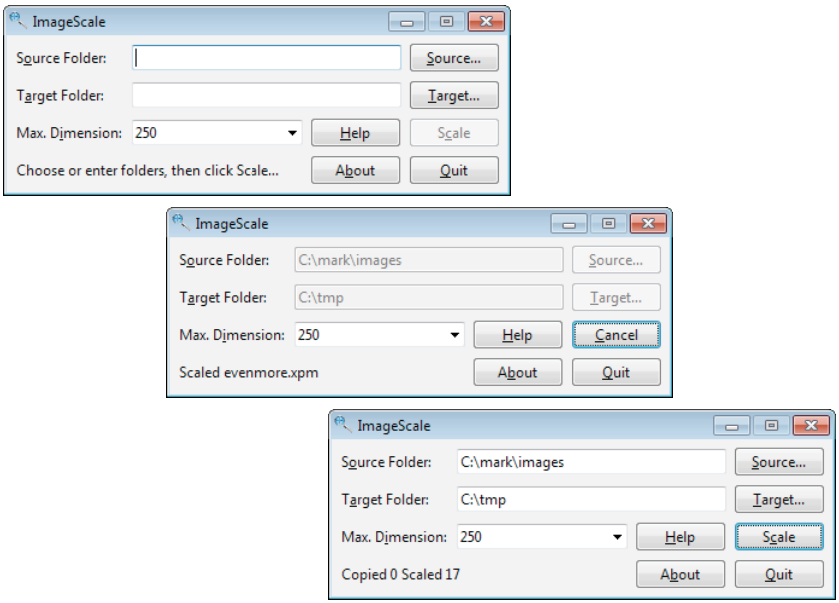

Рис. 4.5. Приложение ImageScale до, во время и после масштабирования нескольких изображений

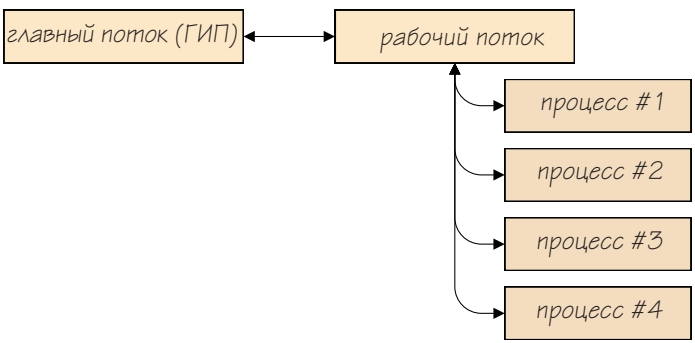

Рис. 4.6. Модель распараллеливания приложения ImageScale (стрелки показывают взаимодействия)

Сопоставимая консольная программа `imagescale-m.py` насчитывает примерно 130 строк кода (мы анализировали ее выше в разделе 4.1). Для сравнения скажем, что графическое приложение ImageScale состоит из пяти файлов (см. рис. 4.7), в совокупности содержащих почти 500 строк. При этом на код масштабирования изображений приходится всего 60 строк, все остальное – организация ГИП.

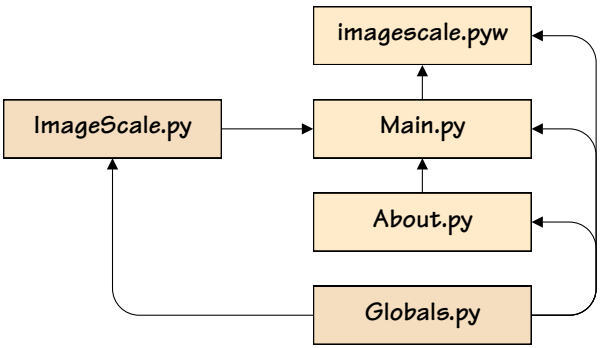

Рис. 4.7. Файлы приложения ImageScale в контексте (стрелки показывают, кто что импортирует)

В двух подразделах этого раздела мы рассмотрим прежде всего код, относящийся к программированию параллельного ГИП, а также код, необходимый для понимания происходящего.

4.3.1. Создание ГИП

В этом подразделе мы рассмотрим самый важный код, связанный с созданием ГИП и поддержкой параллелизма. Он находится в файлах `imagescale/imagescale.pyw` и `imagescale/Main.py`.

```
import tkinter as tk
import tkinter.ttk as ttk
import tkinter.filedialog as filedialog
import tkinter.messagebox as messagebox
```

Здесь в `Main.py` импортируются модули, относящиеся к ГИП. Некоторые пользователи Tkinter импортируют их в одном предложении `from tkinter import *`, но мы предпочитаем делать так, как показано, чтобы имена функций и классов находились в своих собственных пространствах имен и чтобы с этими пространствами имен было удобнее работать – писать `tk` вместо `tkinter`.

```
def main():
    application = tk.Tk()
    application.withdraw() # скрыть, пока не будем готовы показать
    window = Main.Window(application)
    application.protocol("WM_DELETE_WINDOW", window.close)
    application.deiconify() # show
    application.mainloop()
```

Это точка входа в приложение, она находится в файле `imagescale.pyw`. В функции есть и другой – не показанный – код, где задаются пользовательские настройки и устанавливается значок приложения.

Важно отметить, что мы обязательно должны создать объект верхнего уровня `tkinter.Tk`, обычно невидимый. Затем мы создаем экземпляр окна (в данном случае принадлежащего подклассу `tkinter.ttk.Frame`) и, наконец, запускаем цикл обработки событий Tkinter.

Чтобы избежать мигания при появлении не полностью готового окна, мы скрываем приложение сразу после создания (так что пользователь его в этот момент не увидит), а показываем, когда окно будет окончательно сформировано.

Метод `tkinter.Tk.protocol()` вызывается, для того чтобы сообщить Tkinter, что при нажатии пользователем кнопки закрытия окна ×, нужно вызвать метод `Main.Window.close()`[8]. Этот метод мы обсудим ниже (подраздел 4.3.4).

Структура программ с ГИП напоминает структуру некоторых серверных программ в том смысле, что после запуска они просто ждут появления событий, на которые затем реагируют. На сервере в качестве событий могут выступать запросы на установление сетевых соединений и поступающие пакеты, а в графическом приложении события либо являются результатом действий пользователя (например, нажатие на клавишу или щелчок мышью), либо генерируются системой (скажем, срабатывание таймера или сообщение о том, что нужно показать какую-то часть окна, например потому что находившееся поверх него другое окно закрылось или сдвинулось в другое место). Цикл обработки событий ГИП показан на рис. 4.8. Примеры обработки событий мы видели в главе 3 (раздел 3.1).

```
PAD = "0.75m"
WORKING, CANCELED, TERMINATING, IDLE = ("WORKING", "CANCELED",
        "TERMINATING", "IDLE")
class Canceled(Exception): pass
```

[8] В системе OS X кнопка закрытия обычно выглядит как красный кружок – с точкой посередине, если в приложении есть несохраненные изменения.

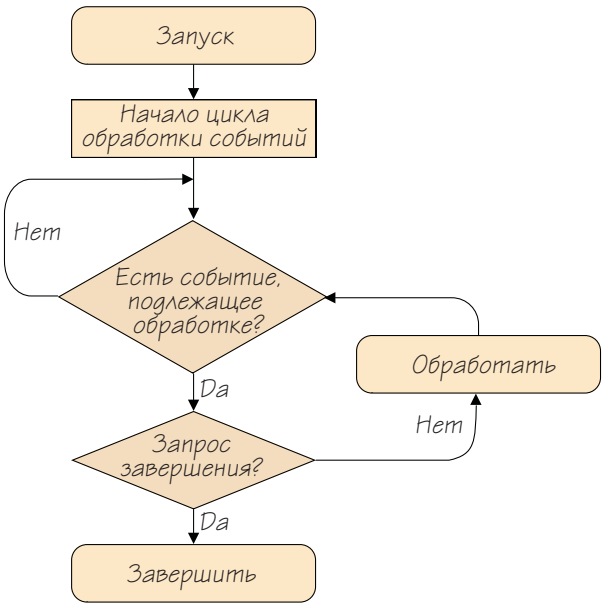

Рис. 4.8. Классический цикл обработки событий ГИП

Это некоторые константы, импортированные модулями программы ImageScale с помощью предложения `Globals import *`. Константа `PAD` задает расстояние (0,75 мм), используемое в ходе размещения виджетов (этот код не показан). Остальные константы просто обозначают состояния приложения: `WORKING`, `CANCELED`, `TERMINATING` или `IDLE`. Как обрабатывается исключение `Canceled`, мы увидим ниже.

```
class Window(ttk.Frame):

    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.create_variables()
        self.create_ui()
        self.sourceEntry.focus()
```

После создания объект `Window` обязан вызвать метод `__init__()` базового класса. В данном случае мы еще создаем переменные, которые будут нужны программе и самому пользовательскому интерфейсу. В конце передаем фокус текстовому полю для ввода имена исходного каталога. Пользователь может ввести имя каталога непосредст-

венно или нажать кнопку Source, которая открывает окно выбора каталога.

Мы не показываем ни create_ui(), ни вызываемые из него методы create_widgets(), layout_widgets() и create_bindings(), поскольку они касаются только создания ГИП и не имеют никакого отношения к параллельному программированию. (Разумеется, мы увидим примеры создания ГИП в главе 7, когда будем знакомиться с библиотекой Tkinter.)

```
def create_variables(self):
    self.sourceText = tk.StringVar()
    self.targetText = tk.StringVar()
    self.statusText = tk.StringVar()
    self.statusText.set("Choose or enter folders, then click Scale...")
    self.dimensionText = tk.StringVar()
    self.total = self.copied = self.scaled = 0
    self.worker = None
    self.state = multiprocessing.Manager().Value("i", IDLE)
```

Мы привели только относящиеся к делу строки этого метода. В переменных типа tkinter.StringVar хранятся строки, ассоциированные с виджетами пользовательского интерфейса. Переменные total, copied и scaled – различные счетчики. В переменную worker, первоначально равную None, будет записана ссылка на второй поток, когда пользователь закажет какую-нибудь операцию обработки.

Если пользователь прервет обработку (нажав кнопку Cancel), то, как мы впоследствии увидим, будет вызван метод scale_or_cancel(), который установит состояние приложения (WORKING, CANCELED, TERMINATING или IDLE). Если же пользователь захочет выйти из приложения (нажав кнопку Quit), то будет вызван метод close().

Естественно, если пользователь прерывает обработку или завершает приложение в середине масштабирования, то хотелось бы отреагировать как можно быстрее. Реакция заключается в том, что мы изменяем надпись Cancel на кнопке на Canceling..., делаем эту кнопку неактивной и не даем процессам, которыми управляет рабочий поток, больше никакой работы. После того как вся работа будет остановлена, кнопку Scale можно снова активировать. Это означает, что и потоки, и рабочие процессы должны регулярно опрашивать состояние приложения, проверяя, не запросил ли пользователь прерывание или выход.

Один из способов дать доступ к состоянию приложения – хранить его в переменной и пользоваться блокировкой. Но это означает, что нужно будет захватывать блокировку перед каждым обращением к переменной, а затем освобождать ее. Это нетрудно сделать с помощью

контекстного менеджера, но может случиться, что где-то мы забудем поставить блокировку. По счастью, в модуле `multiprocessing` имеется класс `multiprocessing.Value`, в котором можно хранить одно значение указанного типа и безопасно обращаться к нему – класс сам позаботится о блокировке (как потокобезопасная очередь). Чтобы создать объект `Value`, необходимо указать идентификатор типа – в данном случае мы указали `"i"`, то есть `int` – и начальное значение; мы задали константу `IDLE`, потому что сразу после запуска приложение находится в состоянии `IDLE`, то есть простаивает.

Следует еще отметить, что мы создаем объект `multiprocessing.Value` не напрямую, а с помощью объекта `multiprocessing.Manager`. Это необходимо для правильной работы `Value`. (Если бы в программе было несколько объектов `Value` или `Array`, то нужно было бы создать экземпляр `multiprocessing.Manager` и от него получать все объекты, но в этом примере такой необходимости нет.)

```
def create_bindings(self):
   if not TkUtil.mac():
      self.master.bind("<Alt-a>", lambda *args:
             self.targetEntry.focus())
      self.master.bind("<Alt-b>", self.about)
      self.master.bind("<Alt-c>", self.scale_or_cancel)
      ...
   self.sourceEntry.bind("<KeyRelease>", self.update_ui)
   self.targetEntry.bind("<KeyRelease>", self.update_ui)
   self.master.bind("<Return>", self.scale_or_cancel)
```

При создании объекта `tkinter.ttk.Button` мы можем ассоциировать с ним команду (то есть функцию или метод), которую Tkinter будет выполнять при нажатии кнопки. Это сделано в методе `create_widgets()` (не показан). Мы хотим также поддержать пользователей, работающих с клавиатурой. Например, метод `scale_or_cancel()` будет вызываться как при нажатии кнопки `Scale`, так и при нажатии клавиш `Alt+C` или `Enter` (на всех платформах, кроме OS X).

Сразу после запуска приложения кнопка `Scale` неактивна, потому что не задан ни исходный, ни конечный каталог. Но после того как они заданы – путем непосредственного ввода или в окне выбора каталога, появляющегося при нажатии кнопок `Source` и `Target` соответственно, кнопку `Scale` необходимо активировать. Для этого у нас есть метод `update_ui()`, который активирует или деактивирует виджеты в зависимости от ситуации, и мы вызываем этот метод, после того как пользователь заполнит текстовое поле, содержащее имя исходного или конечного каталога.

В составе примеров к книге имеется модуль `TkUtil`. Он содержит различные служебные функции, например, `TkUtil.mac()`, которая сообщает, работаем ли мы в операционной системе OS X, общие функции для работы с диалогами «О программе» и модальными диалогами и некоторые другие[9].

```
def update_ui(self, *args):
   guiState = self.state.value
   if guiState == WORKING:
      text = "Cancel"
      underline = 0 if not TkUtil.mac() else -1
      state = "!" + tk.DISABLED
   elif guiState in {CANCELED, TERMINATING}:
      text = "Canceling..."
      underline = -1
      state = tk.DISABLED
   elif guiState == IDLE:
      text = "Scale"
      underline = 1 if not TkUtil.mac() else -1
      state = ("!" + tk.DISABLED if self.sourceText.get() and
               self.targetText.get() else tk.DISABLED)
   self.scaleButton.state((state,))
   self.scaleButton.config(text=text, underline=underline)
   state = tk.DISABLED if guiState != IDLE else "!" + tk.DISABLED
   for widget in (self.sourceEntry, self.sourceButton,
         self.targetEntry, self.targetButton):
      widget.state((state,))
   self.master.update() # Нужно перерисовать ГИП
```

Этот метод вызывается при каждом изменении, которое может отразиться на пользовательском интерфейсе. Он может вызываться напрямую или в ответ на событие – нажатие клавиши или щелчок мышью – к которому привязан, в последнем случае ему передается один или несколько дополнительных аргументов, которые мы игнорируем.

Сначала мы получаем состояние ГИП (`WORKING`, `CANCELED`, `TERMINATING` или `IDLE`). Можно было бы не заводить локальную переменную, а проверять само значение `self.state.value` в каждой ветви `if`, но глубоко внутри при обращении к нему происходит блокировка, и время этой блокировки лучше бы свести к минимуму. Ничего страшного не случится, если во время выполнения этого метода состояние снова изменится, потому что в таком случае метод будет вызван еще раз.

[9] Библиотека Tkinter, а точнее лежащая в ее основе библиотека Tcl/Tk 8.5, учитывает некоторые различия между платформами Linux, OS X и Windows. Тем не менее, многие детали – особенно для OS X – нам приходится обрабатывать самостоятельно.

Когда приложение работает, на кнопке масштабирования должна отображаться надпись `Cancel` (потому что эта кнопка служит и для запуска и для остановки процесса), и она должна быть активна. На большинстве платформ подчеркивается буква, соответствующая клавише-ускорителю (например, чтобы выполнить действие, связанное с кнопкой `Cancel`, можно нажать клавиши `Alt+C`), но в OS X такая возможность не поддерживается, так что на этой платформе мы отключаем подчеркивание, указав недопустимый номер позиции.

Зная состояние приложения, мы изменяем надпись на кнопке масштабирования и позицию подчеркнутой буквы, после чего активируем или деактивируем некоторые виджеты. В конце вызывается метод `update()`, который заставляет Tkinter перерисовать окно, чтобы сделанные нами изменения стали видны.

```
def scale_or_cancel(self, event=None):
   if self.scaleButton.instate((tk.DISABLED,)):
      return
   if self.scaleButton.cget("text") == "Cancel":
      self.state.value = CANCELED
      self.update_ui()
   else:
      self.state.value = WORKING
      self.update_ui()
      self.scale()
```

Кнопка масштабирования используется как для запуска, так и для прерывания масштабирования, поскольку мы изменяем надпись на ней в зависимости от состояния приложения. Этот метод вызывается при нажатии `Alt+C` (на всех платформах, кроме OS X) или `Enter`, а также после щелчка по кнопке `Scale` или `Cancel` (кнопке масштабирования).

Если кнопка не активна, то мы ничего не делаем и сразу возвращаем управление (неактивную кнопку, конечно, нажать нельзя, но вызвать этот метод все равно можно, воспользовавшись клавишей-ускорителем `Alt+C`).

Если кнопка активна и на ней отображается надпись `Cancel`, то мы изменяем состояние приложения на `CANCELED` и обновляем пользовательский интерфейс. В частности, кнопка масштабирования становится неактивной, а надпись на ней изменяется на `Canceling...`. Как мы вскоре увидим, во время обработки периодически проверяется, не изменилось ли состояние приложения, поэтому, обнаружив, что пользователь пожелал прервать программу, мы прекращаем масштабирование. Завершив обработку прерывания, мы снова активируем

кнопку масштабирования и рисуем на ней надпись `Scale`. На рис. 4.5 показано состояние приложения до, во время и после масштабирования изображений, а на рис. 4.9 – до, во время и после прерывания.

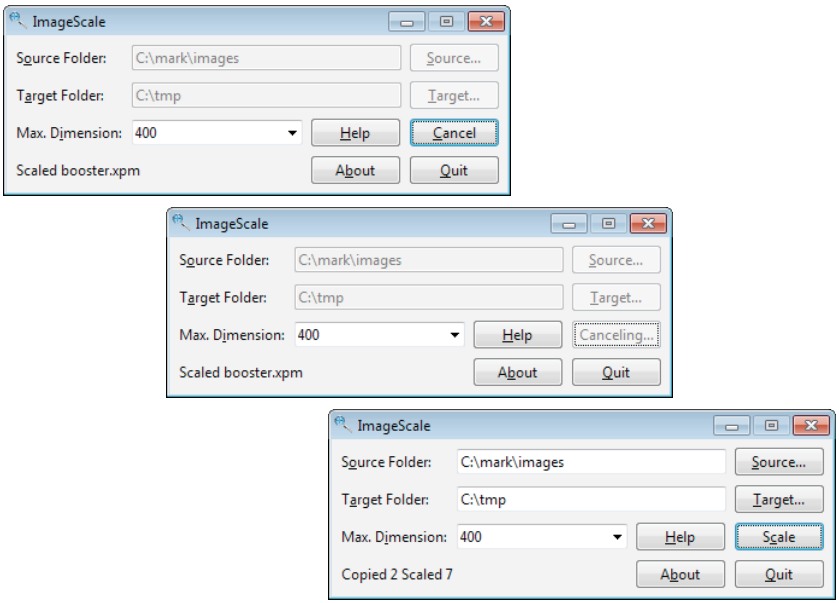

Рис. 4.9. Приложение ImageScale до, во время и после прерывания

Если на кнопке отображается надпись `Scale`, то мы переводим приложение в состояние `WORKING`, обновляем пользовательский интерфейс (так что теперь кнопка называется `Cancel`) и начинаем масштабирование.

```
def scale(self):
   self.total = self.copied = self.scaled = 0
   self.configure(cursor="watch")
   self.statusText.set("Scaling...")
   self.master.update() # Перерисовываем ГИП
   target = self.targetText.get()
   if not os.path.exists(target):
      os.makedirs(target)
   self.worker = threading.Thread(target=ImageScale.scale, args=(
         int(self.dimensionText.get()), self.sourceText.get(),
         target, self.report_progress, self.state,
         self.when_finished))
   self.worker.daemon = True
   self.worker.start() # сразу же возвращаем управление
```

Сначала мы обнуляем все счетчики и меняем форму курсора, отображая индикатор занятости. Затем изменяем текст в метке состояния и обновляем ГИП, чтобы пользователь видел, что масштабирование началось. Далее, если конечный каталог еще не существует, то мы создаем его и все промежуточные каталоги.

Все подготовив, мы создаем новый рабочий поток (на предыдущий рабочий поток не осталось ссылок, поэтому его уберет сборщик мусора). Для создания потока используется функция `threading.Thread()`, которой передается исполняемая потоком функция и передаваемые этой функции аргументы. В качестве аргументов мы указываем максимальный размер файлов масштабированных изображений, исходный и конечный каталоги, вызываемый объект (в данном случае связанный метод `self.report_progress()`), который будет вызываться по завершении каждой задачи, объект `Value`, содержащий состояние приложения, чтобы рабочие процессы могли регулярно проверять, прервано приложение или нет, и вызываемый объект (связанный метод `self.when_finished()`), который вызывается по завершении (или прерывании) обработки.

Создав поток, мы делаем его демоном, чтобы после выхода из приложения он корректно завершался, и запускаем его.

Как мы увидим, сам рабочий поток почти ничего не делает, чтобы потоку ГИП доставалось как можно больше ресурсов процессорного ядра, которое оба потока разделяют между собой. Функция `ImageScale.scale()` делегирует все обязанности рабочего потока нескольким процессам, которые выполняются на других ядрах (в случае машины с многоядерным процессором). Поэтому ГИП остается отзывчивым (хотя при такой архитектуре ГИП будет отзывчивым, даже если имеется всего одно ядро, потому что поток ГИП в любом случае получает не меньше процессорного времени, чем рабочий поток).

4.3.2. Модуль ImageScaleWorker

Мы вынесли функции, вызываемые из рабочего потока, в отдельный модуль `imagescale/ImageScale.py`, в котором и находятся показанные ниже фрагменты кода. Это не просто вопрос удобства организации, а необходимость, потому что мы хотим, чтобы эти функции можно было вызывать также из модуля `multiprocessing`, а для этого они должны допускать импортирование, и все данные модуля должны допускать сериализацию в формате pickle. Модули, которые содержат виджеты ГИП или подклассы виджетов, не должны импортироваться подобным образом, поскольку это может поставить в тупик оконную систему.

Модуль включает три функции, первая из которых, `ImageScale.scale()`, выполняется рабочим потоком.

```
def scale(size, source, target, report_progress, state, when_finished):
   futures = set()
   with concurrent.futures.ProcessPoolExecutor(
          max_workers=multiprocessing.cpu_count()) as executor:
      for sourceImage, targetImage in get_jobs(source, target):
         future = executor.submit(scale_one, size, sourceImage,
             targetImage, state)
         future.add_done_callback(report_progress)
         futures.add(future)
         if state.value in {CANCELED, TERMINATING}:
            executor.shutdown()
            for future in futures:
               future.cancel()
            break
      concurrent.futures.wait(futures) # работает, пока не завершится
   if state.value != TERMINATING:
      when_finished()
```

Эта функция исполняется потоком `self.worker`, который создан в методе `Main.Window.scale()`. В ней используется пул процессов (то есть многопроцессная, а не многопоточная обработка), которые и выполняют всю работу. В результате рабочему потоку нужно только вызвать эту функцию, а настоящий труд достанется отдельным процессам.

Для каждого изображения, полученного от функции `ImageScale.get_jobs()`, мы создаем будущий объект, которому предстоит выполнить функцию `ImageScale.scale_one()`, передав ей максимальный размер изображения (`size`), исходный и конечный файл и состояние приложения в виде объекта `Value`.

В предыдущем разделе мы ожидали завершения будущих объектов с помощью функции `concurrent.futures.as_completed()`, а здесь ассоциируем с каждым будущим объектом функцию обратного вызова (метод `Main.Window.report_progress()`) и пользуемся функцией `concurrent.futures.wait()`.

Добавив все будущие объекты, мы проверяем, не прервал ли пользователь приложение, и, если да, то останавливаем пул процессов и отменяем все будущие объекты. По умолчанию функция `concurrent.futures.Executor.shutdown()` возвращает управление немедленно, а ее действие проявляется только после того, как все будущие объекты завершены или отменены.

После того как все будущие объекты созданы, эта функция блокирует выполнение (рабочего потока, а не потока ГИП), дойдя до об-

ращения к `concurrent.futures.wait()`. Это означает, что в случае прерывания программы после создания всех будущих объектов мы должны проверять необходимость отмены при выполнении вызываемого объекта, ассоциированного с каждым будущим объектом (то есть внутри `ImageScale.scale_one()`).

После того как обработка завершена или прервана и при условии, что мы не находимся в процессе выхода из программы, производится обращение к переданной ранее функции обратного вызова `when_finished()`. По выходе из метода `scale()` поток завершается.

```
def get_jobs(source, target):
   for name in os.listdir(source):
      yield os.path.join(source, name), os.path.join(target, name)
```

Эта небольшая функция-генератор порождает 2-кортежи, содержащие полные пути к файлам с исходным и конечным изображениями.

```
Result = collections.namedtuple("Result", "name copied scaled")

def scale_one(size, sourceImage, targetImage, state):
   if state.value in {CANCELED, TERMINATING}:
      raise Canceled()
   oldImage = Image.Image.from_file(sourceImage)
   if state.value in {CANCELED, TERMINATING}:
      raise Canceled()
   if oldImage.width <= size and oldImage.height <= size:
      oldImage.save(targetImage)
      return Result(targetImage, 1, 0)
   else:
      scale = min(size / oldImage.width, size / oldImage.height)
      newImage = oldImage.scale(scale)
      if state.value in {CANCELED, TERMINATING}:
         raise Canceled()
      newImage.save(targetImage)
      return Result(targetImage, 0, 1)
```

Эта функция выполняет собственно масштабирование (или копирование); в ней используется модуль `cyImage` (см. раздел 5.3) или, если он не установлен, модуль `Image` (см. раздел 3.12). Для каждой задачи она возвращает именованный кортеж `Result` или возбуждает исключение `Canceled`, если пользователь решил прервать масштабирование или выйти из программы.

Если пользователь прерывает или завершает программу в середине загрузки, масштабирования или сохранения изображения, то функция доводит начатую операцию до конца. Это означает, что, запросив прерывание или выход, пользователь должен будет дождаться, пока

закончатся *n* операций загрузки, масштабирования или сохранения (где *n* – число процессов в пуле), и только потом запрос будет выполнен. Проверка прерывания или завершения до начала каждой длительной непрерываемой операции – лучшее, что мы можем сделать для обеспечения максимальной отзывчивости приложения.

При каждом возврате результата (или возбуждении исключения `Canceled`) соответствующий будущий объект завершается. А поскольку с каждым будущим объектом ассоциирован вызываемый объект, то этот объект – в данном случае метод `Main.Window.report_progress()` – вызывается.

4.3.3. Как ГИП обрабатывает продвижение

В этом разделе мы рассмотрим методы ГИП, которые сообщают пользователю о ходе выполнения. Они находятся в файле `imagescale/Main.py`.

Поскольку будущие объекты выполняются несколькими процессами, существует возможность, что два или более процессов вызовут метод `report_progress()` одновременно. На самом деле такого не должно случиться, потому что вызываемый объект, ассоциированный с будущим объектом, вызывается в том потоке, в котором создана ассоциация, – в данном случае в рабочем потоке – а, т. к. рабочий поток у нас всего один то, теоретически, метод не может быть вызван более одного раза в каждый момент времени. Однако, это деталь реализации, и опираться на нее было бы неправильно. Поэтому, как бы мы ни хотели работать на высоком уровне, избегая явного использования таких механизмов, как блокировки, в этой ситуации другого выбора нет. Итак, мы создали блокировку, которая гарантирует, что все вызовы метода `report_progress()` сериализованы.

```
ReportLock = threading.Lock()
```

Эта блокировка находится в файле `Main.py` и используется только в одном методе.

```
def report_progress(self, future):
   if self.state.value in {CANCELED, TERMINATING}:
      return
   with ReportLock:   # Вызовы Window.report_progress() сериализуются
      self.total += 1 # равно как обращения к self.total и т. д.
      if future.exception() is None:
         result = future.result()
         self.copied += result.copied
         self.scaled += result.scaled
```

```
            name = os.path.basename(result.name)
            self.statusText.set("{} {}".format(
               "Copied" if result.copied else "Scaled", name))
            self.master.update() # Перерисовываем ГИП
```

Этот метод вызывается при завершении любого будущего объекта – нормально или в результате исключения. Если пользователь прервал выполнение, то мы выходим сразу, потому что об обновлении пользовательского интерфейса позаботится метод `when_finished()`. А если пользователь хочет выйти из программы, то обновлять интерфейс нет смысла, т. к. он все равно исчезнет с экрана.

Большая часть тела метода охвачена блокировкой, поэтому если два или более будущих объектов завершатся в одно и то же время, только один сможет начать выполнение этой части, а все остальные будут ждать освобождения блокировки. (По поводу `self.state Value` беспокоиться не надо, т. к. это синхронизированный тип.) Поэтому код, выполняемый под защитой блокировки, должен быть как можно короче.

Прежде всего, мы увеличиваем на единицу общий счетчик задач. Если будущий объект возбудил исключение (например, `Canceled`), то больше ничего делать не нужно. В противном случае мы увеличиваем счетчики скопированных и масштабированных изображений (на 0 и 1 или на 1 и 0 соответственно) и изменяем текст в метке состояния. Очень важно, чтобы любое обновление ГИП производилось под защитой блокировки. Так мы сможем избежать неопределенного поведения, когда два и более обновлений ГИП делаются одновременно.

```
def when_finished(self):
   self.state.value = IDLE
   self.configure(cursor="arrow")
   self.update_ui()
   result = "Copied {} Scaled {}".format(self.copied, self.scaled)
   difference = self.total - (self.copied + self.scaled)
   if difference: # Этот код выполняется в случае прерывания программы
      result += " Skipped {}".format(difference)
   self.statusText.set(result)
   self.master.update() # Перерисовываем ГИП
```

Этот метод вызывается рабочим потоком, когда тот завершается – вследствие окончания или прерывания обработки, но не в результате выхода из программы. Поскольку этот метод выполняется, когда рабочий поток и все его процессы уже по существу завершились, использовать блокировку `ReportLock` нет нужды. Метод переводит

приложение в состояние IDLE, восстанавливает нормальную форму курсора (стрелка) и с помощью строки состояния сообщает, завершилась ли работа нормально или была прервана.

4.3.4. Как ГИП обрабатывает выход из программы

Чтобы завершить параллельную программу с графическим интерфейсом, просто выйти недостаточно. Нужно сначала попытаться остановить все рабочие потоки – и особенно процессы – чтобы они не остались в памяти, продолжая потреблять ресурсы (к примеру, ту же память).

Выход из программы обрабатывается в методе close() из модуля imagescale/Main.py.

```
def close(self, event=None):
   ...
   if self.worker is not None and self.worker.is_alive():
      self.state.value = TERMINATING
      self.update_ui()
      self.worker.join() # Ждем завершения рабочих потоков
      self.quit()
```

Этот метод вызывается, когда пользователь нажимает кнопку Quit или кнопку закрытия окна × (или клавишу Alt+Q на платформах, отличных от OS X). Он сохраняет кое-какие пользовательские настройки (не показано), а затем проверяет, существует ли еще рабочий поток (так бывает, когда пользователь выходит из программы, не дождавшись конца масштабирования). В последнем случае метод переводит приложение в состояние TERMINATING и обновляет интерфейс, чтобы пользователь видел, что обработка завершается. Процессы, запущенные рабочим потоком, обнаруживают, что состояние изменилось (т. к. периодически опрашивают переменную Value), и, если новое состояние – TERMINATING, то прекращают работу. Обращение к функции threading.Thread.join() блокирует главный поток до тех пор, пока рабочий поток (и его процессы) не завершатся. Если бы мы пренебрегли ожиданием, то могли бы оставить процессы-зомби, которые не делают ничего полезного, а лишь зря занимают память. В самом конце мы вызываем функцию tkinter.ttk.Frame.quit(), которая завершает приложение.

Приложение ImageScale показывает, как, сочетая многопоточность и многопроцессность, написать приложение с ГИП, которое будет ра-

ботать параллельно и при этом без задержки отзываться на действия пользователя. К тому же, выбранная архитектура позволяет поддержать извещение о ходе выполнения и прерывание.

Писать параллельные программы с помощью высокоуровневых механизмов, например потокобезопасных очередей и будущих объектов, гораздо проще, чем прибегая к механизмам среднего и низкого уровня (блокировкам и т. п.). Тем не менее, следует всегда проверять, действительно ли параллельная версия по производительности превосходит последовательную. Например, программируя на Python, не стоит применять многопоточность в счетных задачах.

Следует также остерегаться изменяемых разделяемых данных. Передавать в качестве параметров следует только неизменяемые данные (например, числа и строки). А если передаются изменяемые данные, то только для чтения (изменения можно производить, например, до входа в распараллеленный участок) или глубокие копии. Однако, изучая программу ImageScale, мы все же столкнулись с необходимостью использовать разделяемые данные. К счастью, с помощью управляемого класса `multiprocessing.Value` (или `multiprocessing.Array`) это можно сделать, не прибегая к явным блокировкам. Вместо этого можно было бы создать собственные потокобезопасные классы. Соответствующий пример будет приведен в главе 6 (раздел 6.2.1.1).

ГЛАВА 5. Расширение Python

Python – достаточно быстрый язык для большинства программ. В тех же случаях, когда это не так, мы зачастую можем достичь необходимого ускорения с помощью распараллеливания, как было показано в предыдущем разделе. Иногда, впрочем, и этого недостаточно, и нужна по-настоящему высокая скорость обработки. Существует три основных способа ускорить программы на Python: воспользоваться интерпретатором PyPy (`pypy.org`), в котором имеется встроенный JIT-компилятор; написать критические участки программы на C или C++; транслировать код на Python (или Cython) на C с помощью Cython[1].

После установки PyPy мы можем исполнять Python-программы с помощью интерпретатора PyPy, а не стандартного интерпретатора CPython. Это дает существенное ускорение долго работающих программ, поскольку затраты на JIT-компиляцию в этом случае амортизируются на все время выполнения, но короткие программы наоборот могут работать медленнее.

Чтобы использовать код на C или C++, написанный нами или входящий в сторонние библиотеки, необходимо сделать его доступным Python-программам. Тем, кто хочет писать код на C или C++ самостоятельно, имеет прямой смысл воспользоваться интерфейсом между Python и C (`docs.python.org/3/extending`). У тех же, кто желает использовать уже написанный код на C или C++, есть несколько возможностей. Первая – воспользоваться оберткой, которая берет код на C или C++ и порождает интерфейс к нему из Python. Два популярных инструмента такого рода – SWIG (`www.swig.org`) и SIP (`www.riverbankcomputing.co.uk/software/sip`). Вторая, рассчитанная на C++, – обратиться к библиотеке `boost::python` (`www.boost.org/libs/python/doc/`). Недавно на этом поле появился новый

[1] Сейчас появляются новые компиляторы Python, например Numba (`numba.pydata.org`) и Nuitka (`nuitka.net`)

игрок – CFFI (C Foreign Function Interface for Python – интерфейс внешних C-функций для Python), который, несмотря на свою молодость, уже используется в давно зарекомендовавшем себя проекте PyPy (`bitbucket.org/cffi/cffi`).

Расширение Python в OS X и Windows

Хотя примеры в этой главе проверялись только в Linux, они должны нормально работать также в OS X и Windows (для многих программистов, использующих `ctypes` и Cython, – это основные платформы разработки). Однако, возможно, понадобятся кое-какие платформенно-зависимые корректировки. Дело в том, что во многих системах на основе Linux уже установлен компилятор GCC и системные библиотеки, рассчитанные на размер слова, соответствующий архитектуре машины. В системах же на платформе OS X и Windows ситуация обычно несколько сложнее или, по крайней мере, иная.

В OS X и Windows необходимо, чтобы компилятор и длина слова (32 или 64 разряда), использованные при сборке Python, были согласованы с внешними разделяемыми библиотеками (файлами с расширениями `.dylib` или `.DLL`) или с инструментами, используемыми для компиляции Cython-кода. В OS X может быть установлен компилятор GCC, но в настоящее время чаще используется Clang; в Windows это может быть та или иная версия GCC или коммерческий компилятор, например, производства Microsoft. Кроме того, в OS X и Windows разделяемые библиотеки часто находятся в каталогах приложений, а не в системных каталогах, а файлы-заголовки надо получать отдельно. Поэтому, чтобы не увязнуть в болоте зависящей от платформы и компилятора информации (которая к тому же быстро устаревает с выходом новых версий), мы сосредоточимся на том, как *использовать* `ctypes` и Cython, оставляя читателям, работающим в системах, отличных от Linux, самим разбираться в детальных требованиях, когда они захотят воспользоваться этими технологиями.

Все возможности, которые мы описывали до сих пор, заслуживают подробного изучения, но в этой главе мы займемся двумя другими технологиями: пакетом `ctypes`, который входит в стандартную библиотеку Python (`docs.python.org/3/library/ctypes.html`) и программой Cython (`cython.org`). Обе можно использовать для реализации интерфейса между Python и своим собственным или сторонним кодом на C и C++, а Cython, кроме того, позволяет транслировать Python и Cython-код на C с целью повышения производительности – иногда это дает сногсшибательные результаты.

5.1. Доступ к написанным на C библиотекам с помощью пакета ctypes

Стандартный пакет `ctypes` дает доступ к коду, написанному на C или C++ (да и вообще на любом компилируемом языке, следующем принятым в C соглашениям о вызове) коду, который представлен в виде автономной разделяемой библиотеки (`.so` в Linux, `.dylib` в OS X, `.DLL` в Windows).

В этом разделе и в первом подразделе следующего мы напишем модуль, который предоставляет Python-программе доступ к некоторым C-функциям в разделяемой библиотеке. Мы будем работать с библиотекой `libhyphen.so`, которая в некоторых системах называется `libhyphen.uno.so` (см. врезку «Расширение Python в OS X и Windows»). Обычно эта библиотека входит в состав OpenOffice.org или LibreOffice и содержит функцию, которая получает слово и возвращает копию этого слова, в котором во всех местах, где возможен перенос, стоят дефисы. Хотя на первый взгляд кажется, что это простая задача, сигнатура функции весьма сложна (почему она и является идеальным примером применения `ctypes`). И на самом деле нам придется использовать три функции: одна загружает словарь переносов, вторая расставляет переносы, а третья освобождает ресурсы после завершения.

Обычный порядок работы с пакетом `ctypes` таков: загрузить библиотеку в память, получить ссылки на необходимые функции, затем вызывать их. Модуль `Hyphenate1.py` так и устроен. Но сначала посмотрим, как этот модуль используется. Ниже показан интерактивный сеанс в оболочке Python (например, IDLE):

```
>>> import os
>>> import Hyphenate1 as Hyphenate
>>>
>>> # Находим файлы hyph*.dic
>>> path = "/usr/share/hyph_dic"
>>> if not os.path.exists(path): path = os.path.dirname(__file__)
>>> usHyphDic = os.path.join(path, "hyph_en_US.dic")
>>> ruHyphDic = os.path.join(path, "hyph_ru_RU.dic")
>>>
>>> # Создаем обертки, чтобы не нужно было всякий раз указывать словарь
>>> hyphenate = lambda word: Hyphenate.hyphenate(word, usHyphDic)
>>> hyphenate_ru = lambda word: Hyphenate.hyphenate(word, ruHyphDic)
>>>
```

```
>>> # Используем обертки
>>> print(hyphenate("extraordinary"))
ex-traor-di-nary
>>> print(hyphenate_ru("поразительный"))
по-ра-зи-тель-ный
```

Единственная функция, используемая вне самого модуля, – `Hyphenate1.hyphenate()`, она обращается к библиотечной функции. Внутри же модуля есть еще две закрытые функции, обращающиеся к двум другим библиотечным функциям. Кстати говоря, словари переносов представлены в формате наборной системы с открытым исходным кодом TEX.

Весь код находится в файле `Hyphenate1.py`. Ниже приведены сигнатуры библиотечных функций, к которым мы обращаемся.

```
HyphenDict *hnj_hyphen_load(const char *filename);

void hnj_hyphen_free(HyphenDict *hdict);

int hnj_hyphen_hyphenate2(HyphenDict *hdict, const char *word,
   int word_size, char *hyphens, char *hyphenated_word, char
***rep,
   int **pos, int **cut);
```

Эти сигнатуры взяты из файла-заголовка `hyphen.h`. Символом `*` в C и C++ обозначается *указатель*. В указателе хранится адрес блока памяти, то есть непрерывного участка байтов. Размер блока может быть любым, в том числе всего один байт. 64-разрядное число представляется блоком из 8 байтов. В строках на один символ отводится от 1 до 4 байтов (в зависимости от кодировки) плюс фиксированные накладные расходы.

Первая функция, `hnj_hyphen_load()`, принимает имя файла в виде указателя на блок `char` (байтов). Это должен быть словарь переносов в формате TEX. Функция возвращает указатель на структуру `HyphenDict` – составной объект (похожий на экземпляр класса Python). К счастью, нам не нужно ничего знать о внутреннем устройстве `HyphenDict`, поскольку мы лишь передаем указатель на нее из одной функции в другую.

В языке C функции, принимающие *C-строки* – указатели на блоки символов, или байтов – обычно следуют одному из двух соглашений: либо передается только указатель и тогда предполагается, что строка заканчивается байтом `0x00` (`'\0'`) (говорят, что C-строка завершается нулем), либо указатель и счетчик байтов. Функция `hnj_hyphen_load()` принимает только указатель, поэтому переданная ей строка

должна завершаться нулем. Как мы увидим, если функции `ctypes.create_string_buffer()` передать `str`, то она вернет эквивалентную C-строку, завершающуюся нулем.

Всякий загруженный словарь переносов в конечном итоге необходимо освободить (если этого не сделать, то библиотека расстановки переносов останется в памяти, хотя необходимость в ней уже отпала). Вторая функция, `hnj_hyphen_free()`, принимает указатель на `HyphenDict` и освобождает связанные со словарем ресурсы. Она не возвращает значение. После освобождения указатель уже нельзя использовать – точно так же, как нельзя использовать переменную Python, удаленную оператором `del`.

Третья функция, `hnj_hyphen_hyphenate2()`, как раз и осуществляет расстановку переносов. Аргумент `hdict` – это указатель на словарь `HyphenDict`, полученный от функции `hnj_hyphen_load()` (и еще не освобожденный функцией `hnj_hyphen_free()`). Аргумент `word` – слово, в котором мы хотим расставить переносы, переданное в виде указателя на блок байтов в кодировке UTF-8. Аргумент `word_size` – количество байтов в этом блоке. Указатель на блок байтов `hyphens` мы использовать не будем, тем не менее, это должен быть корректный указатель, иначе функция работать не будет. `hyphenated_word` – указатель на блок байтов, достаточно длинный для хранения исходного слова со вставленными дефисами (на самом деле, библиотека вставляет не дефисы, а знаки =). Этот блок должен быть инициализирован байтами `0x00`. `rep` – указатель на указатель на указатель на блок байтов; нам он не понадобится, но передать его нужно. Аргументы `pos` и `cut` – указатели на целые (`int`), которые нам тоже не интересны, но должны присутствовать. Функция возвращает флаг, равный 1, если была ошибка, и 0 в случае успеха.

Теперь, зная, что предстоит обернуть, обратимся к модулю `Hyphenate1.py` (предложения импорта, как обычно, опущены) и начнем с поиска и загрузки разделяемой библиотеки расстановки переносов.

```
class Error(Exception): pass

   _libraryName = ctypes.util.find_library("hyphen")
   if _libraryName is None:
      _libraryName = ctypes.util.find_library("hyphen.uno")
   if _libraryName is None:
      raise Error("cannot find hyphenation library")

   _LibHyphen = ctypes.CDLL(_libraryName)
```

Сначала создается класс исключения `Hyphenate1.Error`, чтобы пользователи модуля могли отличить исключения, относящиеся к этому модулю, от более общих, например `ValueError`. Функция `ctypes.util.find_library()` ищет разделяемую библиотеку. В Linux ее имя начинается префиксом `lib` и заканчивается расширением `.so`, поэтому первое предложение ищет файл `libhyphen.so` в различных стандартных местах. В OS X оно будет искать файл `hyphen.dylib`, а в Windows – `hyphen.dll`. Иногда библиотека называется `libhyphen.uno.so`, поэтому в случае неудачи мы пытаемся найти ее под таким именем. Если и это не удается, то мы сдаемся и возбуждаем исключение.

Если библиотека найдена, мы загружаем ее в память с помощью функции `ctypes.CDLL()` и запоминаем ссылку на нее в закрытой переменной `_LibHyphen`. Для тех, кто собирается писать только программы для Windows с использованием специфичных для Windows интерфейсов, имеются функции `ctypes.OleDLL()` и `ctypes.WinDLL()`, которые загружают библиотеки, содержащие Windows API.

Загрузив библиотеку, мы можем создать Python-обертки интересных нам функций. Обычно для этого запоминают ссылку на библиотечную функцию в переменной Python, а затем указывают типы аргументов (в виде списка типов `ctypes`) и тип возвращаемого значения (в виде одиночного типа `ctypes`).

Если указать неверное количество или неверные типы аргументов или неверный тип возвращаемого значения, то наша программа грохнется! Пакет CFFI (`bitbucket.org/cffi/cffi`) в этом отношении менее привередлив; к тому же, он лучше, чем `ctypes`, работает с интерпретатором PyPy (`pypy.org`).

```
_load = _LibHyphen.hnj_hyphen_load
_load.argtypes = [ctypes.c_char_p] # const char *filename
_load.restype = ctypes.c_void_p    # HyphenDict *
```

Здесь мы создали закрытую функцию модуля `_load()`, которая вызывает стоящую за ней библиотечную функцию `hnj_hyphen_load()`. Имея ссылку на библиотечную функцию, мы должны задать типы ее аргументов и возвращаемого значения. В данном случае есть всего один аргумент (C-типа `const char *`), который можно представить непосредственно типом `ctypes.c_char_p` («C-указатель на символ»). Эта функция возвращает указатель на структуру `HyphenDict`. Для ее представления можно было бы создать класс, наследующий `ctypes.Structure`. Но поскольку нам предстоит только передавать этот ука-

затель, а не обращаться к тому, на что он указывает, то можно просто объявить, что функция возвращает тип `ctypes.c_void_p` («C-указатель на void»), который может указывать на все, что угодно.

Эти три строки (помимо приведенного выше кода поиска и загрузки библиотеки) – все, что нужно для написания метода `_load()`, загружающего библиотеку расстановки переносов.

```
_unload = _LibHyphen.hnj_hyphen_free
_unload.argtypes = [ctypes.c_void_p] # HyphenDict *hdict
_unload.restype = None
```

Этот код устроен так же, как предыдущий. Функция `hnj_hyphen_free()` принимает один аргумент – указатель на структуру `HyphenDict struct`, но поскольку мы лишь передаем такие указатели, то можем без опаски описать его как указатель на `void` – при условии, конечно, что на самом деле передается указатель на структуру `HyphenDict`. Эта функция ничего не возвращает, поэтому мы говорим, что `restype` для нее равен `None` (если `restype` не указан, предполагается, что функция возвращает `int`).

```
_int_p = ctypes.POINTER(ctypes.c_int)
_char_p_p = ctypes.POINTER(ctypes.c_char_p)
_hyphenate = _LibHyphen.hnj_hyphen_hyphenate2
_hyphenate.argtypes = [
      ctypes.c_void_p,    # HyphenDict *hdict
      ctypes.c_char_p,    # const char *word
      ctypes.c_int,       # int word_size
      ctypes.c_char_p,    # char *hyphens [не используется]
      ctypes.c_char_p,    # char *hyphenated_word
      _char_p_p,          # char ***rep [не используется]
      _int_p,             # int **pos [не используется]
      _int_p]             # int **cut [не используется]
_hyphenate.restype = ctypes.c_int # int
```

Это самая сложная из всех обертываемых функций. Аргумент `hdict` – указатель на структуру `HyphenDict`, который мы описываем как C-указатель на `void`. Затем идет аргумент `word` – слово, в котором нужно расставить переносы, передаваемое в виде C-указателя на символ. За ним следует `word_size` – количество байтов в слове, представленное в виде целого (`ctypes.c_int`). Далее идет не нужный нам буфер `hyphens`, а за ним – `hyphenated_word`, слово с расставленными переносами, опять же представленное в виде C-указателя на символ. Для указателя на указатель на символ (байт) в `ctypes` нет встроенного типа, поэтому мы создали собственный тип `_char_p_p`, определив

его как указатель на указатель на C-символ. То же самое мы проделали для двух указателей на указатели на целые.

Строго говоря, определять `restype` нет необходимости, потому что функция возвращает целое значение, но мы предпочитаем указывать все явно.

Мы создали закрытые функции-обертки для функций из библиотеки расстановки переносов, поскольку хотим экранировать пользователей нашего модуля от низкоуровневых деталей. С этой целью мы предоставляем единственную открытую функцию `hyphenate()`, которая принимает исходное слово, нужный словарь переносов и символ, которым будет обозначаться перенос. Для повышения эффективности каждый словарь переносов загружается только один раз. И разумеется, мы не забываем освободить все загруженные словари при выходе из программы.

```
def hyphenate(word, filename, hyphen="-"):
   originalWord = word
   hdict = _get_hdict(filename)
   word = word.encode("utf-8")
   word_size = ctypes.c_int(len(word))
   word = ctypes.create_string_buffer(word)
   hyphens = ctypes.create_string_buffer(len(word) + 5)
   hyphenated_word = ctypes.create_string_buffer(len(word) * 2)
   rep = _char_p_p(ctypes.c_char_p(None))
   pos = _int_p(ctypes.c_int(0))
   cut = _int_p(ctypes.c_int(0))
   if _hyphenate(hdict, word, word_size, hyphens, hyphenated_word, rep,
           pos, cut):
      raise Error("hyphenation failed for '{}'".format(originalWord))
   return hyphenated_word.value.decode("utf-8").replace("=", hyphen)
```

Сначала эта функция сохраняет ссылку на переданное ей слово, чтобы можно было включить его в сообщение об ошибке, если понадобится. Затем мы получаем словарь переносов – закрытая функция `_get_hdict()` возвращает указатель на структуру `HyphenDict`, соответствующую файлу с указанным именем. Если словарь уже загружен, то возвращается указатель, запомненный в момент загрузки; в противном случае словарь загружается в первый и единственный раз, указатель на него запоминается для будущего использования и возвращается вызывающей стороне.

Слово передается функции расстановки переносов в виде блока байтов в кодировке UTF-8, который легко получить с помощью метода `str.encode()`. Необходимо также передать количество байтов в слове, мы вычисляем его и преобразуем тип Python `int` в тип C `int`.

Нельзя передавать C-функции объект Python `bytes`, поэтому мы создаем строковый буфер (на самом деле, блок C-символов – данных типа `char`), содержащий байты слова. Функция `ctypes.create_string_buffer()` создает блок C-символов, соответствующий объекту `bytes`, или просто заданного размера. Хотя аргумент `hyphens` нам не нужен, его все равно необходимо подготовить, а в документации сказано, что это должен быть указатель на блок C-символов длиной на пять символов больше, чем длина слова (в байтах). Поэтому мы создаем блок символов подходящего размера. Слово с расставленными переносами будет помещено в блок C-символов, переданный функции, поэтому мы должны создать блок достаточного размера. Документация рекомендует брать размер в два раза больше, чем размер исходного слова.

Мы не собираемся использовать аргументы `rep`, `pos` и `cut`, но должны передать корректные значения, иначе функция не будет работать. `rep` – указатель на указатель на указатель на `char`, поэтому мы создали указатель на пустой блок (нулевой указатель в C, то есть указатель, который ни на что не указывает), а затем присвоили указатель на указатель на этот указатель переменной `rep`. В случае аргументов `pos` и `cut` мы создали указатели на указатели на целые, равные 0.

После того как все аргументы подготовлены, мы вызываем закрытую функцию `_hyphenate()` (в результате чего вызывается библиотечная функция `hnj_hyphen_hyphenate2()`) и возбуждаем исключение, если эта функция возвращает ненулевое значение, свидетельствующее об ошибке. В противном случае мы извлекаем байты слова с расставленными переносами с помощью свойства `value` (которое возвращает объект `bytes`, в котором последний байт равен `0x00`). Затем мы декодируем байты, применяя кодировку UTF-8, и получаем строку `str`, после чего остается заменить вставленные библиотекой знаки = указанным пользователем символом переноса (по умолчанию -). Результирующая строка возвращается в качестве значения функции `hyphenate()`.

Отметим, что для C-функций, которые принимают аргумент типа `char *` и размер, а не заканчивающуюся нулем строку, получить байты можно с помощью свойства `raw`, а не `value`.

```
_hdictForFilename = {}

def _get_hdict(filename):
   if filename not in _hdictForFilename:
      hdict = _load(ctypes.create_string_buffer(
             filename.encode("utf-8")))
```

```
      if hdict is None:
         raise Error("failed to load '{}'".format(filename))
      _hdictForFilename[filename] = hdict
   hdict = _hdictForFilename.get(filename)
   if hdict is None:
      raise Error("failed to load '{}'".format(filename))
   return hdict
```

Эта закрытая вспомогательная функция возвращает указатель на структуру `HyphenDict`; если словарь был уже загружен, то возвращается ранее запомненный указатель.

Если имя файла отсутствует в Python-словаре `_hdictForFilename`, значит, это новый словарь переносов, который необходимо загрузить. Поскольку имя файла передается в виде C-типа `const char *` (неизменяемого), то мы можем создать из него строковый буфер `ctypes` непосредственно. Если функция `_load()` возвращает `None`, сигнализируя об ошибке при загрузке, то мы возбуждаем исключение. В противном случае указатель сохраняется для последующего использования.

В конце функции мы возвращаем указатель на словарь – вне зависимости от того, был ли он загружен только что или раньше.

```
def _cleanup():
   for hyphens in _hdictForFilename.values():
      _unload(hyphens)

atexit.register(_cleanup)
```

Python-словарь `_hdictForFilename` содержит указатели на все загруженные словари переносов. Мы должны освободить их перед выходом из программы. Для этого мы создали закрытую функцию `_cleanup()`, которая вызывает написанную нами закрытую функцию `_unload()` для каждого хранящегося указателя на словарь переносов (а эта функция уже вызывает библиотечную функцию `hnj_hyphen_free()`). Нам нет нужды в конце очищать сам словарь `_hdictForFilename`, потому что функция `_cleanup()` вызывается только перед выходом из программы (так что словарь в любом случае будет освобожден). Чтобы функция `_cleanup()` гарантированно вызывалась, мы регистрируем ее как «функцию при выходе», пользуясь функцией `register()` из стандартного модуля `atexit`.

Итак, мы рассмотрели весь код, необходимый для написания функции `hyphenate()` в модуле, который предоставляет доступ к библиотеке расстановки переносов. Работа с пакетом `ctypes` требу-

ет аккуратности (в частности, при задании типов аргументов и инициализации аргументов), зато открывает программам на Python весь мир богатейшей функциональности, написанной на C и C++. Одно из практических применений `ctypes` – кодирование на C или C++ критических с точки зрения быстродействия участков кода, которые должны находиться в разделяемой библиотеке, чтобы их можно было использовать как из Python (с помощью `ctypes`), так и непосредственно из программ на C или C++. Другое применение – доступ к сторонним разделяемым библиотекам, написанным на C или C++, хотя в большинстве случаев уже существуют стандартные или написанным кем-то другим модули, которые обертывают интересующую нас библиотеку.

Функциональность модуля `ctypes` куда богаче, но в этой книге просто нет места представить ее в полном объеме. И хотя работать с этим пакетом труднее, чем с CFFI или Cython, в некоторых случаях он все же удобнее, поскольку входит в стандартную библиотеку Python.

5.2. Использование Cython

На сайте Cython (`cython.org`) говорится, что это язык программирования, «который делает написание расширений языка Python на C таким же простым делом, как сам язык Python». Cython можно использовать тремя способами. Во-первых, для обертывания кода на C или C++, как при работе с `ctypes`, хотя Cython, пожалуй, проще, особенно для тех, кто хорошо знаком с C или C++. Во-вторых, для трансляции Python-кода в быстрый код на C. Для этого всего-то и нужно, что заменить расширение `.py` на `.pyx` и откомпилировать модуль. Этого обычно достаточно для двукратного ускорения счетных задач. Третий способ похож на второй, но вместо того чтобы оставить код в `pyx`-файле, мы его *Cython*'изируем, то есть пользуемся предоставляемыми Cython расширениями языка, чтобы после компиляции получить гораздо более эффективный C-код. В этом случае можно добиться ускорения счетных задач в 100 и более раз.

5.2.1. Доступ к написанным на C библиотекам с помощью Cython

В этом подразделе мы напишем модуль `Hyphenate2`, который предоставляет те же возможности, что модуль `Hyphenate1.py` из пре-

дыдущего раздела, только с использованием Cython вместо `ctypes`. В решении на основе `ctypes` нам был нужен единственный файл `Hyphenate1.py`, а в случае Cython придется создать каталог и в нем четыре файла.

Первым делом нам понадобится файл `Hyphenate2/setup.py`. В этом крохотном инфраструктурном файле будет всего одно предложение, которое сообщает Cython, где искать библиотеку расстановки переносов и что строить. Второй файл – `Hyphenate2/__init__.py`. Этот необязательный файл содержит одно предложение, в котором экспортируются открытая функция `Hyphenate2.hyphenate()` и исключение `Hyphenate2.Error`. Третий файл – тоже совсем небольшой – `Hyphenate2/chyphenate.pxd`. Он сообщает Cython о библиотеке расстановки переносов и функциях из нее, которые мы собираемся использовать. Четвертый файл – `Hyphenate2/Hyphenate.pyx`. Это модуль Cython, в котором будет находиться реализация открытой функции `hyphenate()` и закрытых вспомогательных функций.

```
distutils.core.setup(name="Hyphenate2",
      cmdclass={"build_ext": Cython.Distutils.build_ext},
      ext_modules=[distutils.extension.Extension("Hyphenate",
           ["Hyphenate.pyx"], libraries=["hyphen"])])
```

Это все содержимое файла `Hyphenate2/setup.py`, за исключением предложений импорта. Здесь используется пакет `distutils`[2]. Атрибут `name` необязателен. Атрибут `cmdclass` необходимо задать точно так, как написано. Первый аргумент функции `Extension()` – имя будущего откомпилированного модуля (например, `Hyphenate.so`). За ним следует список `pyx`-файлов, содержащих подлежащий компиляции код, и далее необязательный список внешних библиотек на C или C++. Разумеется, для данного примера необходима библиотека `hyphen`.

Чтобы построить расширение, выполните следующую команду, находясь в каталоге, содержащем все описанные выше файлы (например, `Hyphenate2`):

```
$ cd pipeg/Hyphenate2
$ python3 setup.py build_ext --inplace
running build_ext
```

[2] Пожалуй, лучше установить пакет Python `distribute` версии ≥ 0.6.28 или – еще лучше – пакет `setuptools` версии ≥ 0.7 (`python-packaging-user-guide.readthedocs.org`). Современный инструмент управления пакетами необходим для установки многих сторонних пакетов, в том числе некоторых описываемых в этой книге.

```
cythoning Hyphenate.pyx to Hyphenate.c
building 'Hyphenate' extension
creating build
creating build/temp.linux-x86_64-3.3
...
```

Если на машине установлено несколько интерпретаторов Python, то нужно указать полный путь к тому, который мы хотим использовать. В версии Python 3.1 будет создан `Hyphenate.so`, а в более поздних – разделяемая библиотека, зависящая от номера версии, например, `Hyphenate.cpython-33m.so` в случае Python 3.3.

```
from Hyphenate2.Hyphenate import hyphenate, Error
```

Это все, что есть в файле `Hyphenate2/__init__.py`. Он просто обеспечивает небольшое дополнительное удобство, чтобы пользователь мог написать, скажем, `import Hyphenate2 as Hyphenate`, а затем `Hyphenate.hyphenate()`. Без него пришлось бы писать `import Hyphenate2.Hyphenate as Hyphenate`.

```
cdef extern from "hyphen.h":
   ctypedef struct HyphenDict:
      pass

   HyphenDict *hnj_hyphen_load(char *filename)
   void hnj_hyphen_free(HyphenDict *hdict)
   int hnj_hyphen_hyphenate2(HyphenDict *hdict, char *word,
        int word_size, char *hyphens, char *hyphenated_word,
        char ***rep, int **pos, int **cut)
```

Это содержимое файла `Hyphenate2/chyphenate.pxd`. Такой файл необходим для доступа к внешним разделяемым библиотекам на C или C++ из кода, написанного на Cython.

В первой строке объявлено имя файла-заголовка C/C++, в котором находятся объявления интересующих нас функций и типов. Далее в теле объявляются эти функции и типы. Cython предлагает удобный способ ссылаться на структуру C/C++, не объявляя ее во всех подробностях. Но это разрешено, только если мы просто передаем указатель на структуру, а не обращаемся к ее полям самостоятельно; такая ситуация встречается часто, в том числе для библиотеки расстановки переносов. Объявления функций по существу просто скопированы из заголовка C/C++, мы лишь убрали из них завершающие точки с запятой.

Cython использует `pxd`-файл, чтобы создать мост между C-кодом, получающимся в результате компиляции, и внешней библиотекой, на которую ссылается `pxd`-файл.

Теперь, имея файлы `setup.py`, `__init__.py` и `chyphenate.pxd`, мы готовы создать последний файл: `Hyphenate.pyx`. Он содержит Cython-код, то есть код на Python с расширениями Cython. Начнем с предложений импорта, а затем разберемся, что делают отдельные функции.

```
import atexit
cimport chyphenate
cimport cpython.pycapsule as pycapsule
```

Стандартный библиотечный модуль `atexit` необходим для освобождения загруженных словарей переносов перед выходом из программы.

В Cython-файлы можно импортировать обычные Python-модули с помощью предложения `import`, а также pxd-файлы Cython (обертки внешних C-библиотек) с помощью предложения `cimport`. Таким образом, здесь мы импортируем файл `chyphenate.pxd` в виде модуля `chyphenate`, в результате чего получаем тип `chyphenate.HyphenDict` и три функции из библиотеки расстановки переносов.

Мы хотим создать Python-словарь, ключами которого являются имена файлов, содержащих словари переносов, а значениями – указатели на объекты `chyphenate.HyphenDict`. Однако в словарях Python невозможно хранить указатели (поскольку в Python нет такого типа данных). По счастью, Cython предлагает решение: `pycapsule`. Этот модуль Cython умеет инкапсулировать указатель в виде объекта Python, а уж инкапсулирующий объект, конечно, можно сохранить в любой коллекции Python. Как мы увидим ниже, `pycapsule` позволяет также извлечь указатель из объекта Python.

```
def hyphenate(str word, str filename, str hyphen="-"):
   cdef chyphenate.HyphenDict *hdict = _get_hdict(filename)
   cdef bytes bword = word.encode("utf-8")
   cdef int word_size = len(bword)
   cdef bytes hyphens = b"\x00" * (word_size + 5)
   cdef bytes hyphenated_word = b"\x00" * (word_size * 2)
   cdef char **rep = NULL
   cdef int *pos = NULL
   cdef int *cut = NULL
   cdef int failed = chyphenate.hnj_hyphen_hyphenate2(hdict, bword,
          word_size, hyphens, hyphenated_word, &rep, &pos, &cut)
   if failed:
      raise Error("hyphenation failed for '{}'".format(word))
   end = hyphenated_word.find(b"\x00")
   return hyphenated_word[:end].decode("utf-8").replace("=", hyphen)
```

Эта функция структурно аналогична версии для ctypes, написанной в предыдущем разделе (стр. 212). Наиболее заметное различие состоит в том, что все аргументы и переменные имеют значения по умолчанию. Это не *требование* Cython, но при такой организации Cython может выполнить некоторые оптимизации для повышения производительности.

hdict – это указатель на структуру HyphenDict, а в переменной bword хранятся представленные в кодировке UTF-8 байты слова, в котором мы хотим расставить переносы. Значение переменной word_size типа int легко вычисляется. Для параметра hyphens, который мы не собираемся использовать, все равно нужно создать достаточно большой буфер (блок C-символов), для этого естественно повторить нулевой байт требуемое число раз. Такая же техника применяется для создания буфера hyphenated_word.

Мы не используем аргументы rep, pos и cut, но тем не менее их значения должны быть корректны, иначе функция работать не будет. Все три аргумента создаются с помощью синтаксиса указателей C (то есть cdef char **rep), причем уровень косвенности на единицу меньше, чем нужно (то есть меньше звездочек *). Затем при вызове функции мы используем C-оператор взятия адреса (&), чтобы передать адрес аргумента, и тем самым добавляем недостающий уровень косвенности. В качестве этих аргументов нельзя передать просто нулевой указатель (NULL), потому что функция ожидает получить настоящий указатель, даже если указывать он будет на NULL. Напомним, что в C NULL – это указатель, не указывающий ни на что.

Инициализировав все аргументы, мы вызываем функцию, которая экспортируется из Cython-модуля chyphenate (на самом деле, из файла chyphenate.pxd). Если при расстановке переносов произойдет ошибка, то мы возбуждаем обычное исключение Python. Если же все пройдет нормально, то мы возвращаем слово с расставленными переносами. Для этого мы должны вырезать из буфера hyphenated_word всё до первого нулевого байта, затем декодировать вырезанные байты, применяя кодировку UTF-8, чтобы получить строку, и, наконец, заменить поставленные библиотекой знаки = заданными пользователем символами переноса (по умолчанию –).

```
_hdictForFilename = {}

cdef chyphenate.HyphenDict *_get_hdict(
        str filename) except <chyphenate.HyphenDict*>NULL:
   cdef bytes bfilename = filename.encode("utf-8")
   cdef chyphenate.HyphenDict *hdict = NULL
   if bfilename not in _hdictForFilename:
```

```
        hdict = chyphenate.hnj_hyphen_load(bfilename)
        if hdict == NULL:
            raise Error("failed to load '{}'".format(filename))
        _hdictForFilename[bfilename] = pycapsule.PyCapsule_New(
            <void*>hdict, NULL, NULL)
    capsule = _hdictForFilename.get(bfilename)
    if not pycapsule.PyCapsule_IsValid(capsule, NULL):
        raise Error("failed to load '{}'".format(filename))
    return <chyphenate.HyphenDict*>pycapsule.PyCapsule_GetPointer(capsule,
            NULL)
```

Эта закрытая функция определена с помощью ключевого слова `cdef`, а не `def`; это означает, что мы имеем дело с функцией, написанной на Cython, а не на Python. После `cdef` задается тип возвращаемого функцией значения, в данном случае указатель на `chyphenate.HyphenDict`. Затем мы как обычно задаем имя функции и аргументы, обычно с типами. В этом примере существует всего один строковый аргумент – имя файла (`filename`).

Поскольку возвращается указатель, а не объект Python (то есть `object`), то уведомить вызывающую сторону об исключении обычным способом невозможно. На самом деле, при возникновении исключения будет просто напечатано сообщение об ошибке, а исключение как таковое будет проигнорировано. Однако мы хотим, чтобы эта функция могла возбуждать исключения Python. Для этого мы в типе возвращаемого значения указали, какое значение следует трактовать как исключение; в данном случае это нулевой указатель на `chyphenate.HyphenDict`.

В начале мы объявляем указатель на `chyphenate.HyphenDict`, инициализируя его значением `NULL` (то есть этот указатель не указывает ни на что). Затем мы смотрим, есть ли переданное имя файла в словаре `_hdictForFilename`. Если нет, то мы должны загрузить новый словарь переносов с помощью библиотечной функции `hnj_hyphen_load()`, доступной через наш модуль `chyphenate`. В случае успешной загрузки мы получаем ненулевой указатель `chyphenate.HyphenDict`, приводим его к типу `void*` (указатель на все что угодно) и создаем объект `pycapsule.PyCapsule` для его хранения. Синтаксис *<type>* используется в Cython для приведения значения одного C-типа к другому C-типу. Например, `<int>(x)` преобразует значение *x* (которое должно быть числом или C-символом – `char`) к C-типу `int`. Это напоминает синтаксис Python `int(x)` с тем отличием, что в Python *x* может иметь Python-тип `int` или `float` или являться строкой, представляющей числовое значение (например, `"123"`), а возвращается значение Python-типа `int`.

Второй аргумент `pycapsule.PyCapsule_New()` – имя, присваиваемое инкапсулированному указателю (скажем, C-указателю `char *`), а третий – указатель на функцию-деструктор. Нам ни то, ни другое не нужно, поэтому мы передаем нулевые указатели. После этого инкапсулированный указатель записывается в словарь как значение, соответствующее имени файла.

В конце мы пытаемся получить капсулу, содержащую указатель на словарь переносов – вне зависимости от того, был ли он загружен сейчас или раньше. Мы должны проверить, что капсула содержит действительный (ненулевой) указатель, передав саму капсулу и ассоциированное с ней имя функции `pycapsule.PyCapsule_IsValid()`. В качестве имени мы передаем `NULL`, потому что не присваивали капсулам имен. Если капсула действительна, то мы извлекаем из нее указатель с помощью функции `pycapsule.PyCapsule_GetPointer()` – и на этот раз передавая функции капсулу и `NULL` вместо имени – и приводим указатель из типа `void*` к типу указателя на `chyphenate.HyphenDict`. Результат приведения возвращается в качестве результата функции.

```
def _cleanup():
   cdef chyphenate.HyphenDict *hdict = NULL
   for capsule in _hdictForFilename.values():
      if pycapsule.PyCapsule_IsValid(capsule, NULL):
         hdict = (<chyphenate.HyphenDict*>
                  pycapsule.PyCapsule_GetPointer(capsule, NULL))
         if hdict != NULL:
            chyphenate.hnj_hyphen_free(hdict)

atexit.register(_cleanup)
```

Непосредственно перед завершением программы вызываются все функции, зарегистрированные с помощью `atexit.register()`. В данном случае будет вызвана закрытая функция `_cleanup()` из нашего модуля. Сначала мы объявляем указатель на `chyphenate.HyphenDict`, инициализированный нулем. Затем обходим все значения в словаре `_hdictForFilename`, каждое из которых является капсулой, содержащей указатель на безымянный `chyphenate.HyphenDict`. Если капсула содержит ненулевой указатель, то мы вызываем для него функцию `chyphenate.hnj_hyphen_free()`.

Cython-обертка разделяемой библиотеки расстановки переносов очень похожа на версию на основе пакета `ctypes`, только в отличие от нее требует создания отдельного каталога и трех небольших вспомогательных файлов. Если нас интересует только предоставление

доступа к существующим библиотекам на C или C++ из Python, то пакета `ctypes` вполне достаточно, хотя некоторые программисты считают, что с Cython (или CFFI) работать проще. Однако Cython предлагает также возможность писать код на языке Cython, то есть на Python с расширениями, который можно транслировать в быстрый код на C. Об этом мы и поговорим в следующем разделе.

5.2.2. Создание Cython-модулей для повышения производительности

Как правило, производительности написанного на Python кода вполне хватает или же его быстродействие ограничено внешними факторами (например, сетевыми задержками), которые никакими манипуляциями с кодом не устранить. Однако если задача счетная, то можно получить быстродействие откомпилированного кода на C, если воспользоваться расширениями Cython.

Но прежде чем приступать к какой-либо оптимизации, необходимо профилировать код. Большинство программ тратят большую часть времени в небольшом участке кода, поэтому любые усилия по оптимизации пропадут втуне, если не направлены на этот участок. Профилирование точно указывает узкие места и позволяет подвергнуть оптимизации именно тот код, который в ней больше всего нуждается. Кроме того, мы получаем возможность измерить эффект оптимизации, сравнив профили до и после.

В примере модуля `Image` (раздел 3.12) мы видели, что метод плавного масштабирования `scale()` работает не слишком быстро. В этом подразделе мы постараемся его оптимизировать.

```
Scaling using Image.scale()...
         18875915 function calls in 21.587 seconds
 ncalls   tottime  percall  cumtime  percall filename:lineno(function)
      1     0.000    0.000   21.587   21.587 <string>:1(<module>)
      1     1.441    1.441   21.587   21.587 __init__.py:305(scale)
 786432     7.335    0.000   19.187    0.000 __init__.py:333(_mean)
3145728     6.945    0.000    8.860    0.000 __init__.py:370(argb_for_color)
 786432     1.185    0.000    1.185    0.000 __init__.py:399(color_for_argb)
      1     0.000    0.000    0.000    0.000 __init__.py:461(<lambda>)
      1     0.000    0.000    0.002    0.002 __init__.py:479(create_array)
      1     0.000    0.000    0.000    0.000 __init__.py:75(__init__)
```

Это результат профилирования метода (из которого исключены встроенные функции, которые мы оптимизировать не можем), созданный стандартным библиотечным модулем `cProfile` (см. програм-

му benchmark_Scale.py в примерах). Свыше 21 секунды на масштабирование цветной фотографии размером 2048 × 1536 (3 145 728 пикселей) – это, конечно, не быстро, и сразу видно, где тратится время: в методе _mean() и в статических методах argb_for_color() и color_for_argb().

Мы хотим честно сравнить скорость с Cython, поэтому первым делом скопировали метод scale() и те, что он вызывает (_mean() и т. д.) в модуль Scale/Slow.py и преобразовали их в функции. Затем мы выполнили профилирование получившейся программы.

```
Scaling using Scale.scale_slow()...
        9438727 function calls in 14.397 seconds
 ncalls   tottime  percall  cumtime  percall filename:lineno(function)
      1     0.000    0.000   14.396   14.396 <string>:1(<module>)
      1     1.358    1.358   14.396   14.396 Slow.py:18(scale)
 786432     6.573    0.000   12.109    0.000 Slow.py:46(_mean)
3145728     3.071    0.000    3.071    0.000 Slow.py:69(_argb_for_color)
 786432     0.671    0.000    0.671    0.000 Slow.py:77(_color_for_argb)
```

После того как мы убрали накладные расходы на объектную ориентированность, функция scale() стала делать в два раза меньше вызовов (девять миллионов против восемнадцати), но производительность возросла всего в 1,5 раза. Тем не менее, изолировав интересующие нас функции, мы можем создать оптимизированную версию с помощью Cython и сравнить результаты.

Мы поместили Cython-код в модуль Scale/Fast.pyx и с помощью cProfile получили профиль масштабирования той же самой фотографии.

```
Scaling using Scale.scale_fast()...
        4 function calls in 0.114 seconds
 ncalls   tottime  percall  cumtime  percall filename:lineno(function)
      1     0.000    0.000    0.114    0.114 <string>:1(<module>)
      1     0.113    0.113    0.113    0.113 Scale.Fast.scale
```

Модуль cProfile не может проанализировать метод Scale.Fast.scale(), потому что он написан не на Python, а на C. Но это и не важно, потому что мы наблюдаем ускорение в 189 раз! Конечно, масштабирование одного изображения – еще не показатель, но тесты, проведенные для ряда изображений с самыми разными характеристиками, показывают, что во всех случаях коэффициент ускорения составлял не менее 130.

Такого результата удалось достичь за счет многочисленных оптимизаций, часть из которых специфична для функции scale() и вы-

зываемых из нее, а часть носит более общий характер. Перечислим наиболее существенные действия, благодаря которым Cython повысил производительность функции `scale()`.

- Копирование исходного Python-файла (например, `Slow.py`) в Cython-файл (`Fast.pyx`) дает двукратное ускорение.
- Замена всех закрытых Python-функций Cython-функциями, написанными на C, дает дополнительное ускорение в 3 раза.
- Использование функции `round()` из библиотеки `libc` вместо встроенной в Python функции `round()` дает ускорение в 4 раза[3].
- Передача представлений памяти вместо массивов дает ускорение еще в 3 раза.

Дополнительные, не столь существенные улучшения были достигнуты за счет строгой типизации всех переменных, передачи `struct` вместо Python-типа `object`, встраивания небольших функций и выполнения таких традиционных оптимизаций, как предварительное вычисление смещений.

Теперь, осознав, что может дать Cython, рассмотрим быструю версию кода, то есть модуль `Fast.pyx` и в особенности Cython-изированные варианты функции `scale()` и вспомогательных функций `_mean()`, `_argb_for_color()` и `_color_for_argb()`.

Оригинальный метод `Image.scale()` обсуждался выше (стр. 152), но показанная ниже функция – это Cython-версия функции `Scale.Slow.scale()` из модуля `Slow.py`. То же самое относится к функциям `_mean()` (стр. 153) и `_argb_for_color()` (стр. 154). Код методов и функций почти идентичен. Различия состоят только в том, что методы обращаются к пиксельным данным через `self` и вызывают другие методы, тогда как функции передают пиксельные данные явно и вызывают другие функции.

Начнем изучение `Scale/Fast.pyx` с предложений импорта и вспомогательных объявлений.

```
from libc.math cimport round
import numpy
cimport numpy
cimport cython
```

[3] Эти две функции не всегда взаимозаменяемы, поскольку ведут себя по-разному. Но при том способе использования, что встречается в функциях `scale()` и `_mean()`, их поведение одинаково.

Сначала мы импортируем из библиотеки `libc` функцию `round()`, чтобы заменить ей встроенную в Python функцию `round()`. Можно было бы, конечно, написать `cimport libc.math`, а затем писать `libc.math.round()` для C-функции и `round()` – для Python-функции, если бы нам была нужна та и другая. Далее мы импортируем модуль NumPy и входящий в состав Cython модуль `numpy.pxd`, который дает Cython доступ к NumPy на уровне языка C. В интересах Cython-функции `scale()` мы решили потребовать наличия модуля NumPy, потому что это полезно для быстрой обработки массивов. Мы также импортируем Cython-модуль `cython.pxd` ради некоторых содержащихся в нем декораторов.

```
_DTYPE = numpy.uint32
ctypedef numpy.uint32_t _DTYPE_t

cdef struct Argb:
   int alpha
   int red
   int green
   int blue

DEF MAX_COMPONENT = 0xFF
```

Здесь в первых двух строчках создаются два типа: тип Python `_DTYPE` и тип C `_DTYPE_t` – оба являются псевдонимами 32-разрядного целого без знака в NumPy. Затем мы создаем структуру C с именем `Argb`, содержащую четыре именованных целочисленных поля. (Это эквивалент именованного кортежа `Argb = collections.namedtuple("Argb", "alpha red green blue")`). Мы также определяем константу C, пользуясь предложением Cython `DEF`.

```
@cython.boundscheck(False)
def scale(_DTYPE_t[:] pixels, int width, int height, double ratio):
   assert 0 < ratio < 1
   cdef int rows = <int>round(height * ratio)
   cdef int columns = <int>round(width * ratio)
   cdef _DTYPE_t[:] newPixels = numpy.zeros(rows * columns, dtype=_DTYPE)
   cdef double yStep = height / rows
   cdef double xStep = width / columns
   cdef int index = 0
   cdef int row, column, y0, y1, x0, x1
   for row in range(rows):
      y0 = <int>round(row * yStep)
      y1 = <int>round(y0 + yStep)
      for column in range(columns):
```

```
            x0 = <int>round(column * xStep)
            x1 = <int>round(x0 + xStep)
            newPixels[index] = _mean(pixels, width, height, x0, y0, x1, y1)
            index += 1
    return columns, newPixels
```

В функции `scale()` применяется такой же алгоритм, как в `Image.scale()`, только в качестве первого аргумента она принимает одномерный массив пикселей, а затем размеры изображения и коэффициент масштабирования. Мы отказались от контроля выхода за границы массива, хотя в данном случае это не привело бы к снижению производительности. Массив пикселей передается в виде представления памяти, это эффективнее, чем передавать объекты `numpy.ndarray`, и позволяет избежать накладных расходов на уровне Python. Разумеется, с точки зрения графического программирования, возможны и другие оптимизации, например выравнивание памяти на определенную границу, но нас сейчас интересует Cython, а не детали графического программирования.

Как мы уже упоминали, синтаксис `<type>` применяется в Cython для приведения типов. Переменные создаются практически так же, как в методе `Image.scale()`, только теперь мы пользуемся типами C (`int` для целых и `double` для чисел с плавающей точкой двойной точности). Но по-прежнему можно употреблять обычный синтаксис Python, например, циклы `for … in`.

```
@cython.cdivision(True)
@cython.boundscheck(False)
cdef _DTYPE_t _mean(_DTYPE_t[:] pixels, int width, int height, int x0,
            int y0, int x1, int y1):
    cdef int alphaTotal = 0
    cdef int redTotal = 0
    cdef int greenTotal = 0
    cdef int blueTotal = 0
    cdef int count = 0
    cdef int y, x, offset
    cdef Argb argb
    for y in range(y0, y1):
        if y >= height:
            break
        offset = y * width
        for x in range(x0, x1):
            if x >= width:
                break
            argb = _argb_for_color(pixels[offset + x])
            alphaTotal += argb.alpha
            redTotal += argb.red
```

```
            greenTotal += argb.green
            blueTotal += argb.blue
            count += 1
    cdef int a = <int>round(alphaTotal / count)
    cdef int r = <int>round(redTotal / count)
    cdef int g = <int>round(greenTotal / count)
    cdef int b = <int>round(blueTotal / count)
    return _color_for_argb(a, r, g, b)
```

Компоненты цвета каждого пикселя масштабированного изображения вычисляются путем усреднения компонентов цвета тех пикселей исходного изображения, которые должен представлять данный пиксель. Пиксели исходного изображения передаются в виде представления памяти; помимо них, функция получает размеры исходного изображения и координаты противоположных углов прямоугольной области, по которой производится усреднение цветов.

Вместо того чтобы вычислять выражение $(y \times width) + x$ для каждого пикселя, мы вычисляем первое слагаемое один раз для всей строки (присваивая его переменной `offset`).

Кстати говоря, декоратор `@cython.cdivision` говорит Cython, что нужно использовать оператор / из C, а не из Python, чтобы немного ускорить работу функции.

```
cdef inline Argb _argb_for_color(_DTYPE_t color):
    return Argb((color >> 24) & MAX_COMPONENT,
                (color >> 16) & MAX_COMPONENT, (color >> 8) & MAX_COMPONENT,
                (color & MAX_COMPONENT))
```

Эта функция встраивается, то есть весь ее код вставляется в месте обращения (в функции `_mean()`), чтобы не нести накладных расходов на вызов функции и тем самым добиться максимального быстродействия.

```
cdef inline _DTYPE_t _color_for_argb(int a, int r, int g, int b):
    return (((a & MAX_COMPONENT) << 24) | ((r & MAX_COMPONENT) << 16) |
                ((g & MAX_COMPONENT) << 8) | (b & MAX_COMPONENT))
```

Эта функция также встраивается, потому что вызывается по одному разу для каждого пикселя масштабированного изображения.

Директива `inline` в Cython – это всего лишь *пожелание*, которое обычно удовлетворяется только для небольших простых функций типа тех, что приведены выше. Отметим также, что хотя в данном случае встраивание действительно повышает производительность, иногда эффект бывает прямо противоположным. Такое может случиться, если встраиваемый код занимает слишком большую часть

процессорного кэша. Как обычно, необходимо выполнять профилирование до и после применения любой оптимизации на тех машинах, где мы планируем развернуть программу. Только так можно принять обоснованное решение о том, оставлять оптимизацию или нет.

Функциональность Cython гораздо богаче, чем мы смогли рассмотреть в этом примере, и при этом хорошо документирована. Основной недостаток Cython состоит в том, что для работы с ним необходим компилятор и все прилагающиеся к нему инструменты на всех платформах, где мы собираемся собирать Cython-модули. Но если такие инструменты имеются, то Cython позволяет добиться фантастического ускорения счетных задач.

5.3. Пример: ускоренная версия пакета Image

В главе 3 мы изучали модуль `Image`, написанный на чистом Python (раздел 3.12). В этом разделе мы коротко рассмотрим Cython-модуль `cyImage`, который предоставляет большую часть функциональности `Image`, но работает гораздо быстрее.

Таблица 5.1. Сравнение скорости масштабирования изображений

Программа	Степень параллелизма	Cython	Секунд	Ускорение
`imagescale-s.py`	*Нет*	Нет	780	*Эталон*
`imagescale-cy.py`	*Нет*	Да	88	8,86×
`imagescale-m.py`	Пул из 4 процессов	Нет	206	3,79×
`imagescale.py`	Пул из 4 процессов	Да	23	33,91×

Между `Image` и `cyImage` есть два основных различия: 1) первый модуль автоматически импортирует все имеющиеся форматные модули, тогда как у второго имеется фиксированный набор форматных модулей; 2) для `cyImage` требуется модуль NumPy, тогда как `Image` будет использовать NumPy, если он установлен, но готов довольствоваться массивами типа `array` в противном случае.

В табл. 5.1 приведены результаты сравнения Cython-модуля `cyImage` с Python-модулем `Image` в программах масштабирования изображений. Но почему использование Cython дает только восьмикратное ускорение (на одно ядро), а не 130-кратное, как написанная на Cython функция `scale()`? На самом деле, будучи переписано на

Cython, масштабирование практически не потребляет времени, но ведь исходные изображения по-прежнему нужно загружать, а масштабированные – сохранять. Cython почти не ускоряет операции работы с файлами, поскольку в Python (начиная с версии 3.1) они и так уже написаны на C. Поэтому узким местом теперь является загрузка и сохранение файлов, и тут мы поделать ничего не можем.

Для создания модуля `cyImage` нужно первым делом создать каталог `cyImage` и скопировать в него все модули из каталога `Image`. Затем мы должны переименовать те модули, которые хотим Cython-изировать: в данном случае файл `__init__.py` переименовывается в `Image.pyx`, `Xbm.py` – в `Xbm.pyx`, а `Xpm.py` – в `Xpm.pyx`. Еще нужно создать новые файлы `__init__.py` и `setup.py`.

Эксперименты показывают, что после замены тела метода `Image.Image.scale()` кодом функции `Scale.Fast.scale()`, а тела метода `Image.Image._mean()` – кодом `Scale.Fast._mean()` ускорение получается совсем незначительным. Проблема, похоже, в том, что Cython гораздо лучше оптимизирует функции, а не методы. Поэтому мы скопировали модуль `Scale.Fast.pyx` в каталог `cyImage` и переименовали его в `_Scale.pyx`. Затем мы удалили метод `Image.Image._mean()` и изменили `Image.Image.scale()`, так чтобы он делегировал всю свою работу функции `_Scale.scale()`. В результате мы получили ожидаемое ускорение в 130 раз, но, конечно, общее ускорение, как уже отмечалось, значительно меньше.

```
try:
   import cyImage as Image
except ImportError:
   import Image
```

Хотя `cyImage` – не полная замена `Image` (он не поддерживает формат PNG и требует наличия модуля NumPy), для тех случаев, когда его достаточно, мы можем написать показанное выше предложение импорта, которое позволяет использовать `cyImage`, если это возможно.

```
distutils.core.setup(name="cyImage",
       include_dirs=[numpy.get_include()],
       ext_modules=Cython.Build.cythonize("*.pyx"))
```

Это тело файла `cyImage/setup.py` без предложений импорта. Здесь мы сообщаем Cython, где искать заголовочные файлы NumPy, и просим откомпилировать все `pyx`-файлы, которые он найдет в каталоге `cyImage`.

```
from cyImage.cyImage.Image import (Error, Image, argb_for_color,
        rgb_for_color, color_for_argb, color_for_rgb, color_for_name)
```

В модуле `Image` вся общая функциональность находилась в файле `Image/__init__.py`, но в случае `cyImage` мы перенесли ее в `cyImage/Image.pyx`, оставив в `cyImage/__init__.py` только показанную выше строчку. Этот файл только импортирует различные откомпилированные объекты – исключение, класс и несколько функций – делает их доступными по именам `cyImage.Image.from_file()`, `cyImage.color_for_name()` и т. д. Поскольку при импорте мы воспользовались ключевым словом `as`, то можем писать короче: `Image.Image.from_file()`, `Image.Image.Image()` и т. д.

Мы не станем подробно рассматривать `pyx`-файлы, т. к. в предыдущем подразделе уже видели, как Python-код преобразуется в Cython-код. Однако все же обратим внимание на предложения импорта в файле `cyImage/Image.pyx` и на новый метод `cyImage.Image.scale()`.

```
import sys
from libc.math cimport round
from libc.stdlib cimport abs
import numpy
cimport numpy
cimport cython
import cyImage.cyImage.Xbm as Xbm
import cyImage.cyImage.Xpm as Xpm
import cyImage.cyImage._Scale as Scale
from cyImage.Globals import *
```

Мы решили использовать C-функции `round()` и `abs()` вместо одноименных Python-функций. А вместо динамического импорта, как в модуле `Image`, мы напрямую импортируем форматные модули (то есть файлы `cyImage/Xbm.pyx` и `cyImage/Xpm.pyx`, а точнее разделяемые C-библиотеки, в которые их компилирует Cython).

```
def scale(self, double ratio):
    assert 0 < ratio < 1
    cdef int columns
    cdef _DTYPE_t[:] pixels
    columns, pixels = Scale.scale(self.pixels, self.width, self.height,
            ratio)
    return self.from_data(columns, pixels)
```

Это весь код метода `cyImage.Image.scale()`. Он оказался таким коротким, потому что содержательная работа передана функции `cyImage._Scale.scale()` (копии рассмотренной в предыдущем подразделе функции `Scale.Fast.scale()`).

Писать на Cython не так удобно, как на чистом Python, поэтому чтобы оправдать дополнительные усилия, нужно начать с профилирования Python-кода и понять, где в нем находятся узкие места. Если замедление вызвано файловым вводом-выводом или сетевыми задержками, то Cython вряд ли поможет (и стоит подумать о распараллеливании). Если же значительная часть времени тратится на вычисления, занимающие процессор, то Cython может дать заметный выигрыш, поэтому имеет смысл установить его и настроить инструменты компиляции.

Выполнив профилирование и определив, что мы хотим оптимизировать, нужно вынести медленный код в отдельный модуль и снова выполнить профилирование программы, чтобы убедиться, что мы правильно нащупали проблему. Затем нужно скопировать и переименовать (заменив расширение `.py` на `.pyx`) модуль, который мы собираемся Cython-изировать, и создать подходящий файл `setup.py` (и, возможно, вспомогательный файл `__init__.py`). И снова профилировать, на этот раз чтобы убедиться, дает ли Cython ожидаемое двукратное ускорение. Теперь можно повторять циклы Cython-изации и профилирования кода: объявить типы, использовать представления памяти, заменить тела медленных методов обращениями к Cython-изированным функциям. После каждого цикла оптимизации мы можем откатить назад бесполезные изменения, оставив только те, которые реально повышают производительность. И так поступаем до тех пор, пока не достигнем желаемого быстродействия или не исчерпаем все возможные оптимизации.

Дональд Кнут говорил: «В 97% случаев мы должны забыть о мелкой неэффективности: преждевременная оптимизация – корень всех зол» («Структурное программирование с операторами goto», ACM Journal Computing Surveys Vol. 6, No 4, December 1974, стр. 268). К тому же, никакая оптимизация не компенсирует неудачный выбор алгоритма. Но если мы выбрали хороший алгоритм и профилирование показывает наличие узких мест, то `ctypes` и Cython помогут ускорить работу счетных задач.

Доступ к функциональности, скрытой в библиотеках, написанных на языке, следующем принятым в C соглашениям о вызове, с помощью `ctypes` или Cython позволяет писать на Python высокоуровневые программы, которые для повышения быстродействия прибегают

к низкоуровневому коду. Более того, мы можем сами написать на C или C++ какой-то код и обращаться к нему, пользуясь `ctypes`, Cython или напрямую применяя интерфейс с C из Python-кода. Если мы хотим повысить производительность счетной программы, то распараллеливание даст лишь ускорение, пропорциональное количеству ядер. Напротив, программа, написанная на компилируемом языке C, может оказаться быстрее программы на чистом Python в 100 и более раз. Cython сочетает в себе лучшее из двух миров: удобство и синтаксис Python и скорость C вместе с доступом к написанным на C библиотекам.

ГЛАВА 6. Высокоуровневое сетевое программирование на Python

В стандартной библиотеке Python имеется отличная поддержка сетевого программирования на всех уровнях – от низкого до высокого. Низкоуровневая поддержка предоставляется такими модулями, как `socket`, `ssl`, `asyncore` и `asynchat`, среднеуровневая – например, модулем `socketserver`. На верхнем уровне есть много модулей, поддерживающих различные протоколы Интернета, в том числе такие хорошо известные, как `http` и `urllib`.

Существует также ряд сторонних модулей для сетевого программирования, например: Pyro4 (удаленные объекты в Python; `packages.python.org/Pyro4`), PyZMQ (интерфейс из Python к написанной на C библиотеке 0MQ; `zeromq.github.com/pyzmq`) и Twisted (`twistedmatrix.com`). Тем, кто интересуется лишь протоколами HTTP и HTTPS, порекомендуем простой в использовании пакет `requests` (`python-requests.org`).

В этой главе мы рассмотрим два модуля, поддерживающих высокоуровневое сетевое программирование: стандартный модуль `xmlrpc` (вызовы удаленных процедур с помощью XML) и сторонний модуль RPyC (удаленные вызовы Python; `rpyc.sourceforge.net`). Оба изолируют программиста от деталей низкого и среднего уровня, и, тем не менее, являются удобными и мощными.

В этой главе мы представим по одному серверу и по два клиента для `xmlrpc` и RPyC. Серверы и клиенты делают по существу одно и то же, так что будет нетрудно сравнить оба подхода. Сервер читает показания счетчиков (например, за коммунальные услуги), а клиенты запрашивают эти показания или причины, по которым их нельзя получить.

Самое существенное различие между примерами заключается в том, что сервер `xmlrpc` последовательный, а сервер RPyC – параллельный. Ниже мы увидим, что это различие в реализации оказывает заметное влияние на порядок управления серверными данными.

Чтобы не усложнять серверы, мы вынесли все управление показаниями счетчиков в отдельный модуль (последовательный называется `Meter.py`, а поддерживающий параллелизм – `MeterMT.py`). Дополнительное преимущество такого выделения состоит в том, что можно без особого труда заменить модуль работы со счетчиками модулем, работающим совсем с другими данными, и тем самым упростить адаптацию серверов и клиентов к другим целям.

6.1. Создание приложений на базе технологии XML-RPC

При использовании низкоуровневых протоколов для обмена данными по сети мы должны сформировать пакет данных, отправить его, распаковать на принимающей стороне и выполнить в ответ ту или иную операцию. Это утомительный и подверженный ошибкам процесс. Возможное решение – воспользоваться библиотекой вызова удаленных процедур (RPC). Она дает возможность просто указать имя функции и ее аргументы (строки, числа, даты и т. д.), переложив ответственность за упаковку, отправку, распаковку и выполнение операции (то есть вызов функции) на библиотеку RPC. Одним из популярных протоколов такого рода является XML-RPC, он, к тому же, стандартизирован. Библиотеки, реализующие этот протокол, кодируют данные (то есть имена и аргументы функций) в формате XML, а в качестве транспортного механизма используют HTTP.

В стандартной библиотеке Python имеются модули `xmlrpc.server` и `xmlrpc.client`, поддерживающие этот протокол. Сам протокол не зависит от языка программирования, так что даже если сервер XML-RPC написан на Python, клиент может быть написан на любом другом языке, в котором имеется поддержка протокола. Можно также писать на Python клиенты XML-RPC, которые подключаются к серверам, написанным на другом языке.

Модуль `xmlrpc` позволяет использовать некоторые специфические для Python расширения – например, передавать объекты Python – но, пойдя по этому пути, мы будем вынуждены ограничиться только серверами и клиентами, написанными на Python. В примерах из этого раздела мы не станем пользоваться этой возможностью.

Облегченной по сравнению с XML-RPC альтернативой является протокол JSON-RPC. Он предоставляет столь же богатую функциональность, но в гораздо более компактном формате (то есть обычно накладные расходы протокола – число дополнительные посылаемых по сети байтов – гораздо ниже). В библиотеке Python имеется модуль `json` для кодирования и декодирования данных Python в формате JSON, но нет ни клиентских, ни серверных модулей, поддерживающих JSON-RPC. Однако существует много сторонних Python-модулей для работы с этим протоколом (см. `en.wikipedia.org/wiki/JSON-RPC`). Еще один вариант для случая, когда клиенты и серверы написаны на Python, – воспользоваться технологией RPyC, о которой мы поговорим в следующем разделе.

6.1.1. Обертка данных

Данные, обрабатываемые клиентом и сервером, инкапсулированы в модуле `Meter.py`, который предоставляет класс `Manager`. Этот класс отвечает за хранение показаний счетчиков и содержит методы, позволяющие клиенту аутентифицироваться, получить задачи и послать результаты. Этот модуль легко заменить другим, который будет работать с иными данными.

```
class Manager:

    SessionId = 0
    UsernameForSessionId = {}
    ReadingForMeter = {}
```

В поле `SessionID` хранится уникальный идентификатор сеанса, созданного после успешной аутентификации.

В этом классе есть еще два статических словаря: в одном хранится соответствие между идентификаторами сеансов (ключами) и именами пользователей (значениями), в другом – между номерами счетчиков и их показаниями.

Обеспечивать потокобезопасность статических данных необязательно, потому что сервер `xmlrpc` не является параллельным. В версии `MeterMT.py` этого модуля параллелизм поддерживается, и мы рассмотрим ниже в подразделе 6.2.1 его отличия от `Meter.py`.

В реальной программе данные, наверное, хранились бы в DBM-файле или в базе данных, то и другое легко подставить вместо используемого здесь словаря.

```
    def login(self, username, password):
        name = name_for_credentials(username, password)
```

```
    if name is None:
       raise Error("Invalid username or password")
    Manager.SessionId += 1
    sessionId = Manager.SessionId
    Manager.UsernameForSessionId[sessionId] = username
    return sessionId, name
```

Мы хотим, чтобы клиент сначала аутентифицировался и только потом мог посылать какие-либо запросы и получить результаты.

Если имя и пароль пользователя правильны, то мы возвращаем уникальный идентификатор сеанса и настоящее имя пользователя (например, чтобы его можно было показать в пользовательском интерфейсе). Созданный идентификатор сеанса добавляется в словарь `UsernameForSessionId`. Для вызова всех остальных методов необходимо указать действительный идентификатор сеанса.

```
_User = collections.namedtuple("User", "username sha256")

def name_for_credentials(username, password):
   sha = hashlib.sha256()
   sha.update(password.encode("utf-8"))
   user = _User(username, sha.hexdigest())
   return _Users.get(user)
```

Эта функция вычисляет свертку указанного пароля по алгоритму SHA-256 и, если комбинации имени пользователя с этой сверткой соответствует запись в закрытом словаре `_Users` (не показан), то возвращает истинное имя, в противном случае `None`.

Ключом словаря `_Users` является строка `_User`, состоящая из комбинации имени пользователя (например, `carol`) и свертки SHA-256, а значением – истинное имя (например, «Carol Dent»). Таким образом, пароли в открытом виде не хранятся[1].

```
def get_job(self, sessionId):
   self._username_for_sessionid(sessionId)
   while True:                # Создаем фиктивный счетчик
      kind = random.choice("GE")
      meter = "{}{}".format(kind, random.randint(40000,
              99999 if kind == "G" else 999999))
      if meter not in Manager.ReadingForMeter:
         Manager.ReadingForMeter[meter] = None
         return meter
```

[1] Примененный здесь подход не безопасен. Чтобы сделать его безопасным, нужно было бы добавить в начало каждого пароля уникальный префикс-затравку, чтобы одинаковые пароли давали разные свертки. Еще лучше воспользоваться сторонним пакетом `passlib` (`code.google.com/p/passlib`).

После аутентификации клиент может вызывать этот метод для получения номера счетчика. Сначала метод проверяет, что передан правильный идентификатор сеанса; если это не так, то метод `_username_for_sessionid()` возбудит исключение `Meter.Error`.

У нас нет базы счетчиков, поэтому мы просто создаем фиктивный счетчик, когда клиент запрашивает задачу. Для этого генерируется номер счетчика (например, «E350718» или «G72168»), который затем вставляется в словарь `ReadingForMeter` с показанием `None`, если такого счетчика еще нет в словаре.

```
def _username_for_sessionid(self, sessionId):
   try:
      return Manager.UsernameForSessionId[sessionId]
   except KeyError:
      raise Error("Invalid session ID")
```

Этот метод либо возвращает имя пользователя из сеанса с указанным идентификатором, либо преобразует общее исключение `KeyError`, означающее отсутствие ключа в словаре, в более конкретное исключение `Meter.Error`.

Часто лучше воспользоваться специальным, а не встроенным исключением, потому что тогда мы сможем перехватывать только те исключения, которые ожидаем, и пропускать более общие, свидетельствующие о возможной ошибке в логике программы.

```
def submit_reading(self, sessionId, meter, when, reading, reason=""):
   if isinstance(when, xmlrpc.client.DateTime):
      when = datetime.datetime.strptime(when.value,
             "%Y%m%dT%H:%M:%S")
   if (not isinstance(reading, int) or reading < 0) and not reason:
      raise Error("Invalid reading")
   if meter not in Manager.ReadingForMeter:
      raise Error("Invalid meter ID")
   username = self._username_for_sessionid(sessionId)
   reading = Reading(when, reading, reason, username)
   Manager.ReadingForMeter[meter] = reading
   return True
```

Этот метод принимает идентификатор сеанса, номер счетчика (например, «G72168»), дату и время снятия показания, его значение (положительное целое число или –1, если показания нет) и причину, по которой невозможно получить показание (непустую строку в случае, когда получить показание не удалось).

Мы можем настроить сервер XML-RPC, так чтобы использовались встроенные типы Python, но по умолчанию это не так (и мы не ста-

ли это делать), поскольку протокол XML-RPC не зависит от языка. Это означает, что наш сервер сможет обслуживать клиентов, написанных на любом языке, поддерживающем XML-RPC, а не только на Python. Оборотная сторона отказа от типов Python состоит в том, что дата и время передаются в виде объектов `xmlrpc.client.DateTime`, а не `datetime.datetime`, так что их придется преобразовывать в `datetime.datetime` (возможная альтернатива – передавать в виде строк в формате ISO-8601).

Подготовив и проверив данные, мы получаем имя пользователя, соответствующее переданному идентификатору сеансу, и включаем его в состав объекта `Meter.Reading`. Это просто именованный кортеж:

```
Reading = collections.namedtuple("Reading", "when reading reason username")
```

В конце мы сохраняем показание счетчика. Мы возвращаем `True` (а не подразумеваемое по умолчанию значение `None`), потому что по умолчанию модуль `xmlrpc.server` не поддерживает `None`, а мы хотим сохранить языковую нейтральность сервера (RPyC умеет обрабатывать любое возвращаемое Python значение).

```
def get_status(self, sessionId):
   username = self._username_for_sessionid(sessionId)
   count = total = 0
   for reading in Manager.ReadingForMeter.values():
      if reading is not None:
         total += 1
         if reading.username == username:
            count += 1
   return count, total
```

После того как клиент отправил показание счетчика, у него может возникнуть желание узнать состояние: сколько показаний он отправил и сколько всего показаний сервер обработал с момента запуска. Этот метод вычисляет и возвращает оба значения.

```
def _dump(file=sys.stdout):
   for meter, reading in sorted(Manager.ReadingForMeter.items()):
      if reading is not None:
         print("{}={}@{}[{}]{}".format(meter, reading.reading,
               reading.when.isoformat()[:16], reading.reason,
               reading.username), file=file)
```

Этот метод написан исключительно для отладки, чтобы можно было проверить, все ли отправленные показания корректно сохранены.

Функциональность класса `Meter.Manager` – метод `login()` и методы получения и сохранения данных – типична для любого обертывающего данные класса, которым может пользоваться сервер. Нетрудно заменить этот класс таким, который будет работать совершенно с другими данными, почти не внося изменений в код клиентов и серверов, представленных в этой главе. Единственная проблема состоит в том, что при использовании параллельных серверов нужно обеспечивать синхронизированный доступ к разделяемым данным – с помощью блокировок или потокобезопасных классов. Как это делается, мы увидим в подразделе 6.2.1.

6.1.2. Разработка сервера XML-RPC

Благодаря модулю `xmlrpc.server` писать серверы XML-RPC очень просто. Приведенный в этом подразделе код взят из файла `meterserver-rpc.py`.

```
def main():
   host, port, notify = handle_commandline()
   manager, server = setup(host, port)
   print("Meter server startup at {} on {}:{}{}".format(
          datetime.datetime.now().isoformat()[:19], host, port, PATH))
   try:
      if notify:
         with open(notify, "wb") as file:
            file.write(b"\n")
      server.serve_forever()
   except KeyboardInterrupt:
      print("\rMeter server shutdown at {}".format(
            datetime.datetime.now().isoformat()[:19]))
      manager._dump()
```

Эта функция получает имя хоста и номер порта, создает объекты `Meter.Manager` и `xmlrpc.server.SimpleXMLRPCServer`, после чего начинает обслуживать запросы.

Если переменная `notify` содержит имя файла, то сервер создает файл и записывает в него единственный символ новой строки. Когда сервер запускается вручную, этот параметр не используется, но ниже (подраздел 6.1.3.2) мы увидим, что если сервер запускается графическим клиентом, то последний передает серверу имя файла уведомления. Затем клиент ждет завершения создания файла; когда это происходит, клиент знает, что сервер запущен и готов к работе, после чего он удаляет файл и приступает к обмену данными с сервером.

Чтобы остановить сервер, нужно нажать `Ctrl+C` или послать ему сигнал `INT` (например, командой `kill -2 pid` в Linux), который ин-

терпретатор Python преобразует в исключение `KeyboardInterrupt`. При такой остановке мы просим объект менеджера вывести распечатку данных о показаниях счетчиков для ознакомления (это единственная причина, по которой этой функции нужен доступ к менеджеру).

```
HOST = "localhost"
PORT = 11002

def handle_commandline():
   parser = argparse.ArgumentParser(conflict_handler="resolve")
   parser.add_argument("-h", "--host", default=HOST,
          help="hostname [default %(default)s]")
   parser.add_argument("-p", "--port", default=PORT, type=int,
          help="port number [default %(default)d]")
   parser.add_argument("--notify", help="specify a notification file")
   args = parser.parse_args()
   return args.host, args.port, args.notify
```

Мы приводим код этой функции только потому, что в ней для задания имени хоста используется флаг `-h` (или `--host`). По умолчанию модуль `argparse` считает, что флаг `-h` (и `--help`) означает, что нужно вывести справку и выйти. Мы хотим интерпретировать флаг `-h` самостоятельно (но оставить стандартную интерпретацию `--help`), для чего устанавливаем обработчик конфликтов в анализаторе аргументов.

К сожалению, при переносе `argparse` в версию Python 3 было сохранено старое (в стиле Python 2) форматирование с помощью знаков %, хотя надо было бы заменить его скобками `str.format()`. Поэтому, чтобы включить в текст справки значения по умолчанию, придется писать `%(default)`*t*, где *t* – тип значения (`d` – для десятичного целого, `f` – для числа с плавающей точкой и `s` – для строки).

```
def setup(host, port):
   manager = Meter.Manager()
   server = xmlrpc.server.SimpleXMLRPCServer((host, port),
              requestHandler=RequestHandler, logRequests=False)
   server.register_introspection_functions()
   for method in (manager.login, manager.get_job, manager.submit_reading,
          manager.get_status):
      server.register_function(method)
   return manager, server
```

Эта функция используется для создания менеджера данных (со счетчиков) и сервера. Метод `register_introspection_functions()` предоставляет в распоряжение клиентов три функции интроспек-

ции: `system.listMethods()`, `system.methodHelp()` и `system.methodSignature()` (они не используются рассматриваемыми клиентами XML-RPC, но могут пригодиться для отладки более сложных клиентов). Все методы менеджера, которые должны быть доступны клиентам, необходимо зарегистрировать на сервере, для этого служит метод `register_function()` (см. врезку «Связанные и несвязанные методы» на стр. 77).

```
PATH = "/meter"

class RequestHandler(xmlrpc.server.SimpleXMLRPCRequestHandler):
   rpc_paths = (PATH,)
```

Серверу счетчиков не нужно обрабатывать запросы каким-то специальным образом, поэтому мы создали простейший обработчик: унаследовали классу и указали уникальный путь, чтобы можно было опознать запросы к серверу счетчиков.

Итак, сервер готов, и можно приступать к написанию клиентов для него.

6.1.3. Разработка клиента XML-RPC

В этом подразделе мы рассмотрим два разных клиента: один консольный, предполагающий, что сервер уже запущен, второй – графический, который либо обращается к работающему серверу, либо – если такого еще нет – запускает сервер сам.

6.1.3.1. Консольный клиент XML-RPC

Прежде чем приступать к анализу кода, взглянем на типичный интерактивный сеанс с сервером. Сервер `meterserver-rpc.py` должен быть уже запущен.

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn't find the meter
```

```
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447:
Reason for meter G40447:
$
```

Пользователь Carol запускает клиента. Ее просят ввести имя пользователя или нажать `Enter`, чтобы согласиться с умолчанием (показано в квадратных скобках). Она нажимает `Enter`. Затем ее просят ввести пароль, что она и делает без эхо-контроля. Сервер опознает ее и приветствует, выводя полное имя. Затем клиент запрашивает у сервера номер счетчика и предлагает Carol ввести новое показание. Если она вводит число, то оно передается серверу и, как правило, принимается. Если же допущена ошибка (как в случае второго показания) или если введенное показание не годится еще по каким-то причинам, клиент печатает сообщение и предлагает ввести показание еще раз. Если сервер принял показание (или причину), то пользователю сообщается, сколько показаний он ввел в течение данного сеанса и сколько всего показаний было введено с момента запуска сервера (в том числе другими людьми, обращающимися к серверу). Если Carol нажимает `Enter`, не введя показание, клиент предлагает ей ввести причину, по которой сообщить показание невозможно. Если же она не вводит ни показание, ни причину, то клиент завершает работу.

```
def main():
   host, port = handle_commandline()
   username, password = login()
   if username is not None:
      try:
         manager = xmlrpc.client.ServerProxy("http://{}:{}{}".format(
                 host, port, PATH))
         sessionId, name = manager.login(username, password)
         print("Welcome, {}, to Meter RPC".format(name))
         interact(manager, sessionId)
      except xmlrpc.client.Fault as err:
         print(err)
      except ConnectionError as err:
         print("Error: Is the meter server running? {}".format(err))
```

Сначала эта функция получает имя хоста и номер порта сервера (или значения по умолчанию), а затем имя и пароль пользователя. Затем она создает объект-заместитель (`manager`) экземпляра `Meter.Manager`, работающего на сервере (паттерн Заместитель мы обсуждали в разделе 2.7).

Создав заместитель менеджера, мы используем его, чтобы аутентифицироваться и начать взаимодействие с сервером. Если сервер не запущен, то мы получим исключение `ConnectionError` (или `socket.error` в версиях, предшествующих Python 3.3).

```
def login():
   loginName = getpass.getuser()
   username = input("Username [{}]: ".format(loginName))
   if not username:
      username = loginName
   password = getpass.getpass()
   if not password:
      return None, None
   return username, password
```

Функция `getuser()` из модуля `getpass` возвращает имя текущего пользователя, которое мы предлагаем по умолчанию. Функция `getpass()` запрашивает пароль, отключая эхо-контроль. Функции `input()` и `getpass.getpass()` возвращают строки без завершающих символов новой строки.

```
def interact(manager, sessionId):
   accepted = True
   while True:
      if accepted:
         meter = manager.get_job(sessionId)
         if not meter:
            print("All jobs done")
            break
      accepted, reading, reason = get_reading(meter)
      if not accepted:
         continue
      if (not reading or reading == -1) and not reason:
         break
      accepted = submit(manager, sessionId, meter, reading, reason)
```

Эта функция вызывается после успешной аутентификации для организации взаимодействия между клиентом и сервером. Она в цикле запрашивает у сервера задачу (то есть счетчик, для которого нужно ввести показания), получает от пользователя показание или причину и отправляет данные серверу. Выход из цикла происходит, когда пользователь откажется ввести и показание, и причину.

```
def get_reading(meter):
   reading = input("Reading for meter {}: ".format(meter))
   if reading:
      try:
```

```
            return True, int(reading), ""
         except ValueError:
            print("Invalid reading")
            return False, 0, ""
      else:
         return True, -1, input("Reason for meter {}:
".format(meter))
```

Эта функция должна обрабатывать три случая: пользователь ввел допустимое показание (целое число), пользователь ввел недопустимое показание, пользователь вообще не ввел показание. Если показание не введено, то далее пользователь должен либо ввести причину, либо ничего не вводить (последнее означает, что сеанс окончен).

```
def submit(manager, sessionId, meter, reading, reason):
   try:
      now = datetime.datetime.now()
      manager.submit_reading(sessionId, meter, now, reading, reason)
      count, total = manager.get_status(sessionId)
      print("Accepted: you have read {} out of {} readings".format(
            count, total))
      return True
   except (xmlrpc.client.Fault, ConnectionError) as err:
      print(err)
      return False
```

После получения показания или причины вызывается эта функция, которая отправляет данные серверу с помощью заместителя менеджера. Затем функция запрашивает у сервера состояние (сколько показаний отправил данный пользователь и сколько всего показаний было отправлено с момента запуска сервера).

Код клиента длиннее, чем код сервера, но при этом абсолютно прямолинеен. А поскольку мы работаем с протоколом XML-RPC, клиент можно написать на любом языке, поддерживающем этот протокол. Клиенты также могут использовать другие технологии организации интерфейса, например, Urwid (`excess.org/urwid`) –консольный интерфейс для Unix или какую-нибудь библиотеку ГИП типа Tkinter.

6.1.3.2. Клиент XML-RPC с графическим интерфейсом

Программирование ГИП с помощью Tkinter рассматривается в главе 7, поэтому читателям, не знакомым с Tkinter, рекомендуется сначала прочитать эту главу, а затем вернуться сюда. В этом подразделе мы уделим внимание лишь тем аспектам программы `meter-rpc`.

pyw, которые касаются взаимодействия с сервером счетчиков. Интерфейс программы показан на рис. 6.1.

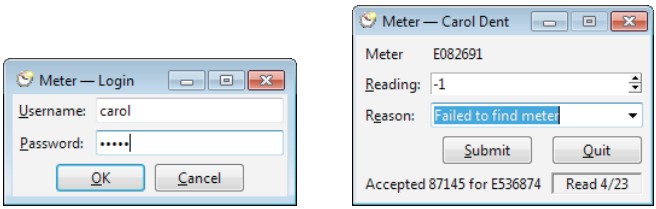

Рис. 6.1. Окно аутентификации и главное окно графического клиента XML-RPC в Windows

```
class Window(ttk.Frame):

   def __init__(self, master):
      super().__init__(master, padding=PAD)
      self.serverPid = None
      self.create_variables()
      self.create_ui()
      self.statusText.set("Ready...")
      self.countsText.set("Read 0/0")
      self.master.after(100, self.login)
```

Создав главное окно, мы присваиваем идентификатору серверного процесса значение `None` и вызываем метод `login()` через 100 миллисекунд после конструирования главного окна. Это оставляет Tkinter время нарисовать главное окно, но еще до того как пользователь получит шанс начать с ним взаимодействие, создается *локально модальное* окно аутентификации. Если в приложении открыто локально модальное окно, то только с ним и может взаимодействовать пользователь данного приложения. Это означает, что хотя пользователь и видит главное окно, он ничего не может в нем сделать, пока не аутентифицируется и модальное окно не исчезнет с экрана.

```
class Result:

   def __init__(self):
      self.username = None
      self.password = None
      self.ok = False
```

Этот крохотный класс (из файла `MeterLogin.py`) служит для хранения результатов взаимодействия пользователя с диалоговым окном аутентификации. Передав диалоговому окну ссылку на экземп-

ляр `Result`, мы сможем узнать, что ввел пользователь уже после того, как окно будет закрыто и удалено.

```
def login(self):
   result = MeterLogin.Result()
   dialog = MeterLogin.Window(self, result)
   if result.ok and self.connect(result.username, result.password):
      self.get_job()
   else:
      self.close()
```

Этот метод создает результирующий объект, а затем модальное диалоговое окно аутентификации. При обращении к `MeterLogin.Window()` окно аутентификации появляется на экране и остается там, до тех пор пока его не закроют, блокируя доступ к другим окнам. Пока модальное окно видно, пользователь не может взаимодействовать ни с какими другими окнами, то есть должен либо ввести имя и пароль, а затем нажать `OK`, либо снять программу, нажав `Cancel`.

После нажатия любой из двух кнопок окно закрывается (и удаляется). Если была нажата кнопка `OK` (это возможно, только если пользователь предварительно ввел непустые имя и пароль), то делается попытка подключиться к серверу и получить первую задачу. Если пользователь отменил аутентификацию или подключение не удалось выполнить, то главное окно закрывается (и удаляется), а приложение завершается.

```
def connect(self, username, password):
   try:
      self.manager = xmlrpc.client.ServerProxy("http://{}:{}{}"
             .format(HOST, PORT, PATH))
      name = self.login_to_server(username, password)
      self.master.title("Meter \u2014 {}".format(name))
      return True
   except (ConnectionError, xmlrpc.client.Fault) as err:
      self.handle_error(err)
      return False
```

Этот метод вызывается сразу после ввода имени и пароля. Сначала он создает заместитель серверного объекта `Meter.Manager`, а затем пытается аутентифицироваться. Затем в заголовке окна отображается имя приложения, длинное тире (—, кодовая позиция в Unicode `U+2014`) и имя пользователя, и метод возвращает `True`.

В случае ошибки открывается окно с текстом сообщения об ошибке и возвращается `False`.

```
def login_to_server(self, username, password):
    try:
        self.sessionId, name = self.manager.login(username, password)
    except ConnectionError:
        self.start_server()
        self.sessionId, name = self.manager.login(username, password)
    return name
```

Если сервер счетчиков уже запущен, то первая попытка подключения окажется успешной, и будет получен идентификатор сеанса и имя пользователя. Если же при подключении возникнет исключение `ConnectionError`, то приложение считает, что сервер не запущен и пытается запустить его, а затем подключиться снова. Если и вторая попытка завершается неудачей, то исключение `ConnectionError` распространяется в вызывающий метод `self.login()`, который его перехватывает, показывает окно с сообщением об ошибке и завершает приложение.

```
SERVER = os.path.join(os.path.dirname(os.path.realpath(__file__)),
        "meterserver-rpc.py")
```

Эта константа содержит полный путь к имени сервера. Предполагается, что сервер находится в том же каталоге, что и графический клиент. Разумеется, чаще бывает, что сервер находится на одной машине, а клиент на другой. Но некоторые приложения сознательно разбиваются на две части – серверную и клиентскую, – которые должны находиться на одной и той же машине.

Такая двухчастная архитектура полезна, когда мы хотим полностью отделить функциональность приложения от его пользовательского интерфейса. Минусом является тот факт, что приходится поставлять два исполняемых файла вместо одного, да и сетевые издержки присутствуют, но в случае, когда клиент и сервер работают на одной машине, пользователь их, скорее всего, не заметит. А плюс в том, что клиент и сервер можно разрабатывать независимо, и перенос на другую платформу существенно упрощается, поскольку код сервера может вообще не зависеть от платформы, так что переносить придется только клиента. Кроме того, это развязывает руки для освоения новых технологий построения пользовательских интерфейсов (например, новой библиотеки ГИП), т. к. все сводится к переписыванию клиента. Еще одно потенциальное преимущество – уточненная безопасность; например, сервер может работать с точно определенными и весьма ограниченными правами, тогда как клиент – с правами пользователя.

```
def start_server(self):
   filename = os.path.join(tempfile.gettempdir(),
        "M{}.$$$".format(random.randint(1000, 9999)))
   self.serverPid = subprocess.Popen([sys.executable, SERVER,
        "--host", HOST, "--port", str(PORT), "--notify",
        filename]).pid
   print("Starting the server...")
   self.wait_for_server(filename)
```

Для запуска сервера применяется функция `subprocess.Popen()`. При таком использовании подпроцесс (то есть сервер) запускается без блокировки.

Если бы мы запускали обычную программу (как подпроцесс) и хотели бы дождаться ее завершения, то можно было бы явно подождать. Но здесь мы запускаем сервер, который должен работать, пока работает клиент, так что ждать не имеет смысла. Кроме того, мы должны дать серверу шанс запуститься и не пытаться повторно аутентифицироваться, пока он не будет готов принимать запросы. Решение простое: мы создаем псевдослучайное имя файла, которое передаем запускаемому серверу в аргументе `notify`. Затем мы ждем, когда сервер создаст этот файл, давая клиенту знать о своей готовности.

```
def wait_for_server(self, filename):
   tries = 100
   while tries:
      if os.path.exists(filename):
         os.remove(filename)
         break
      time.sleep(0.1) # Даем серверу возможность запуститься
      tries -= 1
  else:
      self.handle_error("Failed to start the RPC Meter Server")
```

Этот метод блокирует (замораживает) пользовательский интерфейс на время до 10 секунд (100 попыток × 0,1 с), хотя на практике ждать приходится всего-то долю секунды. Как только сервер создаст файл уведомления, клиент сразу же удалит его и продолжит обработку событий – в данном случае попробует еще раз аутентифицировать пользователя с указанными им учетными данными и, если получится, даст доступ к окну для ввода показания счетчика. Если сервер так и не запустится, то цикл `while` завершится естественным образом (минуя `break`), и будет выполнена ветвь `else`.

Опрос – не идеальное решение, особенно в приложениях с пользовательским интерфейсом, но, поскольку нам нужна кросс-платфор-

менная программа, а работать без сервера приложение не может, то это самый простой подход.

```
def get_job(self):
   try:
      meter = self.manager.get_job(self.sessionId)
      if not meter:
         messagebox.showinfo("Meter \u2014 Finished",
              "All jobs done", parent=self)
         self.close()
      self.meter.set(meter)
      self.readingSpinbox.focus()
   except (xmlrpc.client.Fault, ConnectionError) as err:
      self.handle_error(err)
```

После того как аутентификация успешно завершилась (с запуском сервера при необходимости), вызывается этот метод, который получает первую задачу. Переменная `self.meter` имеет тип `tkinter.StringVar` и ассоциирована с меткой, в которой отображается номер счетчика.

```
def submit(self, event=None):
   if self.submitButton.instate((tk.DISABLED,)):
      return
   meter = self.meter.get()
   reading = self.reading.get()
   reading = int(reading) if reading else -1
   reason = self.reason.get()
   if reading > -1 or (reading == -1 and reason and reason != "Read"):
      try:
         self.manager.submit_reading(self.sessionId, meter,
               datetime.datetime.now(), reading, reason)
         self.after_submit(meter, reading, reason)
      except (xmlrpc.client.Fault, ConnectionError) as err:
         self.handle_error(err)
```

Этот метод вызывается, когда пользователь нажимает кнопку `Submit`, а приложение разрешает это сделать лишь в том случае, когда введено ненулевое показание или непустая причина. Программа получает от интерфейса номер счетчика, показание (в виде `int`) и причину и отправляет все это серверу с помощью заместителя менеджера. Если отправленное показание принято, то вызывается метод `after_submit()`; в противном случае информация об ошибке передается методу `handle_error()`.

```
def after_submit(self, meter, reading, reason):
   count, total = self.manager.get_status(self.sessionId)
```

```
        self.statusText.set("Accepted {} for {}".format(
                reading if reading != -1 else reason, meter))
        self.countsText.set("Read {}/{}".format(count, total))
        self.reading.set(-1)
        self.reason.set("")
        self.get_job()
```

Этот метод запрашивает у заместителя менеджера текущее состояние и обновляет метки состояния и счетчиков. Он также сбрасывает поля для ввода показания и причины и просит у сервера следующую задачу.

```
    def handle_error(self, err):
        if isinstance(err, xmlrpc.client.Fault):
            err = err.faultString
            messagebox.showinfo("Meter \u2014 Error",
                 "{}\nIs the server still running?\n"
                 "Try Quitting and restarting.".format(err), parent=self)
```

Этот метод вызывается в случае ошибки. Он выводит сообщение об ошибке в модальном окне с единственной кнопкой `OK`.

```
    def close(self, event=None):
        if self.serverPid is not None:
            print("Stopping the server...")
            os.kill(self.serverPid, signal.SIGINT)
            self.serverPid = None
        self.quit()
```

Когда пользователь закрывает приложение, мы проверяем, работал уже сервер в момент запуска приложения или приложение запускало его самостоятельно. В последнем случае приложение корректно завершает сервер, посылая ему сигнал прерывания (который интерпретатор Python преобразует в исключение `KeyboardInterrupt`).

Функция `os.kill()` посылает сигнал (каждому сигналу соответствует константа, определенная в модуле `signal`) процессу с указанным идентификатором. Эта функция работала только в Unix в версии Python 3.1, но, начиная с Python 3.2, она работает как в Unix, так и в Windows.

Консольный клиент `meterclient-rpc.py` содержит примерно 100 строк кода. Графический клиент `meter-rpc.pyw` содержит уже 250 строк (и еще 100 строк в коде окна диалогового окна `MeterLogin.py`). Оба просты и легко переносятся на другую платформу, а благодаря поддержке тем в Tkinter интерфейс клиента выглядит так, как принято на конкретной платформе, будь то OS X или Windows.

6.2. Создание приложений на базе технологии RPyC

Если клиенты и серверы написаны на Python, то вместо громоздкого протокола типа XML-RPC можно использовать протокол, «заточенный» под Python. Существует много пакетов, предлагающих вызов удаленных Python-процедур из Python-программ, в этом разделе мы воспользуемся пакетом RPyC (`rpyc.sourceforge.net`). У этого модуля есть два режима работы: «классический» и более современный «сервис-ориентированный». Нас будет интересовать последний.

По умолчанию RPyC-серверы параллельны, поэтому использовать не поддерживающую параллелизм обертку данных `Meter.py` из предыдущего раздела не удастся. Вместо нее мы напишем модуль `MeterMT.py`. Для этого нам понадобятся два новых класса, `ThreadSafeDict` и `_MeterDict`, а также модифицированный класс `Manager`, в котором будут использоваться эти словари вместо стандартных.

6.2.1. Потокобезопасная обертка данных

Модуль `MeterMT` содержит поддерживающий параллелизм класс `Manager` и два потокобезопасных словаря. Начнем с рассмотрения статических данных класса `Manager` и тех его методов, которые отличаются от того, что мы видели в классе `Meter.Manager` из предыдущего раздела.

```
class Manager:

   SessionId = 0
   SessionIdLock = threading.Lock()
   UsernameForSessionId = ThreadSafeDict()
   ReadingForMeter = _MeterDict()
```

Для поддержки параллелизма в классе `MeterMT.Manager` необходимо использовать блокировки, чтобы сериализовать доступ к статическим данным. Синхронизировать доступ к идентификатору сеансов можно непосредственно, но для двух словарей необходимы потокобезопасные версии, которые мы скоро рассмотрим.

```
   def login(self, username, password):
      name = name_for_credentials(username, password)
      if name is None:
         raise Error("Invalid username or password")
```

```
        with Manager.SessionIdLock:
            Manager.SessionId += 1
            sessionId = Manager.SessionId
        Manager.UsernameForSessionId[sessionId] = username
        return sessionId, name
```

Этот метод отличается от оригинального только тем, что инкремент и присваивание значения идентификатору сеанса производятся в контексте блокировки. Не будь блокировки, могла бы сложиться ситуация, когда поток `A` увеличивает идентификатор сеанса, затем то же самое делает поток `B`, после чего оба потока `A` и `B` читают одно и то же дважды увеличенное значение, вместо того чтобы получить уникальный идентификатор.

```
    def get_status(self, sessionId):
        username = self._username_for_sessionid(sessionId)
        return Manager.ReadingForMeter.status(username)
```

Теперь этот метод почти всю свою работу перепоручает методу `_MeterDict.status()`, который будет рассмотрен ниже.

```
    def get_job(self, sessionId):
        self._username_for_sessionid(sessionId)
        while True: # Create fake meter
            kind = random.choice("GE")
            meter = "{}{}".format(kind, random.randint(40000,
                    99999 if kind == "G" else 999999))
            if Manager.ReadingForMeter.insert_if_missing(meter):
                return meter
```

От предыдущей версии этот метод отличается последними двумя строчками. Нам нужно проверить, находится ли фиктивный счетчик в словаре, и, если нет, то вставить его в словарь с начальным показанием `None`. Тем самым гарантируется, что тот же номер нельзя будет использовать повторно. Раньше проверка и вставка производились в двух разных предложениях, но в параллельной программе так делать нельзя, потому что между ними могут вклиниться другие потоки. Поэтому мы делегируем работу методу `_MeterDict.insert_if_missing()`, который возвращает признак, показывающий, была произведена вставка или нет.

```
    def submit_reading(self, sessionId, meter, when, reading,
            reason=""):
        if (not isinstance(reading, int) or reading < 0) and not reason:
            raise Error("Invalid reading")
        if meter not in Manager.ReadingForMeter:
```

```
            raise Error("Invalid meter ID")
        username = self._username_for_sessionid(sessionId)
        reading = Reading(when, reading, reason, username)
        Manager.ReadingForMeter[meter] = reading
```

Это очень похоже на версию, где использовался протокол XML-RPC, только теперь нам не нужно преобразовывать временную метку `when` и не нужно возвращать `True`, потому что для RPyC возвращаемое неявно значение `None` вполне приемлемо.

6.2.1.1. Простой потокобезопасный словарь

Если мы работаем с CPython (стандартный интерпретатор Python, написанный на C), то теоретически наличие GIL (глобальной блокировки интерпретатора) делает словари потокобезопасными, потому что интерпретатор в каждый момент времени может исполнять только один поток (вне зависимости от количества ядер), так что по отдельности вызовы методов исполняются как атомарные действия. Однако это не помогает, если требуется атомарно выполнить два или более методов словаря. Да и в любом случае мы не должны опираться на эту деталь реализации, ведь существуют и другие реализации Python (например, Jython и IronPython), в которых нет GIL, так что методы словарей в них нельзя считать атомарными.

Если нам нужен по-настоящему потокобезопасный словарь, то необходимо использовать стороннюю реализацию или написать свою собственную. Сделать это нетрудно, потому что можно взять существующий стандартный словарь и предоставить к нему доступ с помощью своих потокобезопасных методов. В этом подразделе мы рассмотрим потокобезопасный словарь `ThreadSafeDict`, который предоставляет подмножество функциональности `dict`, достаточное для сервера счетчиков.

```
class ThreadSafeDict:

   def __init__(self, *args, **kwargs):
      self._dict = dict(*args, **kwargs)
      self._lock = threading.Lock()
```

Класс `ThreadSafeDict` агрегирует `dict` и `threading.Lock`. Мы не хотим наследовать `dict`, потому что наша задача – опосредовать любой доступ к `self._dict`, так чтобы все операции были сериализованы (в каждый момент времени обращаться к `self._dict` может не более одного потока).

```
def copy(self):
   with self._lock:
      return self.__class__(**self._dict)
```

Блокировки Python поддерживают протокол контекстного менеджера, поэтому если просто воспользоваться предложением `with`, то блокировка будет гарантированно освобождена, когда в ней отпадет надобность – даже в случае исключения.

Предложение `with self._lock` разрешит потоку доступ внутрь блока только тогда, когда никакой другой поток не удерживает блокировку, а в противном случае приостановит его выполнение. Таким образом, в каждый момент времени захватить блокировку может не более одного потока. Поэтому так важно, чтобы код, исполняемый в контексте блокировки, отрабатывал как можно быстрее. В данном случае выполняется довольно дорогая операция, но лучшего решения не видно.

Если в классе реализован метод `copy()`, то ожидается, что он вернет копию экземпляра, от имени которого вызван. Мы не можем вернуть `self._dict.copy()`, потому что это дало бы экземпляр стандартного словаря `dict`. Возврат `ThreadSafeDict(**self._dict)` почти годится – беда лишь в том, что так мы всегда получаем экземпляр `ThreadSafeDict`, даже при вызове от имени экземпляра подкласса (если только в этом подклассе нет собственной реализации метода `copy()`). Приведенный выше код работает как для `ThreadSafeDict`, так и для его подклассов (см. врезку «Распаковка последовательностей и отображений» на стр. 24).

```
def get(self, key, default=None):
   with self._lock:
      return self._dict.get(key, default)
```

Это потокобезопасный вариант метода `dict.get()`.

```
def __getitem__(self, key):
   with self._lock:
      return self._dict[key]
```

Этот специальный метод поддерживает доступ к словарю по ключу, то есть синтаксис *`value`* `= d[`*`key`*`]`.

```
def __setitem__(self, key, value):
   with self._lock:
      self._dict[key] = value
```

Этот специальный метод поддерживает вставку в словарь и изменение существующего значения, то есть синтаксис `d[key] = value`.

```
def __delitem__(self, key):
   with self._lock:
      del self._dict[key]
```

Этот специальный метод поддерживает предложение `del`: `del d[key]`.

```
def __contains__(self, key):
   with self._lock:
      return key in self._dict
```

Этот специальный метод возвращает `True`, если в словаре имеется элемент с указанным ключом, в противном случае возвращает `False`. Он вызывается при употреблении ключевого слова `in`, например, `if k in d: ….`

```
def __len__(self):
   with self._lock:
      return len(self._dict)
```

Этот специальный метод возвращает количество элементов в словаре. Он поддерживает встроенную функцию `len()`, например, `count = len(d)`.

В классе `ThreadSafeDict` нет методов `clear()`, `fromkeys()`, `items()`, `keys()`, `pop()`, `popitem()`, `setdefault()`, `update()` и `values()`, имеющихся в стандартном словаре. Большинство из них реализовать совсем не трудно. Однако к методам, возвращающим представления (например, `items()`, `keys()` и `values()`), нужно подходить осторожно. Самое простое и безопасное решение – не реализовывать их вовсе. Альтернатива – сделать так, чтобы они возвращали копии данных в виде списка (например, тело `keys()` могло бы выглядеть так: `with self._lock: return list(self._dict.keys())`). Но для больших словарей это потребовало бы очень много памяти и, к тому же, пока этот метод выполняется, все остальные потоки не смогут обратиться к словарю.

Другой подход к организации потокобезопасного словаря заключается в том, чтобы создать обычный словарь и позаботиться о том, чтобы запись в него производилась только в том потоке, в котором он был создан (или перед каждой записью захватывать блокировку). Тогда другим потокам можно было бы предоставить допускающие только чтение (а значит, потокобезопасные) представления этого

словаря с помощью класса `types.MappingProxyType`, появившегося в версии Python 3.3.

6.2.1.2. Подкласс для словаря показаний счетчиков

Вместо того чтобы использовать для словаря показаний счетчиков (ключи – номера счетчиков, значения – их показания) сам класс `ThreadSafeDict`, мы создали его закрытый подкласс `_MeterDict`, в который добавили два метода.

```
class _MeterDict(ThreadSafeDict):

   def insert_if_missing(self, key, value=None):
      with self._lock:
         if key not in self._dict:
            self._dict[key] = value
            return True
      return False
```

Этот метод вставляет в словарь указанную пару (ключ, значение) и возвращает `True` или – если ключ (например, номер фиктивного счетчика) уже есть в словаре – ничего не делает и возвращает `False`. Тем самым гарантируется, что в ответ на каждый запрос задачи возвращается новый уникальный счетчик.

По существу, метод `insert_if_missing()` выполняет такой код:

```
if meter not in ReadingForMeter: # ЭТО НЕПРАВИЛЬНО!
   ReadingForMeter[key] = None
```

`ReadingForMeter` – экземпляр класса `_MeterDict`, поэтому он наследует всю функциональность класса `ThreadSafeDict`. Но несмотря на то, что каждый из методов `ReadingForMeter.__contains__()` (нужен для `in`) и `ReadingForMeter.__setitem__ ()` (нужен для `[]`) по отдельности потокобезопасен, показанный выше код потокобезопасным *не* является. Дело в том, что второй поток может обратиться к словарю `ReadingForMeter`, после того как первый выполнил предложение `if`, но до того как он произвел присваивание. Выйти из положения можно, выполнив обе операции в контексте одной и той же блокировки, что метод `insert_if_missing()` и делает.

```
   def status(self, username):
      count = total = 0
```

```
        with self._lock:
            for reading in self._dict.values():
                if reading is not None:
                    total += 1
                    if reading.username == username:
                        count += 1
        return count, total
```

Этот метод может работать долго, потому что перебирает все хранящиеся в словаре значения в контексте блокировки. Альтернатива – оставить в контексте только одно предложение – `values = self._dict.values()` – а перебор сделать позже (уже после освобождения блокировки). Что быстрее: скопировать элементы под защитой блокировки, а затем обработать их без блокировки, или производить всю обработку под защитой блокировки, зависит от обстоятельств. Узнать точно можно только одним способом – выполнить профилирование обоих вариантов в реалистичных условиях.

6.2.2. Разработка сервера RPyC

Выше мы видели, как легко написать сервер XML-RPC с помощью модуля `xmlrpc.server` (подраздел 6.1.2). Ничуть не труднее – хотя и по-другому – создать сервер RPyC.

```
import datetime
import threading
import rpyc
import sys
import MeterMT

PORT = 11003

Manager = MeterMT.Manager()
```

Это начало файла `meterserver-rpyc.py`. Мы импортируем два стандартных модуля, затем модуль `rpyc` и наш потокобезопасный модуль `MeterMT`. Мы зафиксировали номер порта, хотя нетрудно было бы задавать его в командной строке и выделять с помощью модуля `argparse`, как в версии на основе XML-RPC. И еще мы создали единственный экземпляр класса `MeterMT.Manager`, общий для всех потоков сервера RPyC.

```
if __name__ == "__main__":
    import rpyc.utils.server
    print("Meter server startup at {}".format(
            datetime.datetime.now().isoformat()[:19]))
```

```
server = rpyc.utils.server.ThreadedServer(MeterService, port=PORT)
thread = threading.Thread(target=server.start)
thread.start()
try:
   if len(sys.argv) > 1: # Уведомить, если запущен графическим клиентом
      with open(sys.argv[1], "wb") as file:
         file.write(b"\n")
   thread.join()
except KeyboardInterrupt:
   pass
server.close()
print("\rMeter server shutdown at {}".format(
        datetime.datetime.now().isoformat()[:19]))
MeterMT.Manager._dump()
```

На этом серверная программа заканчивается. Мы импортируем модуль сервера RPyC и сообщаем о запуске. Затем создаем экземпляр многопоточного сервера и передаем ему класс `MeterService`. Сервер будет создавать экземпляры этого класса, который мы рассмотрим чуть ниже, по мере необходимости.

После создания сервера можно было бы просто написать `server.start()` и на этом закончить. Тогда сервер запустился бы и работал «вечно». Однако мы хотим, чтобы пользователь мог остановить сервер нажатием `Ctrl+C` (или отправкой сигнала `INT`) и чтобы при остановке сервер распечатывал показания счетчиков.

Для этого мы запускаем сервер в своем собственном потоке, из которого он сможет создать пул потоков для обработки входящих соединений, а затем ждем завершения серверного потока (с помощью `thread.join()`). Если сервер прерван, мы перехватываем и игнорируем исключение, а затем закрываем сервер. Вызов `close()` блокирует программу до тех пор, пока все потоки не обработают текущий запрос. После этого мы печатаем сообщение о том, что сервер остановлен, и выводим полученные сервером показания счетчиков.

Если сервер был запущен графическим клиентом, то мы ожидаем, что клиент передаст имя файла уведомления в качестве единственного аргумента. Если в командной строке присутствует аргумент, то мы создаем файл и записываем в него символ новой строки, давая клиенту знать, что сервер запущен.

В режиме службы сервер RPyC принимает подкласс `rpyc.Service`, который затем может использовать как фабрику для порождения экземпляров службы (фабрики рассматривались в главе 1, разделы 1.1 и 1.3). Мы создали класс `MeterService`, являющийся тонкой оберткой вокруг рассмотренного выше класса `MeterMT.Manager`.

```
class MeterService(rpyc.Service):

   def on_connect(self):
      pass

   def on_disconnect(self):
      pass
```

При подключении к службе вызывается метод `on_connect()`, а при разрыве соединения – метод `on_disconnect()`. Мы ни в том, ни в другом случае ничего делать не собираемся, поэтому оба метода пусты. Разрешается вообще не реализовывать эти методы, если они не нужны; мы включили их, только чтобы продемонстрировать сигнатуры.

```
exposed_login = Manager.login
exposed_get_status = Manager.get_status
exposed_get_job = Manager.get_job
```

Служба может раскрывать методы (или классы, или другие объекты) клиентам. Любой класс или метод, имя которого начинается префиксом `exposed_`, доступен клиентам, причем в случае метода вызывать его можно по имени с префиксом или без префикса. Например, клиент сервера RPyC мог бы вызвать метод `exposed_login()` или `login()`.

Что касается методов `exposed_login()`, `exposed_get_status()` и `exposed_get_job()`, то это просто ссылки на соответствующие методы экземпляра менеджера счетчиков.

```
def exposed_submit_reading(self, sessionId, meter, when, reading,
          reason=""):
   when = datetime.datetime.strptime(str(when)[:19],
            "%Y-%m-%d %H:%M:%S")
   Manager.submit_reading(sessionId, meter, when, reading, reason)
```

Этот метод является тонкой оберткой вокруг метода менеджера счетчиков. Причина в том, что переменная `when` передается в виде не чистого объекта `datetime.datetime`, а обернутого сетевой ссылкой RPyC. В большинстве случаев это неважно, но здесь мы хотим хранить в словаре счетчиков истинные объекты `datetime.datetime`, а не ссылки на удаленные (то есть клиентские) объекты. Поэтому мы преобразуем обернутую временную метку в строку в формате ISO 8601, получаем из нее объект `datetime.datetime` на сервере и передаем его методу `MeterMT.Manager.submit_reading()`.

В этом подразделе приведен весь код RPyC-сервера счетчиков. Он был бы на несколько строк короче, если бы мы опустили методы `on_connect()` и `on_disconnect()`.

6.2.3. Разработка клиента RPyC

Клиент RPyC создается почти так же, как клиент XML-RPC, так что в этом подразделе мы остановимся только на различиях.

6.2.3.1. Консольный клиент RPyC

Как и в случае XML-RPC, для работы клиента RPyC необходимо, чтобы сервер был уже запущен.

Программы `meterclient-rpyc.py` и `meterclient-rpc.py` (подраздел 6.1.3.1) отличаются только функциями `main()` и `submit()`.

```
def main():
   username, password = login()
   if username is not None:
      try:
         service = rpyc.connect(HOST, PORT)
         manager = service.root
         sessionId, name = manager.login(username, password)
         print("Welcome, {}, to Meter RPYC".format(name))
         interact(manager, sessionId)
      except ConnectionError as err:
         print("Error: Is the meter server running? {}".format(err))
```

Первое различие состоит в том, что мы жестко зашили имя хоста и номер порта. Разумеется, их можно было бы задавать в командной строке, как в клиенте XML-RPC. Кроме того, вместо того чтобы сначала создать заместитель менеджера, а затем подключиться, мы начинаем с подключения к серверу, предоставляющему службы. В данном случае сервер предоставляет всего одну службу (`MeterService`), и ее мы можем использовать как заместитель менеджера счетчиков. Весь остальной код – аутентификация, получение задач, отправка показаний и получение статуса – остался тем же самым с одним исключением: функция `submit()` перехватывает не те исключения, что в клиенте XML-RPC.

Синхронизация имен хостов и номеров портов может вызвать затруднения, особенно если из-за конфликта приходится использовать не тот номер порта, к которому мы привыкли. Эту проблему можно решить с помощью сервера регистрации. Для этого мы должны запустить на какой-нибудь машине в сети сервер `registry_server.py`, пос-

тавляемый в составе RPyC. Серверы RPyC автоматически ищут этот сервер при запуске и, если находят, регистрируют в нем свои службы. После этого клиенты может вызывать не `rpyc.connect(host, port)`, а `rpyc.connect_by_service(service)`, например: `rpyc.connect_by_service("Meter")`.

6.2.3.2. Графический клиент RPyC

Внешний вид графического клиента RPyC, `meter-rpyc.pyw`, показан на рис. 6.2. На самом деле, графические клиенты RPyC и XML-RPyC, запущенные на одной и той же платформе, визуально неотличимы.

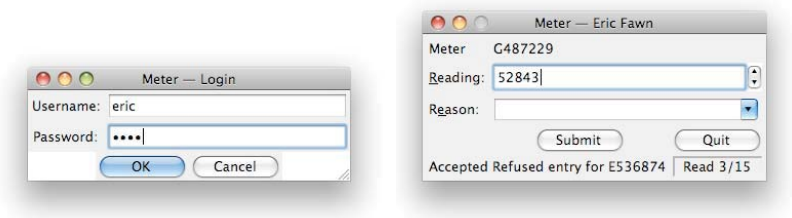

Рис. 6.2. Окно аутентификации и главное окно графического клиента RPyC в OS X

Для создания графического клиента RPyC на основе Tkinter, который автоматически обращается к уже запущенному серверу счетчиков или запускает сервер самостоятельно, нужно сделать почти то же самое, что в случае XML-RPC. Различия сводятся к двум методам, другим предложениям импорта, немного измененным константам и обработке других исключений в блоках `except`.

```
def connect(self, username, password):
   try:
      self.service = rpyc.connect(HOST, PORT)
   except ConnectionError:
      filename = os.path.join(tempfile.gettempdir(),
               "M{}.$$$".format(random.randint(1000, 9999)))
      self.serverPid = subprocess.Popen([sys.executable, SERVER,
               filename]).pid
      self.wait_for_server(filename)
      try:
         self.service = rpyc.connect(HOST, PORT)
      except ConnectionError:
         self.handle_error("Failed to start the RPYC Meter server")
         return False
```

```
        self.manager = self.service.root
        return self.login_to_server(username, password)
```

Этот метод вызывается после получения имени и пароля от диалогового окна аутентификации. Он подключается к серверу счетчиков и пытается аутентифицировать пользователя.

В случае ошибки подключения мы предполагаем, что сервер не запущен и пытаемся запустить его, передав имя файла уведомления. Сервер запускается без блокировки (то есть асинхронно), но мы должны дождаться завершения запуска, перед тем как пытаться подключиться к нему. Метод `wait_for_server()` отличается от рассмотренного ранее (стр. 248) только тем, что возбуждает исключение `ConnectionError`, а не вызывает `handle_error()` самостоятельно. Если соединение установлено, то мы получаем заместитель менеджера счетчиков и пытаемся аутентифицировать пользователя.

```
def login_to_server(self, username, password):
    try:
        self.sessionId, name = self.manager.login(username, password)
        self.master.title("Meter \u2014 {}".format(name))
        return True
    except rpyc.core.vinegar.GenericException as err:
        self.handle_error(err)
        return False
```

Если учетные данные пользователя правильны, то мы получаем идентификатор сеанса и отображаем имя пользователя в полосе заголовка приложения. В противном случае мы возвращаем `False`, в результате чего приложение завершается (вместе с сервером, если оно же его и запускало).

Ни в одном из примеров в этой главе не использовалось шифрование, поэтому теоретически весь трафик между клиентом и сервером можно перехватить. Это несущественно, если приложения не обмениваются секретными данными или если клиент и сервер работают на одной машине или если оба находятся за брандмауэром или если зашифровано само сетевое соединение. Но если шифрование требуется, то его нетрудно обеспечить. В случае XML-RPC можно, например, воспользоваться сторонним пакетом PyCrypto (`www.dlitz.net/software/pycrypto`), чтобы шифровать все передаваемые по сети данные. Другой способ – использовать протокол Transport Layer Security («безопасные сокеты»), который поддерживается Python-

модулем `ssl`. В случае RPyC обеспечить безопасность гораздо легче, поскольку она уже встроена. RPyC может работать с ключами и сертификатами или через SSH-туннель, что еще проще.

В Python имеется отличная поддержка сетевого программирования на всех уровнях – от низкого до высокого. В стандартную библиотеку включены модули для большинства популярных высокоуровневых протоколов: FTP для передачи файлов; POP3, IMAP4 и SMTP для электронной почты; HTTP и HTTPS для веб и, конечно, TCP/IP и другие протоколы на уровне сокетов. Модуль среднего уровня `socketserver` можно использовать как основу для разработки серверов, хотя поддерживаются также средства более высокого уровня, например: модуль `smtpd` для создания почтовых серверов, модуль `http.server` – для веб-серверов и модуль `xmlrpc.server`, с которым мы познакомились в этой главе, – для создания серверов XML-RPC.

Существует также много сторонних сетевых модулей, особенно для разработки веб-каркасов, поддерживающих шлюзовой интерфейс веб-серверов для Python (WSGI, см. `www.python.org/dev/peps/pep-3333`). Дополнительные сведения о сторонних веб-каркасах для Python см. на странице `wiki.python.org/moin/WebFrameworks`, а о веб-серверах – на странице `wiki.python.org/moin/WebServers`.

ГЛАВА 7. Графические интерфейсы пользователя на Python и Tkinter

Хорошо спроектированное приложение с графическим интерфейсом пользователя (ГИП) может предложить пользователю удобные, красивые и современные средства взаимодействия. И чем сложнее приложение, тем больше оно выигрывает от удачного ГИП, особенно если он включает специально «заточенные» под приложение виджеты[1]. Для сравнения отметим, что веб-приложения, в которых, помимо виджетов самого приложения, есть еще меню и панели инструментов браузера, может здорово запутать пользователя. И до тех пор пока холст HTML5 не получит широкого распространения, веб-приложения будут иметь весьма ограниченные средства представления нестандартных виджетов. К тому же, веб-приложения не могут состязаться с платформенными в производительности.

Пользователи смартфонов все чаще получают возможность взаимодействовать с приложениями голосом, но в настольных компьютерах, ноутбуках и планшетах выбор все еще ограничен в основном мышью, клавиатурой и сенсорным экраном. На момент написания этой книги почти для всех сенсорных устройств необходимы библиотеки от производителя и особые языки и инструменты. К счастью, имеется сторонняя библиотека с открытым исходным кодом Kivy (`kivy.org`), которая поддерживает разработку кросс-платформенных сенсорных приложений на Python. Разумеется, это не отменяет того факта, что сенсорные интерфейсы главным образом проектируются для машин с ограниченной вычислительной мощностью и небольшими экранами, на которых в каждый момент времени присутствует только одно приложение.

[1] Программисты Windows часто применяют для описания объектов ГИП термины «элемент управления», «контейнер» или «форма». В этой книге используется универсальный термин *виджет*, заимствованный из программирования ГИП в Unix.

Пользователи настольных компьютеров хотят в полной мере наслаждаться преимуществами большого экрана и мощного процессора, а для этой цели лучше всего подходят традиционные приложения с ГИП. К тому же, управление голосом – например, в современных версиях Windows – проектируется для работы совместно с существующими графическими приложениями. Кросс-платформенными можно сделать не только консольные программы на Python, но и приложения с ГИП при условии, что используется подходящая библиотека ГИП. Таких библиотек несколько. Ниже дан краткий обзор основных четырех, все они перенесены на Python 3 и работают по крайней мере в Linux, OS X и Windows, где обеспечивают стандартный для платформы внешний вид.

- **PyGtk и PyGObject.** PyGtk (`www.pygtk.org`) – стабильная и успешная библиотека. Однако ее разработка прекратилась в 2011 году, когда предпочтение было отдано пришедшей на смену технологии PyGObject (`live.gnome.org/PyGObject`). К сожалению, на момент написания этой книги PyGObject еще нельзя было назвать кросс-платформенной библиотекой, т. к. все усилия направлялись на разработку для систем на базе Unix.
- **PyQt4 и PySide.** PyQt4 (`www.riverbankcomputing.co.uk`) предоставляет интерфейс между Python и каркасом для разработки графических приложений Qt 4 (`qtproject.org`). PySide (`www.pyside.org`) – появившийся позднее проект, обладающий высокой степенью совместимости с PyQt4, но распространяемый на условиях более либеральной лицензии. PyQt4, пожалуй, самая стабильная и зрелая кросс-платформенная библиотека ГИП для Python[2]. (Ожидается, что в 2013 году выйдут версии PyQt и PySide с поддержкой Qt 5.)
- **Tkinter.** Tkinter реализует интерфейс к библиотеке ГИП Tcl/Tk (`www.tcl.tk`). Обычно Python 3 поставляется в комплекте с Tcl/Tk 8.5, хотя, начиная с версии Python 3.4, планируется перейти на Tcl/Tk 8.6. Tkinter гораздо проще других упомянутых здесь библиотек, в ней нет встроенной поддержки панелей инструментов, стыкуемых окон или строки состояния (хотя все это можно добавить). Кроме того, если другие библиотеки автоматически работают с многими платформенными средст-

[2] Раскрытие информации: когда-то автор руководил составлением документации по Qt и написал книгу «Rapid GUI Programming with Python and Qt» (см. краткую библиографию).

вами – например, универсальной полосой меню, имеющейся в OS X, – Tkinter (по крайней мере, для Tcl/Tk 8.5) требует, чтобы платформенные различия учитывал сам программист. Основное достоинство Tkinter – тот факт, что она поставляется вместе с Python, и по сравнению с другими пакетами очень невелика.

- **wxPython.** Библиотека wxPython (`www.wxpython.org`) реализует интерфейс из Python к библиотеке wxWidgets (`www.wxwidgets.org`). Она существует уже много лет, но после выхода Python 3 подверглась значительной переделке, результаты который должны стать доступны к моменту выхода этой книги из печати.

За исключением PyGObject, все перечисленные выше библиотеки предоставляют все необходимое для создания кросс-платформенных приложений с ГИП на Python. Если нас интересует только одна какая-то платформа, то почти наверняка существует интерфейс из Python к «родным» для этой платформам библиотекам ГИП (см. `wiki.python.org/moin/GuiProgramming`) или можно использовать ориентированный на эту платформу интерпретатор Python, например, Jython или IronPython. Трехмерную графику библиотеки ГИП обычно тоже поддерживает. Но можно вместо этого обратиться к проекту PyGame (`www.pygame.org`) или, если требуется что-нибудь попроще, напрямую воспользоваться интерфейсом между Python и OpenGL, как будет показано в следующей главе.

Поскольку Tkinter входит в стандартную поставку, то написанные с ее помощью приложения легко развертывать (при необходимости можно даже включить в дистрибутив Python и Tcl/Tk; см., например, `cx-freeze.sourceforge.net`). Такие приложения выглядят более привлекательно и с ними проще работать, чем с консольными программами. Пользователи, особенно в OS X и Windows, принимают их с большей охотой.

В этой главе мы приведем три примера приложений: крохотную программку «hello world», небольшой конвертер валют и относительно большую игру Gravitate. Эту игру можно рассматривать как вариацию на тему TileFall/SameGame, в которой плитки собираются в центре, заполняя свободное пространство, а не падают со сдвигом влево. На примере Gravitate показано, как создать приложение Tkinter с главным окном и такими современными аксессуарами, как меню, диалоги и строка состояния. В разделе 7.2.2 мы рассмотрим два диалоговых окна Gravitate, а в разделе 7.3 – инфраструктуру ее главного окна.

7.1. Введение в Tkinter

Программирование ГИП не труднее любого другого специализированного вида программирования, зато автор получает удовлетворение от создания профессионально выглядящих приложений, которые доставляют радость людям.

Отметим, однако, что тема программирования ГИП настолько обширна, что сколько-нибудь глубоко раскрыть ее в одной главе нет никакой возможности, для этого нужна целая книга. Но мы все же можем рассмотреть некоторые ключевые аспекты разработки программ с ГИП и, в частности, поговорить о том, как восполнить недостаток ряда средств в Tkinter. А для начала напишем классическую программу «hello world», в данном случае `hello.pyw`, внешний вид которой показан на рис. 7.1.

Рис. 7.1. Приложение Hello в виде диалогового окна в Linux, OS X и Windows

```
import tkinter as tk
import tkinter.ttk as ttk

class Window(ttk.Frame):

   def __init__(self, master=None):
      super().__init__(master) # Создается self.master
      helloLabel = ttk.Label(self, text="Hello Tkinter!")
      quitButton = ttk.Button(self, text="Quit", command=self.quit)
      helloLabel.pack()
      quitButton.pack()
      self.pack()

window = Window() # Неявно создается объект tk.Tk
window.master.title("Hello")
window.master.mainloop()
```

Выше приведен полный код приложения `hello.pyw`. Многие программисты, работающие с Tkinter, импортируют все имена из библиотеки (например, `from tkinter import *`), но мы предпочитаем использовать пространства имен (хотя и сокращенные: `tk` и `ttk`), чтобы было понятно, где какое имя определено. (Кстати говоря, модуль `ttk` – это обертка вокруг официального расширения Tcl/Tk –

Ttk «Tile».) Можно было бы оставить только первое предложение импорта и писать `tkinter.Frame` вместо `tkinter.ttk.Frame` и т. д., но `tkinter.ttk` обеспечивает поддержку тем, поэтому лучше его использовать, особенно в OS X и Windows.

Для большинства простых виджетов `tkinter` существуют также версии `tkinter.ttk` с поддержкой тем. Не всегда у них одинаковый интерфейс, и существуют ситуации, когда можно использовать только простые виджеты, поэтому читайте документацию. (Тем, кто понимает код на Tcl/Tk, мы рекомендуем документацию на сайте `www.tcl.tk`, а всем остальным – документацию на сайте `www.tkdocs.com`, где имеются примеры на Python и некоторых других языках, а также на сайте `infohost.nmt.edu/tcc/help/pubs/tkinter/web`, где есть полезный учебник и одновременно справочник по Tkinter.) Существует также несколько виджетов `tkinter.ttk` с поддержкой тем, у которых нет простых аналогов, например: `tkinter.ttk.Combobox`, `tkinter.ttk.Notebook` и `tkinter.ttk.Treeview`.

В этой книге мы придерживаемся стиля программирования ГИП, при котором каждому окну соответствует один класс, обычно в отдельном модуле. Окно верхнего уровня (то есть главное окно приложения) принято наследовать от `tkinter.Toplevel` или `tkinter.ttk.Frame`, как мы и поступили выше. Tkinter поддерживает иерархию владения (родитель-потомок, или главный-подчиненный). Вообще говоря, нам до этого нет дела, коль скоро мы вызываем встроенную функцию `super()` в методе `__init__()` любого созданного нами класса, наследующего виджету.

Большинство приложений ГИП устроено по одному принципу: создать один или несколько оконных классов, один из которых соответствует главному окну приложения. В каждом оконном классе нужно завести переменные окна (в `hello.pyw` таковых нет), создать виджеты, расположить их и задать методы, вызываемые в ответ на события (щелчок мышью, нажатие клавиши, таймаут и т. д.). В данном случае мы ассоциировали нажатие кнопки `quitButton` с унаследованным методом `tkinter.ttk.Frame.quit()`, который закрывает окно, а поскольку это единственное окно верхнего уровня в данном приложении, то приложение при этом завершается. Разработав все оконные классы, мы должны выполнить последний шаг – создать объект приложения (в этом примере делается неявно) и запустить цикл обработки событий ГИП. С этим циклом мы познакомились раньше (см. рис. 4.8).

Естественно, большинство приложений с ГИП гораздо длиннее и сложнее, чем `hello.pyw`. Однако оконные классы в них принципи-

ально устроены так же, как описано здесь, только виджетов и событий куда больше.

В большинстве современных библиотек ГИП принято использовать менеджеры компоновки, а не зашивать размеры и позиции виджетов в код. Тогда виджеты могут автоматически увеличиваться или уменьшаться в размерах, адаптируясь к своему содержимому (например, тексту в метке или на кнопке), даже если оно изменяется, и сохраняя положение относительно других виджетов. Заодно менеджеры компоновки избавляют программиста от необходимости выполнять утомительные вычисления.

В Tkinter есть три менеджера компоновки: абсолютный (с жестко зашитыми позициями, используется редко), упаковщик (виджеты располагаются вокруг воображаемой центральной зоны) и сетка (виджеты размещаются в ячейках прямоугольной сетки, самый популярный). В этом примере мы упаковали метку и кнопку друг за другом, а затем упаковали все окно. Упаковка вполне приемлема для таких простых окон, но работать с сеткой проще всего, как мы увидим в примерах ниже.

Приложения с ГИП можно разбить на две широких категории: диалоговые и с главным окном. У диалогового приложения нет ни меню, ни панелей инструментов; оно управляется кнопками, комбинированными списками и т. п. Такой стиль идеален для приложений с простым пользовательским интерфейсом: небольших утилит, медиаплееров и некоторых игр. У приложения с главным окном обычно имеются меню и панели инструментов, расположенные выше центральной области, и строка состояния под ней. Могут также присутствовать стыкуемые окна. Главные окна идеальны для более сложных приложений, когда пункты меню и кнопки на панели инструментов служат для открытия дополнительных диалоговых окон. Мы рассмотрим приложения обоих типов, но начнем с диалоговых, потому что почти все, что мы узнаем о них, применимо и к диалогам в приложениях с главным окном.

7.2. Создание диалоговых окон с помощью Tkinter

У диалоговых окон есть четыре модальности и несколько уровней «интеллектуальности». Сначала рассмотрим модальности.

- **Глобально модальное.** Глобально модальное окно блокирует весь пользовательский интерфейс – не только данного, но и

всех вообще приложений – так что пользователь может взаимодействовать только с ним. Два типичных случая использования таких окон – вход в систему при запуске машины и разблокировка экрана, защищенного запароленным хранителем. Прикладные программисты не должны использовать глобально модальные окна, потому что в случае ошибки вся машина становится недоступной.

- **Локально модальное (модальное на уровне приложения).** Локально модальное окно не дает взаимодействовать ни с каким другом окном в приложении. Но пользователь может переключиться на другое приложение. Модальные окна проще программировать, чем немодальные, т. к. пользователь не может «за спиной» программиста изменить состояние приложения. Однако некоторые пользователи считают, что такие окна неудобны.
- **Оконно модальные.** Оконно модальное окно очень похоже на локально модальное, но препятствует взаимодействию не со всеми прочими окнами приложения, а только с окнами в той же иерархии окон. Это полезно, например, когда пользователь открывает два окна документа верхнего уровня, поскольку мы не хотим, чтобы открытый диалог в одном из таких окон мешал работать с другим окном.
- **Немодальное.** Немодальные диалоги не препятствуют взаимодействию с другими окнами как в своем, так и в других приложениях. Вообще говоря, у программиста гораздо больше хлопот с немодальными окнами, так как необходимо учитывать, что в результате действий пользователя может измениться состояние, от которого немодальное окно зависит.

В терминологии Tcl/Tk глобально модальное окно осуществляет *глобальный захват*, а локально и оконно модальные окна (которые принято называть просто модальными) – *локальный захват*. В Tkinter на платформе OS X некоторые модальные окна отображаются как листы.

Пассивное, или «глупое» (dumb) диалоговое окно просто показывает пользователю виджеты и предоставляет все введенные в них данные приложению. Такие диалоги ничего не знают о логике приложения. Типичный пример – диалог входа в приложение, который принимает имя и пароль пользователя и передает их приложению (пример такого диалога мы видели в предыдущей главе в подразделе 6.1.3.2, код находится в файле `MeterLogin.py`).

Активное, или «умное» (smart) диалоговое окно кое-что знает о приложении. Возможно, ему даже передаются ссылки на переменные или структуры данных приложения, чтобы оно могло манипулировать ими непосредственно.

Модальные диалоги могут быть пассивными, активными и промежуточными. Окно можно назвать сравнительно «умным», если оно знает о приложении достаточно, чтобы осуществлять валидацию не только отдельных элементов данных в своих виджетах, но и их сочетания. Например, сравнительно «умное» диалоговое окно для ввода начальной и конечной даты не примет конечную дату, если она раньше начальной.

Немодальные диалоги почти всегда активные. Они бывают двух видов: применить/закрыть и динамичные. Диалог применить/закрыть позволяет пользователю что-то сделать с виджетами, а затем нажать кнопку **Применить** и увидеть результаты в главном окне. Динамичный диалог применяет изменения по мере того, как пользователь взаимодействует с виджетами; такие окна повсеместно встречаются в OS X. Диалоги «поумнее» предлагают механизм отмены/повтора или кнопку **Восстановить значения по умолчанию** (которая заполняет виджеты значениями параметров приложения, подразумеваемыми по умолчанию), а иногда и кнопку **Отменить изменения** (чтобы восстановить те значения, которые были в виджетах в момент открытия окна). Бывают и пассивные немодальные окна, которые только предоставляют информацию, например, окно «Справка». Как правило, в них есть кнопка **Закрыть**.

Немодальные диалоги особенно полезны для изменения цветов, шрифтов, форматов и шаблонов, потому что позволяют оценить эффект изменения, попробовать по-другому и так много раз. Если бы мы в таких случаях использовали модальные окна, то пришлось быть открыть окно, внести изменения, нажать кнопку **OK** и повторить весь цикл, пока не достигнем желаемого результата.

Главное окно диалогового приложения по сути дела представляет собой немодальный диалог. В приложениях с главным окном могут быть модальные и немодальные диалоги, появляющиеся в ответ на выбор пользователем пункта меню или нажатие кнопки на панели инструментов.

7.2.1. Создание диалогового приложения

В этом подразделе мы рассмотрим очень простое и вместе с тем полезное диалоговое приложение для конвертации валют. Его исход-

ный код находится в каталоге `currency`, а внешний вид показан на рис. 7.2.

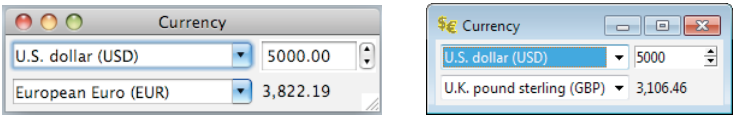

Рис. 7.2. Диалоговое приложение Currency в OS X и Windows

В приложении есть два комбинированных списка с названиями и обозначениями валют, счетчик для ввода денежной суммы и метка, в которой показывается результат конвертации этой суммы из верхней валюты в нижнюю.

Код приложения распределен по трем Python-файлам: `currency.pyw` – исполняемая программа; `Main.py` – класс `Main.Window`; `Rates.py` – функция `Rates.get()`, которая рассматривалась ранее (раздел 1.5). И еще есть два значка приложения для Linux и Windows: `currency/images/icon_16x16.gif` и `currency/images/icon_32x32.gif`.

Файлы приложений с ГИП на Python могут иметь стандартное расширение `.py`, но в OS X и в Windows расширение `.pyw` часто ассоциируется с другим интерпретатором Python (например, `pythonw.exe`, а не `python.exe`). Этот интерпретатор позволяет запускать приложение без окна консоли, что гораздо больше нравится пользователям. Программистам же лучше исполнять приложения с ГИП из окна консоли и работать со стандартным интерпретатором, потому что так они видят, что выводится на `sys.stdout` и `sys.stderr`, – вещь, очень полезная при отладке.

7.2.1.1. Функция main() приложения для конвертации валют

Для больших программ в особенности рекомендуется создать очень маленький «исполняемый» модуль, а весь остальной код вынести в отдельные `py`-файлы с модулями (их размер особого значения не имеет). При первом запуске программы на быстрых современных машинах может показаться, что это несущественно. Однако во время этого первого запуска все `py`-файлы (кроме «исполняемого») компилируются в байт-код – файлы с расширением `.pyc`. При последующих запусках Python будет использовать `pyc`-файлы (если только соответствующий `py`-файл не изменился), поэтому программа будет запускаться гораздо быстрее, чем в первый раз.

Исполняемый файл `currency.pyw` содержит одну небольшую функцию `main()`.

```
def main():
   application = tk.Tk()
   application.title("Currency")
   TkUtil.set_application_icons(application, os.path.join(
         os.path.dirname(os.path.realpath(__file__)), "images"))
   Main.Window(application)
   application.mainloop()
```

Эта функция первым делом создает «объект приложения» Tkinter. На самом деле, это невидимое при обычных обстоятельствах окно верхнего уровня, которое служит предком всех виджетов (его также называют главным или корневым виджетом). В приложении `hello.pyw` мы позволили Tkinter создать этот объект неявно, но вообще-то лучше делать это самостоятельно, чтобы можно было применить настройки на уровне приложения. Например, здесь мы устанавливаем название приложения «Currency».

В примерах к этой книге имеется модуль `TkUtil`, который включает несколько вспомогательных функций для поддержки программирования Tkinter, а также ряд модулей, которые мы обсудим, когда возникнет надобность. Здесь мы пользуемся функцией `TkUtil.set_application_icons()`.

Установив название и значки приложения (правда, значки в OS X игнорируются), мы создаем главное окно, передавая ему объект приложения в качестве родителя, а затем запускаем цикл обработки событий ГИП. Приложение завершится, когда завершится этот цикл, например, если мы вызовем функцию `tkinter.Tk.quit()`.

```
def set_application_icons(application, path):
   icon32 = tk.PhotoImage(file=os.path.join(path, "icon_32x32.gif"))
   icon16 = tk.PhotoImage(file=os.path.join(path, "icon_16x16.gif"))
   application.tk.call("wm", "iconphoto", application, "-default",
         icon32, icon16)
```

Для полноты здесь приведена функция `TkUtil.set_application_icons()`. Класс `tk.PhotoImage` умеет загружать растровые изображения в форматах PGM, PPM и GIF (поддержка PNG ожидается в версии Tcl/Tk 8.6). Создав два изображения, мы вызываем функцию `tkinter.Tk.tk.call()`, которой по сути дела передаем команду Tcl/Tk. По возможности следует избегать опускания на такой низкий уровень, но иногда это необходимо, если Tkinter не предоставляет интерфейса к нужной функциональности.

7.2.1.2. Класс Main.Window приложения для конвертации валют

Главное окно приложения устроено по описанному выше принципу, который отчетливо прослеживается в обращениях внутри метода `__init__()`. Весь приведенный в этом разделе код взят из файла `currency/Main.py`.

```
class Window(ttk.Frame):

    def __init__(self, master=None):
        super().__init__(master, padding=2)
        self.create_variables()
        self.create_widgets()
        self.create_layout()
        self.create_bindings()
        self.currencyFromCombobox.focus()
        self.after(10, self.get_rates)
```

При инициализации класса, наследующего виджету, обязательно нужно вызывать встроенную функцию `super()`. Здесь мы передаем не только параметр `master` («объект приложения» `tk.Tk`, полученный от функции `main()`), но и величину отступа – 2 пикселя. Это расстояние между внутренней границей окна приложения и расположенными внутри него виджетами.

Далее мы создаем переменные и виджеты окна (то есть приложения) и размещаем виджеты. После этого мы привязываем обработчики к событиям, а затем передаем фокус клавиатуры верхнему комбинированному списку, так чтобы пользователь мог изменить исходную валюту. Наконец, мы вызываем унаследованный метод Tkinter `after()`, принимающий время в миллисекундах и вызываемый объект, который будет вызван по прошествии указанного времени.

Курсы валют мы загружаем из Интернета, и на это может уйти несколько секунд. Но мы хотим, чтобы приложение появилось на экране сразу (иначе пользователь подумает, что оно не запустилось, и попробует запустить еще раз). Поэтому получение курсов мы немного откладываем, чтобы у приложения было время показать себя.

```
    def create_variables(self):
        self.currencyFrom = tk.StringVar()
        self.currencyTo = tk.StringVar()
        self.amount = tk.StringVar()
        self.rates = {}
```

Тип `tkinter.StringVar` предназначен для хранения строк, которые можно ассоциировать с виджетами. При изменении переменной типа `StringVar` автоматически изменяется ассоциированный виджет и наоборот. Можно было бы определить `self.amount` как `tkinter.IntVar`, но поскольку на внутреннем уровне Tcl/Tk работает почти исключительно со строками, часто удобнее использовать строки, даже если нужно представить числа. `rates` – это словарь, в котором ключами являются названия валют, а значениями – обменные курсы.

```
Spinbox = ttk.Spinbox if hasattr(ttk, "Spinbox") else tk.Spinbox
```

Виджет `tkinter.ttk.Spinbox` не был включен в библиотеку Tkinter, поставляемую вместе с Python 3, но, будем надеяться, появится в Python 3.4. Показанный фрагмент кода позволяет воспользоваться этим виджетом, если он имеется, а в противном случае ограничиться счетчиком без поддержки тем. Интерфейс у них разный, поэтому нужно следить за тем, чтобы использовать только общие для обоих классов методы.

```
def create_widgets(self):
    self.currencyFromCombobox = ttk.Combobox(self,
            textvariable=self.currencyFrom)
    self.currencyToCombobox = ttk.Combobox(self,
            textvariable=self.currencyTo)
    self.amountSpinbox = Spinbox(self, textvariable=self.amount,
            from_=1.0, to=10e6, validate="all", format="%0.2f",
            width=8)
    self.amountSpinbox.config(validatecommand=(
            self.amountSpinbox.register(self.validate), "%P"))
    self.resultLabel = ttk.Label(self)
```

У каждого виджета, кроме объекта `tk.Tk`, должен быть родитель; обычно им является окно или фрейм, внутри которого размещается виджет. В данном случае мы создаем два комбинированных списка и ассоциируем с каждым свой объект `StringVar`.

Кроме того, мы создаем счетчик, также ассоциированный с `StringVar`, для которого задаем минимум и максимум. Параметр `width` счетчика измеряется в символах, а параметр `format` задается в старом формате Python 2 со знаком `%` (эквивалентен форматной строке `"{:0.2f}"` для функции `str.format()`). Параметр `validate` говорит, что нужно осуществлять проверку при любом изменении значения – будь то прямой ввод числа или использование кнопок уменьшения и увеличения. Создав счетчик, мы регистрируем вызы-

ваемый объект, отвечающий за проверку. При вызове ему будет передан аргумент, соответствующий указанному формату (`"%P"`); это форматная строка Tcl/Tk, а не Python. Кстати, значению счетчика автоматически присваивается минимальное заданное значение (`from_`), которое в данном случае равно 1,0, если иное не установлено явно.

Наконец, мы создаем метку, в которой будет отображаться вычисленная денежная сумма. Начальный текст для нее не задается.

```
def validate(self, number):
   return TkUtil.validate_spinbox_float(self.amountSpinbox, number)
```

Это именно тот вызываемый объект, который зарегистрирован для счетчика. В этом контексте формат `"%P"` Tcl/Tk обозначает текст счетчика. Таким образом, при любом изменении значения счетчика вызывается этот метод, которому передается текст счетчика. Собственно проверка поручается общей вспомогательной функции из модуля `TkUtil`.

```
def validate_spinbox_float(spinbox, number=None):
   if number is None:
      number = spinbox.get()
   if number == "":
      return True
   try:
      x = float(number)
      if float(spinbox.cget("from")) <= x <= float(spinbox.cget("to")):
         return True
   except ValueError:
      pass
   return False
```

Эта функция ожидает получить счетчик и значение числа (в виде строки) или `None`. Если значение не передано, то функция берет текст из счетчика. Она возвращает `True` («допустимое»), если счетчик пуст, чтобы дать пользователю возможность очистить значение и начать вводить его заново. В противном случае функция пытается преобразовать текст в число с плавающей точкой и проверяет, что оно попадает в заданный для счетчика диапазон.

У всех виджетов Tkinter имеется метод `config()`, который принимает один или несколько аргументов вида *`key=value`*, устанавливающих атрибуты виджета, а также метод `cget()`, который принимает аргумент *`key`* и возвращает ассоциированное с этим ключом значение. Имеется также метод `configure()`, но это просто псевдоним `config()`.

```
def create_layout(self):
    padWE = dict(sticky=(tk.W, tk.E), padx="0.5mm", pady="0.5mm")
    self.currencyFromCombobox.grid(row=0, column=0, **padWE)
    self.amountSpinbox.grid(row=0, column=1, **padWE)
    self.currencyToCombobox.grid(row=1, column=0, **padWE)
    self.resultLabel.grid(row=1, column=1, **padWE)
    self.grid(row=0, column=0, sticky=(tk.N, tk.S, tk.E, tk.W))
    self.columnconfigure(0, weight=2)
    self.columnconfigure(1, weight=1)
    self.master.columnconfigure(0, weight=1)
    self.master.rowconfigure(0, weight=1)
    self.master.minsize(150, 40)
```

Этот метод размещает виджеты, как показано на рис. 7.3. Каждый виджет помещается в отдельную ячейку сетки и привязывается к восточной и западной границе, то есть при изменении размера окна виджет будет сжиматься и растягиваться по горизонтали, но его высота будет оставаться неизменной. Отступ между виджетом и линиями сетки в направлениях *x* и *y* составляет 0,5 мм (миллиметра), то есть каждый виджет окружен пустым пространством шириной 0,5 мм (см. врезку «Распаковка последовательностей и отображений» на стр. 24).

(0, 0) ***currencyFromCombobox***	(0, 1) ***amountSpinbox***
(1, 0) ***currencyToCombobox***	(1, 1) ***resultLabel***

Рис. 7.3. Компоновка главного окна приложения для конвертации валют

После того как все виджеты размещены, производится компоновка самого окна – в сетке из одной ячейки, так что оно будет сжиматься и растягиваться во всех направлениях. Затем задаются веса столбцов, то есть коэффициенты растяжения. Так, в данном случае, если окно растягивается по горизонтали, то на каждый дополнительный пиксель счетчика и метки оба комбинированных списка получат по два пикселя. Ненулевые веса присваиваются также единственной ячейке сетки, в которой размещено само окно; это позволяет изменять размеры его содержимого. И наконец, задается разумный минимальный размер окна, иначе пользователь смог бы уменьшить его практически до нуля.

```
def create_bindings(self):
    self.currencyFromCombobox.bind("<<ComboboxSelected>>",
            self.calculate)
    self.currencyToCombobox.bind("<<ComboboxSelected>>",
```

```
            self.calculate)
        self.amountSpinbox.bind("<Return>", self.calculate)
        self.master.bind("<Escape>", lambda event: self.quit())
```

Этот метод нужен для привязки событий к действиям. В данном случае имеются события двух видов: «виртуальные», то есть генерируемые теми или иными виджетами, и «реальные», соответствующие тому, что происходит в пользовательском интерфейсе, например, нажатию на клавишу или изменению размера окна. Имена виртуальных событий заключаются в двойные угловые скобки, а реальных – в одинарные угловые скобки.

При изменении выбранного значения комбинированный список помещает в очередь событий виртуальное событие `<<ComboboxSelected>>`. Для обоих списков мы решили привязать это событие к методу `self.calculate()`, который пересчитывает конвертированную сумму. Что касается счетчика, то сумма пересчитывается только после нажатия клавиши `Enter`. Наконец, если пользователь нажимает клавишу `Esc`, то мы завершаем приложение, вызывая унаследованный метод `tkinter.ttk.Frame.quit()`.

```
def calculate(self, event=None):
    fromCurrency = self.currencyFrom.get()
    toCurrency = self.currencyTo.get()
    amount = self.amount.get()
    if fromCurrency and toCurrency and amount:
        amount = ((self.rates[fromCurrency] / self.rates[toCurrency]) *
                  float(amount))
        self.resultLabel.config(text="{:,.2f}".format(amount))
```

Этот метод получает обе валюты и подлежащую конвертации денежную сумму, после чего выполняет конвертацию. В конце он записывает результат в текст метки `resultLabel` с запятыми в качестве разделителей тысяч и двумя знаками после запятой.

```
def get_rates(self):
    try:
        self.rates = Rates.get()
        self.populate_comboboxes()
    except urllib.error.URLError as err:
        messagebox.showerror("Currency \u2014 Error", str(err),
            parent=self)
        self.quit()
```

Этот метод вызывается после срабатывания таймера, чтобы у окна было время нарисовать себя. Он получает словарь курсов (ключ – на-

звание валюты, значение – обменный курс) и заполняет оба комбинированных списка. Если получить курсы не удается, то выводится окно с сообщением об ошибке, и, когда пользователь его закрывает (например, нажатием кнопки OK), приложение завершается.

Функция `tkinter.messagebox.showerror()` принимает текст заголовка окна, текст сообщения и необязательного родителя (если родитель задан, то окно сообщения располагается в его центре). Поскольку в файлах Python 3 используется кодировка UTF-8, то можно было бы использовать символ длинного тире (–) непосредственно, но в моноширинном шрифте, которым набран код в этой книге, такого символа нет, поэтому пришлось воспользоваться управляющей последовательностью Unicode.

```
def populate_comboboxes(self):
    currencies = sorted(self.rates.keys())
    for combobox in (self.currencyFromCombobox,
                     self.currencyToCombobox):
        combobox.state(("readonly",))
        combobox.config(values=currencies)
    TkUtil.set_combobox_item(self.currencyFromCombobox, "USD", True)
    TkUtil.set_combobox_item(self.currencyToCombobox, "GBP", True)
    self.calculate()
```

Этот метод заполняет комбинированные списки, помещая в них названия валют в алфавитном порядке. Оба списка делаются доступными только для чтения. После этого мы устанавливаем в верхнем списке доллары США, а в нижнем – английские фунты стерлингов. Наконец, мы вызываем метод `self.calculate()`, чтобы получить начальную конвертированную сумму.

У каждого виджета Tkinter, поддерживающего темы, имеется метод `state()`, который устанавливает одно или несколько состояний, и метод `instate()`, который проверяет, находится ли виджет в конкретном состоянии. Чаще всего употребляются состояния `"disabled"`, `"readonly"` и `"selected"`.

```
def set_combobox_item(combobox, text, fuzzy=False):
    for index, value in enumerate(combobox.cget("values")):
        if (fuzzy and text in value) or (value == text):
            combobox.current(index)
            return
    combobox.current(0 if len(combobox.cget("values")) else -1)
```

Эта общая функция находится в модуле `TkUtil`. Она пытается сделать текущим в списке значение, которое либо совпадает с указанным

текстом, либо – если параметр `fuzzy` равен `True` – содержит указанный текст.

Это простое, но полезное приложение насчитывает приблизительно 200 строк кода (не считая модулей из стандартной библиотеки и прилагаемого к книге модуля `TkUtil`). Часто бывает, что простенькие утилиты с ГИП оказываются гораздо длиннее эквивалентных командных программ, но это различие быстро сходит на нет с увеличением сложности приложения.

7.2.2. Создание диалоговых окон в приложении

Диалоговые приложения просты и удобны для небольших утилит, медиаплееров и некоторых игр. Но более сложные приложения обычно имеют главное окно и вспомогательные диалоги. В этом подразделе мы увидим, как создается модальный и немодальный диалог.

С точки зрения виджетов, компоновки и привязок, между модальными и немодальными диалогами разницы нет. Но если модальный диалог обычно присваивает введенные пользователем значения переменным, то немодальный как правило вызывает методы приложения или изменяет данные приложения в ответ на действия пользователя. Кроме того, открытый модальный диалог блокирует приложение, тогда как немодальный позволяет и дальше работать с ним, и в своем коде мы должны учитывать не немаловажное различие.

7.2.2.1. Создание модального диалога

В этом подразделе мы рассмотрим диалог задания настроек в приложении Gravitate. Его код находится в файле `gravitate/Preferences.py`, а внешний вид показан на рис. 7.4.

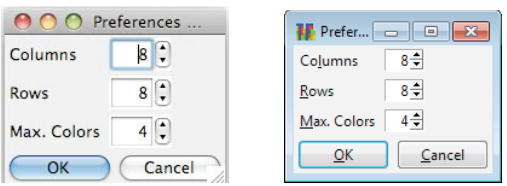

Рис. 7.4. Модальный диалог приложения Gravitate в OS X и в Windows

В Linux и Windows, когда пользователь выбирает из меню Gravitate пункт `File → Preferences`, вызывается метод `Main.Window.preferences()`, что приводит к показу модального диалога Preferences.

В OS X пользователь должен щелкнуть по пункту меню приложения `Preferences` или нажать ⌘ (к сожалению, мы должны обрабатывать оба случая сами, как показано в подразделе 7.3.2.1.)

```
def preferences(self):
    Preferences.Window(self, self.board)
    self.master.focus()
```

Это метод главного окна, который вызывает диалог Preferences. Этот диалог активный, поэтому вместо того чтобы просто передать ему значения, а затем, после нажатия кнопки `OK`, обновить состояние приложения, мы передаем ему сам объект приложения – в данном случае объект `self.board` типа `Board`, подкласса `tkinter.Canvas`, предназначенного для отображения двухмерной графики.

Метод создает новое диалоговое окно Preferences. Это окно отображается на экране и блокирует приложение (поскольку диалог модальный), пока пользователь не нажмет кнопку `OK` или `Cancel`. Никакой дополнительной работы от нас здесь не требуется, потому что диалог сам обновит `Board` после нажатия `OK`. После закрытия диалога мы должны только позаботиться о том, чтобы главное окно получило фокус клавиатуры.

В состав Tkinter входит модуль `tkinter.simpledialog`, который содержит два базовых класса для создания пользовательских диалогов и несколько вспомогательных функций для показа готовых диалогов, которые получают от пользователя одно значение, например, `tkinter.simpledialog.askfloat()`. В готовые диалоги встроены точки подключения, которые упрощают наследование им и добавление собственных виджетов. Однако на момент написания книги эти диалоги уже давно не обновлялись и не позволяют использовать виджеты с поддержкой тем. Поэтому к книге прилагается модуль `TkUtil/Dialog.py`, который содержит базовый класс для диалогов с поддержкой тем и работает по аналогии с `tkinter.simpledialog.Dialog`, а заодно предоставляет такие вспомогательные функции, как `TkUtil.Dialog.get_float()`.

Во всех диалогах в этой книге мы импортируем модуль `TkUtil`, а не `tkinter.simpledialog`, чтобы можно было воспользоваться виджетами с поддержкой тем, благодаря которым приложения Tkinter имеют платформенный внешний вид в OS X и Windows.

```
class Window(TkUtil.Dialog.Dialog):

    def __init__(self, master, board):
```

```
        self.board = board
        super().__init__(master, "Preferences \u2014 {}".format(APPNAME),
                TkUtil.Dialog.OK_BUTTON|TkUtil.Dialog.CANCEL_BUTTON)
```

Этот диалог принимает родителя (`master`) и экземпляр `Board`. С помощью этого экземпляра мы получаем начальные значения виджетов диалога, а когда пользователь нажимает `OK`, ему будут переданы новые значения виджетов, еще до того как диалог уничтожен. В константе `APPNAME` (не показана) хранится строка `"Gravitate"`.

Классы, наследующие `TkUtil.Dialog.Dialog`, должны предоставить метод `body()`, который создает все виджеты диалога, кроме кнопок, – кнопки создает базовый класс. Кроме того, подкласс должен реализовать метод `apply()`, вызываемый, когда пользователь принимает диалог (то есть нажимает `OK` или `Yes` в зависимости от того, какая кнопка была указана в качестве «принимающей»). Можно также реализовать методы `initialize()` и `validate()`, но в данном примере они не нужны.

```
def body(self, master):
    self.create_variables()
    self.create_widgets(master)
    self.create_layout()
    self.create_bindings()
    return self.frame, self.columnsSpinbox
```

Этот метод должен создать переменные диалога, разместить виджеты и привязать события к методам (за исключением кнопок и их привязок). Он должен вернуть либо виджет, содержащий все созданные нами виджеты (обычно фрейм), либо этот виджет и еще виджет, который должен первым получить фокус клавиатуры. В данном случае мы возвращаем фрейм, в который поместили все свои виджеты, и первый счетчик, получающий фокус.

```
def create_variables(self):
    self.columns = tk.StringVar()
    self.columns.set(self.board.columns)
    self.rows = tk.StringVar()
    self.rows.set(self.board.rows)
    self.maxColors = tk.StringVar()
    self.maxColors.set(self.board.maxColors)
```

Это очень простой диалог – в нем есть только метки и счетчики. Для каждого счетчика мы создаем переменную типа `tkinter.StringVar` и инициализируем ее соответствующим значением из переданного экземпляра `Board`. Может показаться более естественным

использовать переменные типа `tkinter.IntVar`, но поскольку Tcl/Tk в действительности работает только со строками, `StringVar` часто оказывается более удачным выбором.

```
def create_widgets(self, master):
   self.frame = ttk.Frame(master)
   self.columnsLabel = TkUtil.Label(self.frame, text="Columns",
          underline=2)
   self.columnsSpinbox = Spinbox(self.frame,
          textvariable=self.columns, from_=Board.MIN_COLUMNS,
          to=Board.MAX_COLUMNS, width=3, justify=tk.RIGHT,
          validate="all")
   self.columnsSpinbox.config(validatecommand=(
      self.columnsSpinbox.register(self.validate_int),
         "columnsSpinbox", "%P"))
   ...
```

Этот метод создает виджеты. Сначала создается внешний фрейм, который можно будет вернуть в качестве родителя, содержащего все остальные виджеты. Родителя фрейма назначает диалог, а сам фрейм (или его потомки) является родителем всех остальных создаваемых нами виджетов.

Мы показали только код создания виджетов, описывающих число столбцов. Код для задания числа строк и максимального количества цветов устроен точно так же. В каждом случае мы создаем метку и счетчик, а с каждым счетчиком ассоциируем переменную типа `StringVar`. Атрибут `width` – это ширина счетчика в символах. Кстати, чтобы не писать, к примеру, `underline=-1 if TkUtil.mac() else 0` при создании меток, мы воспользовались классом `TkUtil.Label`, а не `tkinter.ttk.Label`.

```
class Label(ttk.Label):

   def __init__(self, *args, **kwargs):
      super().__init__(*args, **kwargs)
      if mac():
         self.config(underline=-1)
```

(0, 0) **columnsLabel**	(0, 1) **columnsSpinbox**
(1, 0) **rowsLabel**	(1, 1) **rowsSpinbox**
(2, 0) **maxColorsLabel**	(2, 1) **maxColorsSpinbox**

Рис. 7.5. Компоновка тела диалогового окна Preferences в приложении Gravitate

Этот крохотный класс позволяет задать подчеркнутую букву, обозначающую клавишу-ускоритель, не беспокоясь о том, исполняется ли код в OS X, поскольку на этой платформе мы отменяем такую возможность, присвоив параметру `underline` значение –1. В модуле `TkUtil/__init__.py` есть также классы `Button`, `Checkbutton` и `Radiobutton`, и во всех них метод `__init__()` аналогичен показанному выше.

```
def validate_int(self, spinboxName, number):
   return TkUtil.validate_spinbox_int(getattr(self, spinboxName),
         number)
```

Выше (стр. 276) мы уже обсуждали, как осуществлять валидацию счетчиков, и функцию `TkUtil.validate_spinbox_float()`. Единственная разница между рассмотренным ранее методом `validate()` и этим методом `validate_int()` (если не считать имен и того факта, что здесь проверяется целое число) заключается в том, что в данном случае счетчик является параметром, а раньше мы проверяли конкретный счетчик.

При регистрации функции валидации было задано две строки: имя счетчика и форматная строка Tcl/Tk. Когда наступает время валидации, они передаются самой функции, и Tcl/Tk разбирает их. С именем счетчика Tcl/Tk ничего не делает, а в форматной строке заменяет `"%P"` строковым значением счетчика. Функция `TkUtil.validate_spinbox_int()` ожидает получить в качестве аргументов виджет счетчика и строковое значение. Поэтому мы пользуемся встроенной функцией `getattr()`, передавая ей диалог (`self`) и имя интересующего нас атрибута (`spinboxName`), а в ответ получаем ссылку на виджет счетчика.

```
def create_layout(self):
   padW = dict(sticky=tk.W, padx=PAD, pady=PAD)
   padWE = dict(sticky=(tk.W, tk.E), padx=PAD, pady=PAD)
   self.columnsLabel.grid(row=0, column=0, **padW)
   self.columnsSpinbox.grid(row=0, column=1, **padWE)
   self.rowsLabel.grid(row=1, column=0, **padW)
   self.rowsSpinbox.grid(row=1, column=1, **padWE)
   self.maxColorsLabel.grid(row=2, column=0, **padW)
   self.maxColorsSpinbox.grid(row=2, column=1, **padWE)
```

Этот метод размещает виджеты, как показано на рис. 7.5. Компоновка очень простая: это сетка, в которой все метки выровнены на левую границу (`sticky=tk.W`, запад), а все счетчики заполняют доступное пространство по горизонтали; при этом величина отступа

для всех виджетов составляет 0,75 мм (значение не показанной константы PAD). (См. врезку «Распаковка последовательностей и отображений» на стр. 24.)

```
   def create_bindings(self):
      if not TkUtil.mac():
         self.bind("<Alt-l>", lambda *args: self.columnsSpinbox.
focus())
         self.bind("<Alt-r>", lambda *args: self.rowsSpinbox.
focus())
         self.bind("<Alt-m>",
                   lambda *args: self.maxColorsSpinbox.focus())
```

На всех платформах, кроме OS X, мы хотим предоставить пользователю возможность переходить от одного счетчика к другому и нажимать на кнопки с помощью клавиш. Например, при нажатии Alt+R фокус получает счетчик для задания количества строк. Для кнопок ничего делать не надо, потому что обо всем позаботится базовый класс.

```
   def apply(self):
      columns = int(self.columns.get())
      rows = int(self.rows.get())
      maxColors = int(self.maxColors.get())
      newGame = (columns != self.board.columns or
                 rows != self.board.rows or
                 maxColors != self.board.maxColors)
      if newGame:
         self.board.columns = columns
         self.board.rows = rows
         self.board.maxColors = maxColors
         self.board.new_game()
```

Этот метод вызывается, только если пользователь нажимает кнопку «принятия» диалога (OK или Yes). Мы получаем значения переменных типа StringVar и преобразуем их в int (эта операция всегда должна завершаться успешно). Затем мы записываем результаты в соответствующие атрибуты экземпляра Board. Если хотя бы одно значение изменилось, то мы начинаем новую игру.

В больших сложных приложениях иногда необходимо выполнить много действий – выбор пунктов меню, вызов одних диалогов из других – чтобы добраться до нужного диалога. Чтобы упростить тестирование, часто бывает полезно добавить предложение if __name__== "__main__": в конец модуля, содержащего оконный класс, и в это предложение поместить код, который будет вызывать

диалог для тестирования. Вот код внутри такого предложения в модуле `gravitate/Preferences.py`.

```
def close(event):
   application.quit()
application = tk.Tk()
scoreText = tk.StringVar()
board = Board.Board(application, print, scoreText)
window = Window(application, board)
application.bind("<Escape>", close)
board.bind("<Escape>", close)
application.mainloop()
print(board.columns, board.rows, board.maxColors)
```

Сначала идет тривиальная функция, которая завершает приложение. Затем мы создаем обычно скрытый, но в данном случае видимый объект `tk.Tk`, который служит предком всех объектов приложения. Мы связываем нажатие клавиши `Esc` с функцией `close()`, чтобы пользователю было легко закрыть окно.

Экземпляр `Board` обычно передается диалогу вызывающим главным окном, но поскольку сейчас мы исполняем диалог автономно, то должны создать этот экземпляр самостоятельно.

Далее мы создаем диалог, который заблокирует выполнение программы, пока пользователь не нажмет `OK` или `Cancel`. Разумеется, этот диалог появляется, только когда начнется цикл обработки событий. А после его закрытия мы распечатываем атрибуты `Board`, которые диалог может изменять. Если была нажата кнопка `OK`, то мы должны увидеть изменения, которые произошли бы в главном окне; в противном случае все атрибуты должны остаться неизменными.

7.2.2.2. Создание немодальных диалогов

В этом подразделе мы рассмотрим немодальный диалог Help (Справка) в приложении Gravitate (рис. 7.6).

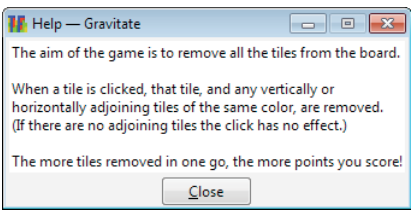

Рис. 7.6. Немодальный диалог Help в приложении Gravitate в Windows

Как уже отмечалось, с точки зрения виджетов, компоновки и привязки событий, между модальными и немодальными диалогами нет никакой разницы. Различие же состоит в том, что немодальный диалог не блокирует вызывающее окно (например, главное), которое продолжает выполнять цикл обработки событий и с которым пользователь может и дальше взаимодействовать. Сначала мы рассмотрим, как вызывается диалог, а затем код самого диалога.

```
def help(self, event=None):
   if self.helpDialog is None:
      self.helpDialog = Help.Window(self)
   else:
      self.helpDialog.deiconify()
```

Это метод `Main.Window.help()`. В объекте `Main.Window` хранится переменная экземпляра (`self.helpDialog`), которой в методе `__init__()` (не показан) присвоено значение `None`. При первом вызове диалога Help создается объект окна, и в качестве родителя ему передается главное окно. Само действие создания приводит к появлению диалога поверх главного окна, но поскольку диалог немодальный, цикл обработки событий в главном окне продолжается, и пользователь может взаимодействовать как с главным окном, так и с диалогом.

При втором и последующих вызовах диалога Help мы уже имеем ссылку на него, поэтому просто показываем на экране (с помощью метода `tkinter.Toplevel.deiconify()`). Это работает, потому что когда пользователь закрывает диалог, тот не уничтожает себя, а только скрывает. Создавать диалог только один раз куда быстрее, чем создавать и уничтожать после каждого использования. Кроме того, скрытый диалог сохраняет свое состояние.

```
class Window(tk.Toplevel):

   def __init__(self, master):
      super().__init__(master)
      self.withdraw()
      self.title("Help \u2014 {}".format(APPNAME))
      self.create_ui()
      self.reposition()
      self.resizable(False, False)
      self.deiconify()
      if self.winfo_viewable():
         self.transient(master)
      self.wait_visibility()
```

Немодальные диалоги обычно наследуют `tkinter.ttk.Frame` или `tkinter.Toplevel`, как показано выше. Диалог принимает родителя в аргументе `master`. Вызов `tkinter.Toplevel.withdraw()` сразу же скрывает окно (еще до того как пользователь его увидел), чтобы при создании не было видно мигания.

Затем мы устанавливаем заголовок окна «Help – Gravitate» и создаем виджеты диалога. Поскольку текст справки совсем короткий, мы запрещаем изменение размера диалога и оставляем Tkinter выбрать размер так, чтобы были видны только справка и кнопка `Close`. Если бы текст был длинный, то можно было бы взять подкласс `tkinter.Text` с полосами прокрутки и разрешить изменение размера окна.

После того как все виджеты созданы и размещены, мы вызываем метод `tkinter.Toplevel.deiconify()`, чтобы показать окно. Если окно просматриваемое (то есть показывается системным менеджером окон) – каковым оно и должно быть – то мы уведомляем Tkinter о том, что это окно недолговечное по отношению к своему родителю. Таким образом, оконная система узнает, что это окно, возможно, скоро исчезнет с экрана, и может оптимизировать объем перерисовки в случае скрытия или уничтожения окна.

Вызов метода `tkinter.Toplevel.wait_visibility()` в конце блокирует выполнение (на такое короткое время, что пользователь ничего не заметит), пока окно не станет видимым. По умолчанию окна верхнего уровня (`tkinter.Toplevel`) немодальные, но если добавить еще два предложения в конце, то окно можно сделать модальным. Это предложения `self.grab_set()` и `self.wait_window(self)`. Первое не дает фокусу («захвату» в терминологии Tk/Tcl) покинуть приложение и тем самым делает его модальным. Второе блокирует выполнение, пока окно не закроется. При обсуждении модальных диалогов мы не встречались с этим методами, потому что стандартный способ создания модального диалога – унаследовать его от `tkinter.simpledialog.Dialog` (или от `TkUtil.Dialog` в этой книге), а там эти предложения уже есть.

Теперь пользователь может взаимодействовать с этим окном, с главным окном приложения и с любым другим немодальным окном приложения, которое в данный момент открыто.

```
def create_ui(self):
    self.helpLabel = ttk.Label(self, text=_TEXT, background="white")
    self.closeButton = TkUtil.Button(self, text="Close", underline=0)
    self.helpLabel.pack(anchor=tk.N, expand=True, fill=tk.BOTH,
          padx=PAD, pady=PAD)
    self.closeButton.pack(anchor=tk.S)
```

```
        self.protocol("WM_DELETE_WINDOW", self.close)
        if not TkUtil.mac():
            self.bind("<Alt-c>", self.close)
        self.bind("<Escape>", self.close)
        self.bind("<Expose>", self.reposition)
```

Пользовательский интерфейс настолько прост, что мы создали его целиком в этом методе. Сначала создается метка для показа текста справки (он хранится в не показанной константе `_TEXT`), затем – кнопка `Close`. Мы воспользовались классом `TkUtil.Button` (производным от `tkinter.ttk.Button`), чтобы подчеркивание корректно игнорировалось в OS X. (Выше был показан подкласс `TkUtil.Label`, почти идентичный `TkUtil.Button`.)

Поскольку у нас всего два виджета, имеет смысл воспользоваться простейшим менеджером компоновки, поэтому здесь мы упаковываем метку сверху окна, разрешаем ей расти в обоих направлениях и упаковываем кнопку внизу.

Если текущая платформа – не OS X, то мы добавляем клавишу-ускоритель `Alt+C` для кнопки `Close`, и на любой платформе связываем нажатие `Esc` с закрытием окна.

Поскольку окно немодального диалога скрывается и показывается, а не уничтожается и создается заново, то возможна такая последовательность действий: пользователь показывает окно справки, затем закрывает (скрывает) его, затем перемещает главное окно и снова показывает окно справки. Было бы вполне разумно оставить окно справки там, где оно было в прошлый раз. Но поскольку текст справки такой короткий, пожалуй, лучше заново позиционировать диалог при каждом показе. Для этого мы привязываем событие `<Expose>` (которое происходит всякий раз, как окно должно перерисовать себя) к методу `reposition()`.

```
    def reposition(self, event=None):
        if self.master is not None:
            self.geometry("+{}+{}".format(self.master.winfo_rootx() + 50,
                self.master.winfo_rooty() + 50))
```

Этот метод располагает окно там же, где находится его родитель (то есть главное окно), но со смещением на 50 пикселей вправо и вниз.

Теоретически нет необходимости явно вызывать этот метод в `__init__()`, но, поступая так, мы гарантируем, что окно будет правильно позиционировано, перед тем как станет видимым. Тем самым удается избежать «перепрыгивания» окна в нужное место после показа.

```
def close(self, event=None):
    self.withdraw()
```

Этот метод вызывается, когда пользователь закрывает диалог – нажатием клавиши `Esc` или `Alt+C`, кнопки `Close` или ×. Он просто скрывает окно, не уничтожая его. Впоследствии окно можно показать снова, вызвав для него метод `tkinter.Toplevel.deiconify()`.

7.3. Создание приложений с главным окном с помощью Tkinter

В этом разделе мы узнаем о наиболее общих аспектах приложения с главным окном Gravitate. Внешний вид приложения показан на рис. 7.7, и правила игры описаны на врезке «Игра Gravitate». В пользовательском интерфейсе имеются стандартные ожидаемые элементы: полоса меню, центральный виджет, строка состояния и диалоги. Tkinter поддерживает меню изначально, но центральный виджет и строку состояния придется создавать самостоятельно. Несложно будет адаптировать код Gravitate к другим приложениям с главным окном – нужно лишь изменить виджет в центре, меню и строку состояния, оставив общую инфраструктуру неизменной.

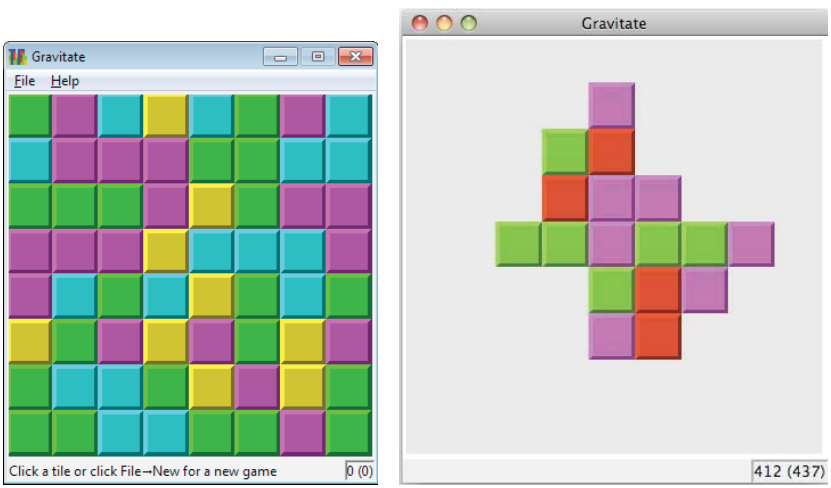

Рис. 7.7. Приложение Gravitate в Windows и OS X

Программа Gravitate состоит из семи Python-файлов и девяти изображений. «Исполняемым» файлом приложения является `gravitate/gravitate.pyw`, а код главного окна находится в файле `gravitate/Main.py`. Имеется три вспомогательных диалога: `gravitate/About.py`, который мы рассматривать не будем, `gravitate/Help.py`, рассмотренный в предыдущем разделе, и `gravitate/Preferences.py`, который будет рассмотрен в подразделе 7.2.2.1. В центральной области главного окна находится виджет `Board` из файла `gravitate/Board.py` – подкласс `tkinter.Canvas` (рассмотреть его целиком нам не хватит места).

Игра Gravitate

Цель игры – убрать с доски все плитки. После щелчка по плитке она сама, а также примыкающие к ней по вертикали и горизонтали плитки того же цвета убираются (если к плитке ничего не примыкает, то щелчок ни к чему не приводит). Чем больше плиток убирается за один ход, тем больше очков зарабатывает игрок.

По своей логике Gravitate похожа на игры Tile Fall или Same Game. Основное отличие между ними и Gravitate заключается в том, что при убирании плиток в Tile Fall или Same Game клетки падают и сдвигаются влево, заполняя пустые места, тогда как в Gravitate плитки «притягиваются» к центру доски.

В примерах к этой книге есть три варианта приложения Gravitate. Первый находится в каталоге `gravitate` и описан в этом разделе. Второй – в каталоге `gravitate2`, логика у него такая же, как у первого варианта, но есть дополнительные возможности: скрываемая панель инструментов и диалог Preferences с большим числом настроек, позволяющий, например, выбрать форму плиток и масштабный коэффициент для показа плиток большего или меньшего размера. Кроме того, `gravitate2` запоминает счет между сеансами и позволяет играть с помощью как мыши, так и клавиш (для перемещения используются клавиши со стрелками, а для убирания – пробел). Третий вариант трехмерный, он будет рассмотрен в главе 8. Имеется также сетевая версия на странице `www.qtrac.eu/gravitate.html`.

```
def main():
    application = tk.Tk()
    application.withdraw()
    application.title(APPNAME)
    application.option_add("*tearOff", False)
    TkUtil.set_application_icons(application, os.path.join(
            os.path.dirname(os.path.realpath(__file__)), "images"))
    window = Main.Window(application)
    application.protocol("WM_DELETE_WINDOW", window.close)
    application.deiconify()
    application.mainloop()
```

Это функция `main()` из файла `gravitate/gravitate.pyw`. Она создает объект верхнего уровня `tkinter.Tk` и сразу же скрывает приложение, чтобы не было миганий во время создания главного окна. По умолчанию, меню в Tkinter отрывные (в память древней библиотеки графического интерфейса Motif); мы отключаем эту функцию, поскольку ни в одном современном ГИП она не используется. Затем мы устанавливаем значки приложения, пользуясь уже обсуждавшейся функцией (стр. 274). Далее мы создаем главное окно приложения и сообщаем Tkinter, что при нажатии на кнопку закрытия × следует вызывать метод `Main.Window.close()`. Наконец, мы показываем приложение (точнее, помещаем в очередь событие показа) и запускаем цикл обработки событий. В этот момент окно приложения появится на экране.

7.3.1. Создание главного окна

В Tkinter главные окна, по существу, не отличаются от диалоговых. Однако на практике главное окно обычно имеет полосу меню и строку состояния, часто – панели инструментов, а иногда – стыкуемые окна. Как правило, имеется один центральный виджет, занятый, например, текстовым редактором, электронной таблицей или графикой (в случае игры, моделирования или визуализации). В приложении Gravitate есть полоса меню, центральный графический виджет и строка состояния.

```
class Window(ttk.Frame):

    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.create_variables()
        self.create_images()
        self.create_ui()
```

Класс `Main.Window` наследует `tkinter.ttk.Frame` и большую часть своей работы делегирует базовому классу и трем вспомогательным методам.

```
    def create_variables(self):
        self.images = {}
        self.statusText = tk.StringVar()
        self.scoreText = tk.StringVar()
        self.helpDialog = None
```

Ненадолго появляющиеся в строке состояния сообщения хранятся в атрибуте `self.statusText`, а постоянно отображаемый счет (как и высшее достижение) – в атрибуте `self.scoreText`. Ссылка на диало-

говое окно справки сначала равна None; это окно обсуждалось выше в подразделе 7.2.2.2.

В приложениях с ГИП часто слева от текста пункта меню отображается значок, а для кнопок на панели инструментов значок вообще обязателен. В игре Gravitate мы поместили все значки в каталог gravitate/images и сопоставили им константы (например, константа NEW содержит строку "New"). При создании объекта Main.Window вызывается метод create_images(), который загружает все необходимые изображения как значения словаря self.images. Важно хранить ссылки на загруженные Tkinter изображения, иначе они будут убраны в мусор (и исчезнут с экрана).

```
def create_images(self):
    imagePath = os.path.join(os.path.dirname(
            os.path.realpath(__file__)), "images")
    for name in (NEW, CLOSE, PREFERENCES, HELP, ABOUT):
        self.images[name] = tk.PhotoImage(
                file=os.path.join(imagePath, name + "_16x16.gif"))
```

Мы решили использовать в меню изображения размером 16 × 16, их мы и загружаем для каждой константы, обозначающей действие (NEW, CLOSE и т. д.).

Во встроенной константе __file__ хранится имя файла, включающее путь. Для получения абсолютного пути мы используем функцию os.path.realpath(), которая исключает компоненты «..» и символические ссылки, а затем вырезаем из полного имени каталог (то есть опускаем имя файла) и добавляем к нему часть "images", чтобы получить путь к подкаталогу images приложения.

```
def create_ui(self):
    self.create_board()
    self.create_menubar()
    self.create_statusbar()
    self.create_bindings()
    self.master.resizable(False, False)
```

Благодаря нашему бескомпромиссному пониманию рефакторинга этот метод перепоручает всю свою работу вспомогательным методам. А подготовив пользовательский интерфейс, метод запрещает изменять размер окна. Действительно, какой смысл изменять размер окна, если все плитки имеют фиксированный размер? (В приложении Gravitate 2 пользователю тоже запрещается изменять размер окна непосредственно, но он может изменить размер плиток, и тогда размер окна подстроится автоматически.)

```
def create_board(self):
    self.board = Board.Board(self.master, self.set_status_text,
            self.scoreText)
    self.board.update_score()
    self.board.pack(fill=tk.BOTH, expand=True)
```

Этот метод создает экземпляр `Board` (подкласса `tkinter.Canvas`), передавая конструктору метод `self.set_status_text()`, чтобы объект мог выводить сообщения в строку состояния главного окна, а также `self.scoreText`, чтобы можно было обновлять счет (и высшее достижение).

Создав доску, мы вызываем ее метод `update_score()`, который выведет текст «0 (0)» в области постоянно присутствующего индикатора счета. Мы также упаковываем доску в главном окне, разрешая ей раздвигаться в обоих направлениях.

```
def create_bindings(self):
    modifier = TkUtil.key_modifier()
    self.master.bind("<{}-n>".format(modifier), self.board.new_game)
    self.master.bind("<{}-q>".format(modifier), self.close)
    self.master.bind("<F1>", self.help)
```

Здесь мы определяем три сочетания клавиш: `Ctrl+N` (или `⌘N`) – начать новую игру, `Ctrl+Q` (или `⌘Q`) – выйти из программы и `F1` – открыть (или показать, если окно скрыто) немодальное окно справки. Метод `TkUtil.key_modifier()` возвращает зависящее от платформы название клавиши-модификатора (`"Control"` или `"Command"`).

7.3.2. Создание меню

В системах Linux и Windows принято располагать меню под полосой заголовка, и Tkinter следует этой традиции. Но в OS X меню объединяется с единственным системным меню в верхней части экрана. Однако, как мы увидим ниже, необходимо помочь Tkinter осуществить такое объединение.

Меню и подменю – экземпляры класса `tkinter.Menu`. Одно меню должно быть создано в окне верхнего уровня (например, в полосе меню главного окна), а все остальные являются его потомками.

```
def create_menubar(self):
    self.menubar = tk.Menu(self.master)
    self.master.config(menu=self.menubar)
    self.create_file_menu()
    self.create_help_menu()
```

Здесь мы создаем пустое меню как потомка главного окна и записываем ссылку на него (`self.menubar`) в атрибут окна `menu` (определяющий полосу меню). Затем мы добавляем подменю – в данном случае их всего два, и мы рассмотрим их в следующих подразделах.

7.3.2.1. Создание меню File

У большинства приложений с главным окном есть меню File, содержащее команды создания нового документа, открытия существующего документа, сохранения текущего документа и завершения приложения. Но для игр большая часть этих команд ни к чему, поэтому в программе Gravitate меню File содержит всего два пункта, показанных на рис. 7.8.

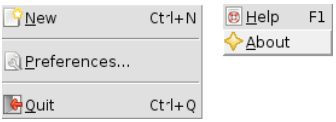

Рис. 7.8. Меню приложения Gravitate в Linux

```
def create_file_menu(self):
    modifier = TkUtil.menu_modifier()
    fileMenu = tk.Menu(self.menubar, name="apple")
    fileMenu.add_command(label=NEW, underline=0,
            command=self.board.new_game, compound=tk.LEFT,
            image=self.images[NEW], accelerator=modifier + "+N")
    if TkUtil.mac():
        self.master.createcommand("exit", self.close)
        self.master.createcommand("::tk::mac::ShowPreferences",
            self.preferences)
    else:
        fileMenu.add_separator()
        fileMenu.add_command(label=PREFERENCES + ELLIPSIS, underline=0,
            command=self.preferences,
            image=self.images[PREFERENCES], compound=tk.LEFT)
        fileMenu.add_separator()
        fileMenu.add_command(label="Quit", underline=0,
            command=self.close, compound=tk.LEFT,
            image=self.images[CLOSE],
            accelerator=modifier + "+Q")
    self.menubar.add_cascade(label="File", underline=0,
        menu=fileMenu)
```

Этот метод создает меню File. Имена констант записываются большими буквами, и, если не оговорено противное, константы содержат одноименные строки; например, в константе `NEW` хранится строка `"New"`.

Сначала этот метод получает модификатор клавиш-ускорителей (⌘ в OS X, Ctrl в Linux и Windows). Затем он создает меню File как потомка полосы меню в окне. Выбранное имя меню ("apple") говорит Tkinter, что в OS X это меню должно быть объединено с меню приложения; на других платформах имя меню игнорируется.

Пункты добавляются в меню методами tkinter.Menu.add_command(), tkinter.Menu.add_checkbutton() и tkinter.Menu.add_radiobutton(), хотя в приложении Gravitate используется только первый. Разделители добавляются методом tkinter.add_separator(). Атрибут underline в OS X и в Windows игнорируется, разделители становятся видны, только если это явно указано в настройках или если нажать клавишу Alt. Для каждого пункта меню задается текст метки, положение подчеркнутого символа, команда, выполняемая при выборе этого пункта, и значок (атрибут image). Атрибут compound говорит, как обрабатывать значки и текст: tk.LEFT означает, что нужно показывать то и другое, расположив значок слева. Мы также задаем клавишу-ускоритель; например, чтобы выбрать пункт File → New, пользователь может нажать Ctrl+N в Linux и Windows или ⌘N в OS X.

В OS X меню Preferences и Quit текущего приложения располагаются в меню приложений (справа от меню Apple находится меню File приложения). Чтобы интегрироваться с OS X, мы используем метод tkinter.Tk.createcommand(), который ассоциирует команды Tcl/Tk ::tk::mac::ShowPreferences и exit с соответствующими методами Gravitate. На других платформах пункты Preferences и Quit добавляются как обычные пункты меню.

Заполнив меню File, мы добавляем его как каскадное меню (то есть подменю) в полосу меню.

```
def menu_modifier():
    return "Command" if mac() else "Ctrl"
```

Эта крохотная функция, находящаяся в файле TkUtil/__init__.py, используется для задания текста пункта меню. В OS X слово "Command" обрабатывается специальным образом и преобразуется в символ ⌘ на экране.

7.3.2.2. Создание меню Help

Меню Help в нашем приложении содержит всего два пункта: Help и About. Однако в OS X оба они обрабатываются иначе, чем в Linux и Windows, поэтому наш код должен учитывать различия.

```
def create_help_menu(self):
    helpMenu = tk.Menu(self.menubar, name="help")
    if TkUtil.mac():
        self.master.createcommand("tkAboutDialog", self.about)
        self.master.createcommand("::tk::mac::ShowHelp", self.help)
    else:
        helpMenu.add_command(label=HELP, underline=0,
                command=self.help, image=self.images[HELP],
                compound=tk.LEFT, accelerator="F1")
        helpMenu.add_command(label=ABOUT, underline=0,
                command=self.about, image=self.images[ABOUT],
                compound=tk.LEFT)
    self.menubar.add_cascade(label=HELP, underline=0,
            menu=helpMenu)
```

Сначала мы создаем меню справки с именем `"help"`. В Linux и Windows имя игнорируется, а в OS X меню объединяется с системным меню справки. В OS X мы вызываем метод `tkinter.Tk.createcommand()`, чтобы ассоциировать команды Tcl/Tk `tkAboutDialog` и `::tk::mac::ShowHelp` с соответствующими методами приложения Gravitate. На других платформах пункты `Help` и `About` создаются обычным образом.

7.3.3. Создание строки состояния с индикаторами

В приложении Gravitate имеется типичная строка состояния, в которой слева отображаются временные текстовые сообщения, а справа находится постоянный индикатор состояния. На рис. 7.7 показан как индикатор, так и временное сообщение (слева).

```
def create_statusbar(self):
    statusBar = ttk.Frame(self.master)
    statusLabel = ttk.Label(statusBar, textvariable=self.statusText)
    statusLabel.grid(column=0, row=0, sticky=(tk.W, tk.E))
    scoreLabel = ttk.Label(statusBar, textvariable=self.scoreText,
            relief=tk.SUNKEN)
    scoreLabel.grid(column=1, row=0)
    statusBar.columnconfigure(0, weight=1)
    statusBar.pack(side=tk.BOTTOM, fill=tk.X)
    self.set_status_text("Click a tile or click File→New for a new "
            "game")
```

Создание строки состояния мы начинаем с создания фрейма. Затем добавляем метку и ассоциируем ее с переменной `self.statusText` (типа `StringVar`). Теперь для показа текста сообщения нам доста-

точно присвоить значение переменной `self.statusText`, хотя на практике мы будем вызывать метод. Мы также добавляем один постоянный индикатор состояния – метку, в которой отображается счет (и высшее достижение), – и ассоциируем его с переменной `self.scoreText`.

Обе метки размещаются в ячейках сетки внутри фрейма, при этом `statusLabel` (метка, в которой показываются временные сообщения) занимает все доступное место по горизонтали. Сам фрейм строки состояния упаковывается в нижней части главного окна и растягивается по горизонтали на всю ширину окна. В самом конце мы показываем начальное временное сообщение с помощью метода `set_status_text()`.

```
def set_status_text(self, text):
    self.statusText.set(text)
    self.master.after(SHOW_TIME, lambda: self.statusText.set(""))
```

Этот метод записывает в переменную `self.statusText` указанный текст (возможно, пустой) и через `SHOW_TIME` миллисекунд (5 секунд в данном примере) стирает его.

Мы поместили в строку состояния только метки, но можно было бы добавить другие виджеты, например, комбинированные списки, счетчики или кнопки.

В этой главе у нас хватило места, только чтобы показать некоторые базовые приемы работы с Tkinter. С тех пор как в Python включена версия Tcl/Tk 8.5 (первая с поддержкой тем), Tkinter стала куда более привлекательным решением, поскольку теперь приложения выглядят естественно в OS X и в Windows. Tkinter содержит несколько очень мощных и гибких виджетов и, прежде всего, `tkinter.Text`, который позволяет редактировать и показывать стилизованный и отформатированный текст, а также `tkinter.Canvas` для работы с двухмерной графикой (используется в программах Gravitate и Gravitate 2). Очень полезны еще три виджета: `tkinter.ttk.Treeview` для представления таблиц или деревьев, `tkinter.ttk.Notebook` для представления вкладок (используется в диалоговом окне настроек в программе Gravitate 2) и `tkinter.ttk.Panedwindow`, позволяющий создать расщепленное окно.

Хотя Tkinter не содержит некоторых высокоуровневых функций, имеющихся в других библиотеках ГИП, мы видели, как легко само-

стоятельно создать строку состояния с областью временных сообщений и постоянным индикатором состояния. Система меню в Tkinter позволяет намного больше, чем нам было нужно в этой главе; она поддерживает подменю произвольного уровня, а также пункты, допускающие отметку (флажки и переключатели). Создать контекстное меню тоже несложно.

Из современных элементов больше всего недостает панелей инструментов. Создать их довольно просто, хотя нужно немного повозиться, чтобы сделать их скрываемыми и обеспечить автоматическую компоновку при изменении размера окна. Еще один современный механизм – стыкуемые окна. Ничто не мешает создать такое окно, которое можно будет скрывать и показывать, перетаскивать из одной области стыковки в другую и даже делать плавающим.

В примерах к книге есть два не рассмотренных в тексте этой главы приложения: `texteditor` и `texteditor2`, последнее показано на рис. 7.9. В обоих демонстрируется, как реализовать скрываемые панели инструментов, подменю, пункты меню с флажками и переключателями, контекстные меню и список недавно открывавшихся файлов. В обоих имеется диалоговое окно с дополнительной частью (рис. 7.10). Демонстрируется также виджет `tkinter.Text` и работа с буфером обмена. Кроме того, в программе `texteditor2` показано, как можно реализовать стыкуемые окна (хотя плавающий режим в OS X работает некорректно).

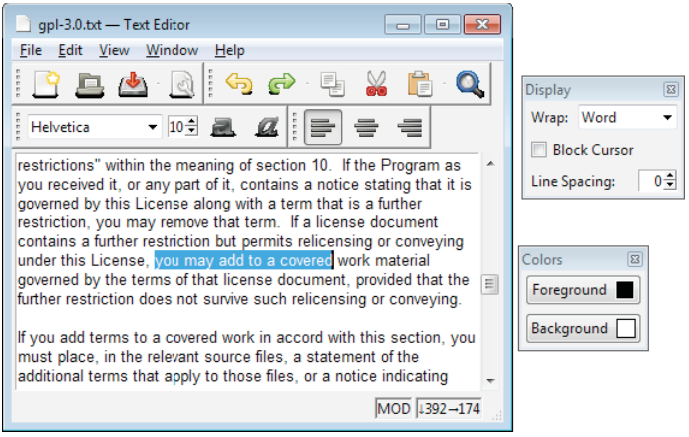

Рис. 7.9. Приложение Text Editor 2 в Windows

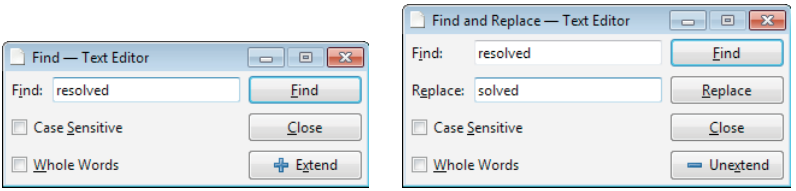

Рис. 7.10. Диалог с дополнительной частью в приложение Text Editor в Windows

Очевидно, что при работе с Tkinter для организации базовой инфраструктуры придется приложить больше усилий, чем в других библиотеках ГИП. Но зато Tkinter накладывает не так уж много ограничений и, если аккуратно написать необходимую инфраструктуру (например, панели инструментов и стыкуемые окна), то ее можно будет повторно использовать во всех приложениях. Tkinter очень надежна и поставляется вместе с Python, так что она прекрасно подходит для разработки простых для развертывания приложений с ГИП.

ГЛАВА 8. Трехмерная графика на Python с применением OpenGL

Во многих современных приложениях – средствах конструирования, инструментах визуализации данных, играх и, конечно же, хранителях экрана – используется трехмерная графика. Все упомянутые в предыдущей главе библиотеки ГИП поддерживают трехмерную графику – непосредственно или с помощью надстроек. Почти всегда ее поддержка выступает в виде интерфейса к системным библиотекам OpenGL.

Существует также немало пакетов трехмерной графики на Python, которые предоставляют высокоуровневые интерфейсы, упрощающие программирование OpenGL. Назовем лишь несколько примеров: Python Computer Graphics Kit (`cgkit.sourceforge.net`), OpenCASCADE (`github.com/tenko/occmodel`) и VPython (`www.vpython.org`).

К OpenGL можно обращаться и непосредственно. Из пакетов, предлагающих такую функциональность, наиболее известны PyOpenGL (`pyopengl.sourceforge.net`) и pyglet (`www.pyglet.org`). Оба точно обертывают библиотеки OpenGL, что позволяет безо всякого труда транслировать на Python написанные на C программы (C – язык, на котором написаны библиотеки OpenGL, он используется во всех учебниках по OpenGL). Оба пакета годятся для написания автономных программ с трехмерной графикой; в случае PyOpenGL используется обертка библиотеки GLUT, а pyglet имеет собственные средства обработки событий и поддержки окон верхнего уровня.

При разработке автономных 3D-программ, пожалуй, лучше использовать в сочетании с PyOpenGL какую-нибудь существующую библиотеку ГИП (годится Tkinter, PyQt, PySide, wxPython и многие

другие), а если достаточно более простого интерфейса, то подойдет pyglet.

Существует много версий OpenGL и два совершенно различных способа работы с этой библиотекой. Традиционный подход (так называемый «непосредственный режим») работает на всех платформах и для всех версий, в этом случае вызываются функции OpenGL, которые исполняются непосредственно. Более современный подход, доступный, начиная с версии 2.0, заключается в том, чтобы подготовить сцену с помощью традиционных средств, а затем написать OpenGL-программу на шейдерном языке OpenGL (специализированный вариант языка C). Такая программа затем передается графическому процессору в виде простого текста, а тот компилирует и выполняет ее. Программы при этом получаются гораздо более быстрыми и гибкими, но этот подход пока не слишком широко поддержан.

В этой главе мы рассмотрим одну программу на PyOpenGL и две на pyglet. В совокупности они иллюстрируют различные основополагающие концепции программирования трехмерной графики на OpenGL. Мы будем везде использовать традиционный подход, поскольку 3D-графику гораздо проще понять, рассматривая вызовы функций, чем изучая шейдерный язык, да и в любом случае нас больше интересует программирование на Python. В этой главе предполагается знакомство с программированием OpenGL, поэтому большая часть встречающихся вызовов OpenGL не поясняется. Читателям, не знакомым с OpenGL, рекомендуется обратиться к книге «OpenGL SuperBible», упомянутой в краткой библиографии.

Важно отметить соглашение об именовании в OpenGL. Имена многих функций OpenGL заканчиваются числом, за которым следует одна или несколько букв. Число – это количество аргументов, а буквами обозначаются их типы. Например, функция `glColor3f()` задает текущий цвет и ожидает получить три аргумента с плавающей точкой – красную, зеленую и синюю компоненты в диапазоне от 0,0 до 1,0. С другой стороны, функция `glColor4ub()` задает цвет, определяемый четырьмя байтами без знака – красная, зеленая, синяя и альфа (прозрачность) компоненты в диапазоне от 0 до 255. Естественно, в Python можно использовать числа любого типа, полагаясь на автоматическое преобразование.

Для представления трехмерной сцены на двухмерной плоскости (например, на экране монитора) обычно применяется один из двух способов: ортографическая или перспективная проекция. Ортографическое проецирование сохраняет размеры объектов и чаще ис-

пользуется в программах автоматизированного проектирования. В перспективной проекции более близкие к зрителю объекты кажутся крупнее, а отдаленные – мельче. В результате получается более реалистичный эффект, особенно при изображении ландшафтов. В играх применяются обе проекции. В первом разделе этой главы мы создадим сцену в перспективной проекции, а во втором – в ортографической.

8.1. Сцена в перспективной проекции

В этом разделе мы напишем программы Cylinder, результаты их работы показаны на рис. 8.1. Обе программы рисуют три раскрашенных оси и освещаемый полый цилиндр. Версия на базе PyOpenGL (слева) ближе всего к интерфейсам OpenGL, тогда как версия на базе pyglet (справа), пожалуй, немного проще для программирования и чуть более эффективна.

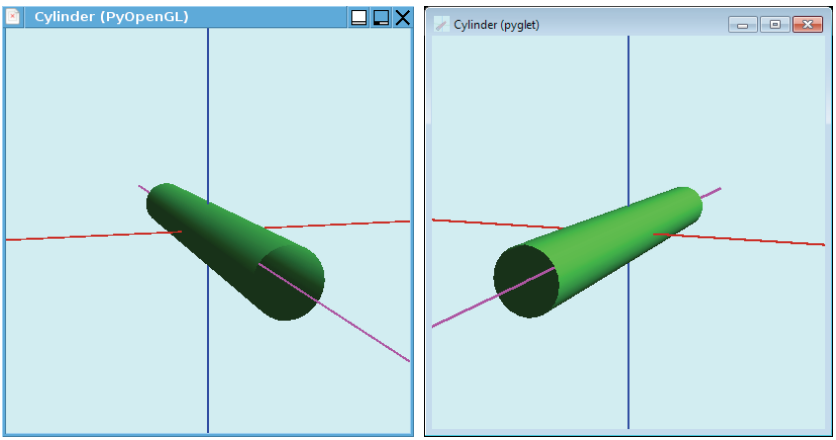

Рис. 8.1. Программы Cylinder в Linux и Windows

Код в обеих программах по большей части одинаковый, а некоторые методы отличаются только по именам. Поэтому в первом подразделе мы рассмотрим программу на базе PyOpenGL целиком, а во втором – только отличия, имеющиеся в программе на базе pyglet. Впоследствии (раздел 8.2) мы познакомимся с pyglet гораздо ближе.

8.1.1. Создание программы Cylinder с помощью PyOpenGL

Программа `cylinder1.pyw` создает простую сцену, которую пользователь может вращать относительно осей *x* и *y*. А при изменении размера объемлющего окна сцена автоматически масштабируется.

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
```

В программе используются следующие библиотеки OpenGL: `GL` (ядро), `GLU` (служебные функции) и `GLUT` (оконная библиотека). Обычно использовать синтаксис `from module import *` не рекомендуется, но в случае PyOpenGL это разумно, потому что все импортируемые имена начинаются префиксами `gl`, `glu`, `glut` или `GL`, так что отличить их легко, а конфликты маловероятны.

```
SIZE = 400
ANGLE_INCREMENT = 5

def main():
   glutInit(sys.argv)
   glutInitWindowSize(SIZE, SIZE)
   window = glutCreateWindow(b"Cylinder (PyOpenGL)")
   glutInitDisplayString(b"double=1 rgb=1 samples=4 depth=16")
   scene = Scene(window)
   glutDisplayFunc(scene.display)
   glutReshapeFunc(scene.reshape)
   glutKeyboardFunc(scene.keyboard)
   glutSpecialFunc(scene.special)
   glutMainLoop()
```

Библиотека `GLUT` предоставляет средства обработки событий и создания окон верхнего уровня, обычные для любой библиотеки ГИП. Чтобы воспользоваться ей, мы должны в самом начале вызвать функцию `glutInit()`, передав ей аргументы, заданные в командной строке; она применит и уберет те аргументы, которые понимает. После этого можно задать начальный размер окна (что мы и сделали). Затем мы создаем окно и устанавливаем его заголовок. Функция `glutInitDisplayString()` определяет некоторые параметры контекста OpenGL, в данном случае включает двойную буферизацию, использование цветовой модели RGBA (красный, зеленый, синий, альфа), поддержку сглаживания и устанавливает глубину буфера 16 бит (полный список параметров с описанием их назначения см. в документации по PyOpenGL).

В интерфейсах OpenGL используются 8-разрядные строки (обычно в кодировке ASCII). Чтобы передать такую строку, можно было бы вызвать метод `str.encode()`, который возвращает массив байтов в указанной кодировке (например, `"title".encode("ascii")` вернет `b'title'`), но мы воспользовались байтовыми литералами непосредственно.

Класс `Scene` мы будем использовать для отрисовки графики OpenGL в окне. После того как объект сцены создан, мы регистрируем некоторые его методы как обратные вызовы `GLUT`, то есть функции, которые OpenGL будем вызывать в ответ на определенные события. Регистрируется метод `Scene.display()`, который вызывается при показе окна (в первый раз и после того, как прежде скрытая часть становится видна); метод `Scene.reshape()`, который вызывается при изменении размера окна; метод `Scene.keyboard()`, который вызывается при нажатии клавиши (кроме некоторых), и метод `Scene.special()`, который вызывается, когда пользователь нажимает клавишу, не обработанную зарегистрированной функцией `keyboard()`.

Создав окно и зарегистрировав функции обратного вызова, мы запускаем цикл обработки событий `GLUT`. Он будет работать, пока программа не завершится.

```
class Scene:

   def __init__(self, window):
      self.window = window
      self.xAngle = 0
      self.yAngle = 0
      self._initialize_gl()
```

Сначала мы сохраняем ссылку на окно OpenGL и задаем нулевые углы осей *x* и *y*. Инициализация, специфичная для OpenGL, вынесена в отдельную функцию, которую мы вызываем в конце метода `__init__`.

```
   def _initialize_gl(self):
      glClearColor(195/255, 248/255, 248/255, 1)
      glEnable(GL_DEPTH_TEST)
      glEnable(GL_POINT_SMOOTH)
      glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
      glEnable(GL_LINE_SMOOTH)
      glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)
      glEnable(GL_COLOR_MATERIAL)
      glEnable(GL_LIGHTING)
      glEnable(GL_LIGHT0)
```

```
    glLightfv(GL_LIGHT0, GL_POSITION, vector(0.5, 0.5, 1, 0))
    glLightfv(GL_LIGHT0, GL_SPECULAR, vector(0.5, 0.5, 1, 1))
    glLightfv(GL_LIGHT0, GL_DIFFUSE, vector(1, 1, 1, 1))
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50)
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, vector(1, 1, 1, 1))
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
```

Этот метод вызывается только один раз для настройки контекста OpenGL. Сначала мы задаем цвет фона – светло-голубой. Затем включаем различные возможности OpenGL, самая важная из которых – создание источника света. Именно благодаря ему цилиндр окрашен неравномерно. Мы также делаем так, что основной (без освещения) цвет цилиндра будет зависеть от обращений к функциям `glColor…()`; например, при включенном режиме `GL_COLOR_MATERIAL` установка красного текущего цвета – `glColor3ub(255, 0, 0)` – отразится также на цвете материала (в данном случае это цвет цилиндра).

```
def vector(*args):
    return (GLfloat * len(args))(*args)
```

Эта вспомогательная функция создает массив OpenGL значений с плавающей точкой (каждое имеет тип `GLfloat`).

```
def display(self):
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    glPushMatrix()
    glTranslatef(0, 0, -600)
    glRotatef(self.xAngle, 1, 0, 0)
    glRotatef(self.yAngle, 0, 1, 0)
    self._draw_axes()
    self._draw_cylinder()
    glPopMatrix()
```

Этот метод вызывается при первом показе окна сцены и после того, как становится видна ранее скрытая часть окна (например, если перекрывающее окно закрывается или сдвигается в сторону). Он отодвигает сцену назад (вдоль оси z), так что мы смотрим на нее спереди, и поворачивает ее вокруг осей x и y при взаимодействии с пользователем (первоначально оба угла поворота равны нулю). После того как сцена подвергнута параллельному переносу и повороту, мы рисуем оси и сам цилиндр.

```
def _draw_axes(self):
    glBegin(GL_LINES)
    glColor3f(1, 0, 0) # x-axis
```

```
    glVertex3f(-1000, 0, 0)
    glVertex3f(1000, 0, 0)
    glColor3f(0, 0, 1) # y-axis
    glVertex3f(0, -1000, 0)
    glVertex3f(0, 1000, 0)
    glColor3f(1, 0, 1) # z-axis
    glVertex3f(0, 0, -1000)
    glVertex3f(0, 0, 1000)
    glEnd()
```

Вершиной (vertex) в OpenGL называется точка в трехмерном пространстве. Все оси рисуются одинаково: задаем цвет оси и начальную и конечную вершину. Функции `glColor3f()` и `glVertex3f()` принимают по три аргумента с плавающей точкой, но мы задали целые числа, предоставив Python позаботиться о преобразовании.

```
def _draw_cylinder(self):
    glPushMatrix()
    try:
        glTranslatef(0, 0, -200)
        cylinder = gluNewQuadric()
        gluQuadricNormals(cylinder, GLU_SMOOTH)
        glColor3ub(48, 200, 48)
        gluCylinder(cylinder, 25, 25, 400, 24, 24)
    finally:
        gluDeleteQuadric(cylinder)
    glPopMatrix()
```

Библиотека служебных функций `GLU` поддерживает несколько простых трехмерных тел, включая и цилиндры. Сначала мы отодвигаем начальную точку назад по оси *z*. Затем создаем «квадрику» – объект, с помощью которого можно рисовать различные тела. Мы задаем цвет с помощью трех байтов без знака (красная, зеленая и синяя компонента в диапазоне от 0 до 255). Функция `gluCylinder()` принимает квадрику общего вида, радиусы цилиндра на каждом конце (в данном случае одинаковые), высоту цилиндра и два коэффициента частоты разбиения (чем они больше, тем более гладким получается результат и тем больше времени занимает его построение). В конце мы явно удаляем квадрику, не полагаясь на сборщик мусора Python, чтобы минимизировать потребление ресурсов.

```
def reshape(self, width, height):
    width = width if width else 1
    height = height if height else 1
    aspectRatio = width / height
    glViewport(0, 0, width, height)
```

```
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(35.0, aspectRatio, 1.0, 1000.0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

Этот метод вызывается при изменении размера окна сцены. Почти вся работа перепоручается функции `gluPerspective()`. На самом деле, показанный здесь код можно считать отправной точкой для любой сцены в перспективной проекции.

```
def keyboard(self, key, x, y):
   if key == b"\x1B": # Escape
      glutDestroyWindow(self.window)
```

Этот метод (зарегистрированный с помощью функции `glutKeyboardFunc()`) вызывается, когда пользователь нажимает какую-нибудь клавишу (кроме функциональных, стрелок, `Page Up`, `Page Down`, `Home`, `End` и `Insert`). Здесь мы смотрим, нажата ли клавиша `Esc`, и, если да, то удаляем окно. А поскольку больше никаких окон нет, то это действие завершает программу.

```
def special(self, key, x, y):
   if key == GLUT_KEY_UP:
      self.xAngle -= ANGLE_INCREMENT
   elif key == GLUT_KEY_DOWN:
      self.xAngle += ANGLE_INCREMENT
   elif key == GLUT_KEY_LEFT:
      self.yAngle -= ANGLE_INCREMENT
   elif key == GLUT_KEY_RIGHT:
      self.yAngle += ANGLE_INCREMENT
   glutPostRedisplay()
```

Этот метод, зарегистрированный с помощью функции `glutKeyboardFunc()`, вызывается, когда пользователь нажимает любую функциональную клавишу, стрелку, `Page Up`, `Page Down`, `Home`, `End` или `Insert`. В данном случае мы реагируем только на клавиши со стрелками – уменьшаем или увеличиваем угол поворота вокруг оси *x* и *y* и просим `GLUT` перерисовать окно. В результате вызывается метод `Scene.display()`, зарегистрированный с помощью функции `glutDisplayFunc()`.

Мы рассмотрели весь код программы `cylinder1.pyw`, написанной с использованием пакета PyOpenGL. Читатели, знакомые с OpenGL, наверное, почувствовали себя как дома, потому что вызовы OpenGL почти такие же, как на C.

8.1.2. Создание программы Cylinder с помощью pyglet

Версия на базе pyglet (`cylinder2.pyw`) устроена очень похоже на рассмотренную выше. Основное различие состоит в том, что pyglet предоставляет собственный интерфейс для обработки событий и создания окна, так что обращаться к `GLUT` не придется.

```
def main():
   caption = "Cylinder (pyglet)"
   width = height = SIZE
   resizable = True
   try:
      config = Config(sample_buffers=1, samples=4, depth_size=16,
               double_buffer=True)
      window = Window(width, height, caption=caption, config=config,
               resizable=resizable)
   except pyglet.window.NoSuchConfigException:
      window = Window(width, height, caption=caption,
               resizable=resizable)
   path = os.path.realpath(os.path.dirname(__file__))
   icon16 = pyglet.image.load(os.path.join(path, "cylinder_16x16.png"))
   icon32 = pyglet.image.load(os.path.join(path, "cylinder_32x32.png"))
   window.set_icon(icon16, icon32)
   pyglet.app.run()
```

Вместо того чтобы задавать контекст OpenGL в виде строки байтов, pyglet предоставляет объект `pyglet.gl.Config` для хранения настроек. Здесь мы сначала задаем желаемую конфигурацию, а потом на ее основе создаем свой объект `Window` (подкласс `pyglet.window.Window`); в случае ошибки создается окно с конфигурацией по умолчанию.

Приятно, что pyglet позволяет задать значок приложения, который обычно отображается в углу полосы заголовка и в области переключения задач. Создав окно и задав значки, мы запускаем цикл обработки событий pyglet.

```
class Window(pyglet.window.Window):

   def __init__(self, *args, **kwargs):
      super().__init__(*args, **kwargs)
      self.set_minimum_size(200, 200)
      self.xAngle = 0
      self.yAngle = 0
      self._initialize_gl()
      self._z_axis_list = pyglet.graphics.vertex_list(2,
```

```
        ("v3i", (0, 0, -1000, 0, 0, 1000)),
        ("c3B", (255, 0, 255) * 2)) # по одному цвету на вершину
```

Этот метод аналогичен эквивалентному методу класса `Scene`, который был рассмотрен в предыдущем подразделе. Одно из различий состоит в том, что мы задали минимальный размер окна. Как мы скоро увидим, pyglet умеет рисовать линии тремя способами. Третий заключается в том, чтобы нарисовать отрезки, соединяющие вершины из заданного списка, в котором указаны координаты и цвета. Здесь мы как раз и создаем такой список. Функция создания списка принимает количество пар (вершина, цвет) и саму последовательность пар. Каждая пара содержит форматную строку и последовательность. В данном случае форматная строка в первой паре означает «вершины, заданные тремя целочисленными координатами», так что здесь мы имеем две вершины. Форматная строка во второй паре означает «цвета, заданные тремя байтами без знака»; здесь указаны два цвета (одинаковых), по одному для каждой вершины.

Мы не показываем методы `_initialize_gl()`, `on_draw()`, `on_resize()` и `_draw_cylinder()`. Метод `_initialize_gl()` очень похож на встречавшийся в программе `cylinder1.pyw`. Тело метода `on_draw()`, который pyglet автоматически вызывает для отрисовки подклассов `pyglet.window.Window`, ничем не отличается от тела метода `Scene.display()` из программы `cylinder1.pyw`. Аналогично тело метода `on_resize()`, который вызывается при изменении размера окна, такое же, как у метода `Scene.reshape()` из ранее рассмотренной программы. Оба метода `_draw_cylinder()` (`Scene._draw_cylinder()` и `Window._draw_cylinder()`) идентичны.

```
def _draw_axes(self):
    glBegin(GL_LINES)                  # ось x (традиционный стиль)
    glColor3f(1, 0, 0)
    glVertex3f(-1000, 0, 0)
    glVertex3f(1000, 0, 0)
    glEnd()
    pyglet.graphics.draw(2, GL_LINES, # ось y (динамичный стиль pyglet)
            ("v3i", (0, -1000, 0, 0, 1000, 0)),
            ("c3B", (0, 0, 255) * 2))
    self._z_axis_list.draw(GL_LINES) # ось z (эффективный стиль pyglet)
```

Оси мы нарисовали разными способами, чтобы продемонстрировать имеющиеся возможности. Ось *x* нарисована с помощью стандартной функции OpenGL – точно так же, как в версии на базе PyOpenGL. При рисовании оси *y* мы попросили pyglet нарисовать отрезок, соединяющий две точки (их могло бы быть и больше), для ко-

торых задали координаты и цвета. Если количество отрезков велико, то этот способ должен быть чуть быстрее традиционного. Ось *z* нарисована самым эффективным способом: мы взяли готовый список пар (вершина, цвет), хранящийся в `pyglet.graphics.vertex_list` и попросили pyglet соединить вершины отрезками.

```
def on_text_motion(self, motion): # поворот вокруг оси x или y
   if motion == pyglet.window.key.MOTION_UP:
      self.xAngle -= ANGLE_INCREMENT
   elif motion == pyglet.window.key.MOTION_DOWN:
      self.xAngle += ANGLE_INCREMENT
   elif motion == pyglet.window.key.MOTION_LEFT:
      self.yAngle -= ANGLE_INCREMENT
   elif motion == pyglet.window.key.MOTION_RIGHT:
      self.yAngle += ANGLE_INCREMENT
```

Этот метод вызывается, когда пользователь нажимает клавишу со стрелкой. Делает он то же самое, что в предыдущем примере делал метод `special()`, только теперь для обозначения клавиш используются константы, определенные в pyglet, а не в `GLUT`.

Мы не стали определять метод `on_key_press()` (а если бы определили, он вызывался бы при нажатии других клавиш), поскольку по умолчанию pyglet и так закрывает окно (а значит, завершает программу) при нажатии `Esc`, а это как раз то, что нам нужно.

Каждая из программ рисования цилиндра насчитывает примерно 140 строк. Однако благодаря использованию списков вершин `pyglet.graphics.vertex_list` и других расширений pyglet мы получаем как удобство – особенно при обработке событий и создании окон, – так и эффективность.

8.2. Игра в ортографической проекции

В главе 7 мы привели код двухмерной игры Gravitate, опустив, правда, код рисования плиток. На самом деле, каждая плитка – это квадрат, окруженный со всех сторон четырьмя равнобедренными трапециями. Верхняя и левая трапеции рисуются чуть более светлым цветом, чем квадрат, а нижняя и правая – чуть более темным; в результате создается иллюзия трехмерности (см. рис. 7.7 и врезку «Игра Gravitate» на стр. 291).

В этом разделе мы опишем большую часть кода игры Gravitate 3D, показанной на рис. 8.2. В этой программе плитки заменены шариками,

между которыми оставлены промежутки, чтобы пользователь видел трехмерную структуру, поворачивая сцену вокруг осей *x* и *y*. Мы уделим основное внимание пользовательскому интерфейсу и коду, отвечающему за трехмерное представление, опустив низкоуровневые детали, связанные с реализацией логики игры. Полный код находится в файле `gravitate3d.pyw`.

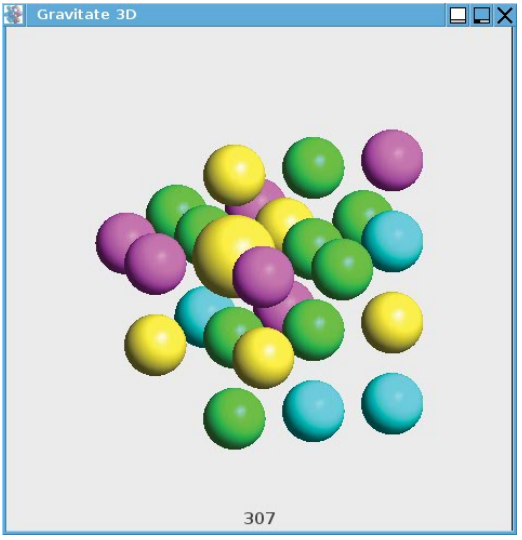

Рис. 8.2. Программа Gravitate 3D в Linux

Функция `main()` программы (не показана) отличается от одноименной функции программы `cylinder2.pyw` только текстом в заголовке окна и именами файлов, содержащих значки.

```
BOARD_SIZE = 4 # Должно быть > 1
ANGLE_INCREMENT = 5
RADIUS_FACTOR = 10
DELAY = 0.5    # в секундах
MIN_COLORS = 4
MAX_COLORS = min(len(COLORS), MIN_COLORS)
```

Это некоторые из встречающихся в программе констант. `BOARD_SIZE` – количество шариков вдоль каждой оси; если оно равно 4, то всего на доске будет $4 \times 4 \times 4 = 64$ шарика. Значение 5 константы `ANGLE_INCREMENT` означает, что при нажатии клавиши со стрелкой сцена поворачивается на угол 5°. `DELAY` – задержка между удалением

выделенного шарика, по которому пользователь щелкнул (и всех соседних с ним шариков того же цвета, равно как и *их* соседей), и сдвигом оставшихся шариков к центру для заполнения освободившегося места. `COLORS` (не показана) – список 3-кортежей целых чисел (в диапазоне от 0 до 255), представляющих цвета.

Когда пользователь щелкает по ранее не выбранному шарику, тот становится выбранным (а ранее выбранный перестает быть таковым). Визуально выбранный шарик отличается прочих тем, что его радиус в `RADIUS_FACTOR` раз больше. Если щелкнуть по выбранному шарику, то он сам и все шарики того же цвета, примыкающие к нему под углом 90° (но не по диагонали), их соседи и т. д. удаляются – при условии, что удалению подлежат хотя бы два шарика. В противном случае шарик просто перестает быть выбранным.

```
class Window(pyglet.window.Window):

   def __init__(self, *args, **kwargs):
      super().__init__(*args, **kwargs)
      self.set_minimum_size(200, 200)
      self.xAngle = 10
      self.yAngle = -15
      self.minColors = MIN_COLORS
      self.maxColors = MAX_COLORS
      self.delay = DELAY
      self.board_size = BOARD_SIZE
      self._initialize_gl()
      self.label = pyglet.text.Label("", bold=True, font_size=11,
               anchor_x="center")
      self._new_game()
```

В этом методе `__init__()` больше предложений, чем в соответствующем методе программы cylinder, потому что нужно задать цвета, задержку и размер доски. Кроме того, доска изначально повернута на некоторый угол, чтобы у пользователя не было сомнений в том, что игра трехмерная.

В pyglet есть очень полезная возможность – текстовые метки. В данном случае мы создаем пустую метку в нижней части сцены, разместив ее по центру. В ней мы будем показывать сообщения и текущий счет.

Для задания цвета фона и источника света мы обращаемся к методу `_initialize_gl()` (не показан, но аналогичен тому, что мы видели раньше). Всё настроив, мы запускаем игру.

```
def _new_game(self):
    self.score = 0
    self.gameOver = False
    self.selected = None
    self.selecting = False
    self.label.text = ("Click to Select • Click again to Delete • "
            "Arrows to Rotate")
    random.shuffle(COLORS)
    colors = COLORS[:self.maxColors]
    self.board = []
    for x in range(self.board_size):
        self.board.append([])
        for y in range(self.board_size):
            self.board[x].append([])
            for z in range(self.board_size):
                color = random.choice(colors)
                self.board[x][y].append(SceneObject(color))
```

Этот метод создает доску; цвета шариков выбираются случайно из списка `COLORS`, а всего используется не более `self.maxColors` цветов. Доска представлена списком списков объектов `SceneObject`. У каждого объекта имеется цвет (цвет шарика передается его конструктору) и цвет в выбранном состоянии (генерируется автоматически и используется при выборе, как объяснено ниже в подразделе 8.2.2).

Поскольку мы изменили текст метки, pyglet перерисовывает сцену (вызывая наш метод `on_draw()`), после чего новая игра становится видимой и ожидает действий пользователя.

8.2.1. Рисование сцены с доской

Когда сцена показывается впервые или приоткрывается после закрытия или перемещения перекрывавшего ее окна, pyglet вызывает метод `on_draw()`. А после изменения размера сцены (то есть размера содержащего ее окна) вызывается метод `on_resize()`.

```
def on_resize(self, width, height):
    size = min(self.width, self.height) / 2
    height = height if height else 1
    width = width if width else 1
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    if width <= height:
        glOrtho(-size, size, -size * height / width,
                size * height / width, -size, size)
    else:
```

```
            glOrtho(-size * width / height, size * width / height,
                    -size, size, -size, size)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
```

В игре Gravitate 3D мы пользуемся ортографической проекцией. Приведенный выше код будет работать без всяких изменений для любой сцены в ортографической проекции. (Так, если бы мы использовали PyOpenGL, то следовало бы назвать этот метод `reshape()` и зарегистрировать его с помощью функции `glutReshapeFunc()`.)

```
    def on_draw(self):
        diameter = min(self.width, self.height) / (self.board_size * 1.5)
        radius = diameter / 2
        offset = radius - ((diameter * self.board_size) / 2)
        radius = max(RADIUS_FACTOR, radius - RADIUS_FACTOR)
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glPushMatrix()
        glRotatef(self.xAngle, 1, 0, 0)
        glRotatef(self.yAngle, 0, 1, 0)
        with Selecting(self.selecting):
            self._draw_spheres(offset, radius, diameter)
        glPopMatrix()
        if self.label.text:
            self.label.y = (-self.height // 2) + 10
            self.label.draw()
```

Это эквивалент метода PyOpenGL `display()`, регистрируемого функцией `glutDisplayFunc()`. Мы хотим, чтобы доска заполняла столько места в объемлющем окне, сколько возможно без обрезки шариков при вращении. Кроме того, нужно вычислить смещение, при котором доска правильно выровнена по центру.

Позаботившись о предварительных условиях, мы поворачиваем сцену (если пользователь нажимает клавиши со стрелками), а затем рисуем шарики в контексте менеджера `Selecting`. Этот контекстный менеджер гарантирует включение или выключение определенных параметров в зависимости от того, рисуется ли схема реально – для просмотра пользователем – или вне экрана – чтобы определить, по какому шарику щелкнул пользователь (реализация выбора обсуждается в следующем подразделе).

Если метка не пуста, то мы изменяем ее координату `y`, так чтобы она находилась внизу окна (поскольку размер окна мог измениться), а затем просим ее нарисовать себя (то есть вывести свой текст).

```
def _draw_spheres(self, offset, radius, diameter):
   try:
      sphere = gluNewQuadric()
      gluQuadricNormals(sphere, GLU_SMOOTH)
      for x, y, z in itertools.product(range(self.board_size),
             repeat=3):
      sceneObject = self.board[x][y][z]
      if self.selecting:
         color = sceneObject.selectColor
      else:
         color = sceneObject.color
      if color is not None:
         self._draw_sphere(sphere, x, y, z, offset, radius,
               diameter, color)
   finally:
      gluDeleteQuadric(sphere)
```

С помощью квадрик можно рисовать не только цилиндры, но и сферы. В данном случае нам нужно нарисовать много сфер (до 64), а не всего один цилиндр, но для всех сфер можно использовать одну и ту же квадрику.

Мы не стали писать `for x in range(self.board.size): for y in range(self.board.size): for z in range(self.board.size):` для порождения тройки *x*, *y*, *z* для каждой сферы из списка списков списков, представляющего доску, а добились того же эффекта, воспользовавшись единственным циклом в сочетании с функцией `itertools.product()`.

Для каждой тройки мы получаем соответствующий объект на сцене (если объект был удален, то его цвета будут равны `None`) и устанавливаем цвет, совпадающий с цветом выделения, если рисование производится, для того чтобы узнать, по какому шарику щелкнул пользователь, или цвету шарика, если рисование производится для показа пользователю. Если цвет отличен от `None`, то шарик рисуется.

```
def _draw_sphere(self, sphere, x, y, z, offset, radius, diameter,
         color):
   if self.selected == (x, y, z):
      radius += RADIUS_FACTOR
   glPushMatrix()
   x = offset + (x * diameter)
   y = offset + (y * diameter)
   z = offset + (z * diameter)
   glTranslatef(x, y, z)
   glColor3ub(*color)
   gluSphere(sphere, radius, 24, 24)
   glPopMatrix()
```

Этот метод рисует шарик с нужным смещением в трехмерной сетке. Если данный шарик выбран, то при рисовании его радиус увеличивается. Последние два аргумента функции `gluSphere()` – коэффициенты частоты разбиения (чем они больше, тем более гладким получается результат и тем больше времени занимает его построение).

8.2.2. Обработка выбора объекта на сцене

Выбрать объект в трехмерном пространстве, изображенном на двухмерной плоскости, не так-то просто! За много лет были разработаны различные приемы, в Gravitate 3D мы воспользовались наиболее надежным и широко распространенным.

Принцип работы такой. Когда пользователь щелкает по сцене, сцена перерисовывается во внеэкранном буфере, невидимом пользователю, причем каждый объект рисуется уникальным цветом. Затем мы смотрим, какой цвет оказался в той точке буфера, где пользователь щелкнул мышью. Зная цвет, можно определить объект, который этим цветом нарисован. Чтобы все получилось правильно, необходимо рисовать сцену, отключив сглаживание, освещение, текстуры, – тогда у каждого объекта действительно будет уникальный цвет, не измененный в ходе дополнительной обработки.

Сначала мы рассмотрим класс `SceneObject`, с помощью которого представлены все шарики, а затем перейдем к контекстному менеджеру `Selecting`.

```
class SceneObject:

   __SelectColor = 0

   def __init__(self, color):
      self.color = color
      SceneObject.__SelectColor += 1
      self.selectColor = SceneObject.__SelectColor
```

Каждому объекту на сцене сопоставляется цвет, которым он отображается (`self.color`), – он не обязан быть уникальным – и уникальный цвет выделения. Статический атрибут `__SelectColor` увеличивается на 1 при добавлении на сцену каждого нового объекта и служит для того, чтобы назначить объекту уникальный цвет выделения.

```
   @property
      def selectColor(self):
         return self.__selectColor
```

Это свойство возвращает цвет выделения объекта, который либо равен `None` (если объект удален), либо содержит 3-кортеж целочисленных компонент цвета (каждая в диапазоне от 0 до 255).

```
@selectColor.setter
def selectColor(self, value):
   if value is None or isinstance(value, tuple):
      self.__selectColor = value
   else:
      parts = []
      for _ in range(3):
         value, y = divmod(value, 256)
         parts.append(y)
      self.__selectColor = tuple(parts)
```

Этот метод установки цвета выделения принимает переданное значение, если оно равно `None` или содержит кортеж; в противном случае он вычисляет уникальный кортеж цветов, исходя из полученного уникального целого числа. Для первого объекта на сцене передается число 1, ему будет назначен цвет `(1, 0, 0)`. Для второго передано 2, его цвет – `(2, 0, 0)` и т. д. вплоть до 255-го объекта, которому будет назначен цвет `(255, 0, 0)`. На 256-й раз получается цвет `(0, 1, 0)`, на 257-й – `(1, 1, 0)`, на 258-й – `(2, 1, 0)` и т. д. Такая система позволяет создать свыше 16 миллионов уникальных объектов, в большинстве случаев этого достаточно.

```
SELECTING_ENUMS = (GL_ALPHA_TEST, GL_DEPTH_TEST, GL_DITHER,
       GL_LIGHT0, GL_LIGHTING, GL_MULTISAMPLE, GL_TEXTURE_1D,
       GL_TEXTURE_2D, GL_TEXTURE_3D)
```

Нам нужно включать или выключать сглаживание, освещение, текстуры и вообще все, что может изменить цвет объекта, в зависимости от того, рисуем мы на экране (для показа пользователю) или вне экрана (чтобы узнать, по какому объекту пользователь щелкнул). Выше приведен список всех параметров OpenGL, которые влияют на цвет объекта в программе Gravitate 3D.

```
class Selecting:

   def __init__(self, selecting):
      self.selecting = selecting
```

Контекстный менеджер `Selecting` запоминает, для чего рисуются шарики в текущем контексте: для показа пользователю или для определения выбранного объекта, то есть для его выделения.

```
def __enter__(self):
   if self.selecting:
      for enum in SELECTING_ENUMS:
         glDisable(enum)
   glShadeModel(GL_FLAT)
```

Если при входе в контекстный менеджер рисование производится для выделения, то мы отключаем все механизмы OpenGL, способные изменить цвет, и переходим к модели плоского затенения.

```
def __exit__(self, exc_type, exc_value, traceback):
   if self.selecting:
      for enum in SELECTING_ENUMS:
         glEnable(enum)
      glShadeModel(GL_SMOOTH)
```

Если при выходе из контекстного менеджера выясняется, что рисование производилось для выделения, то мы включаем все ранее выключенные механизмы и переходим к модели плавного затенения.

Как работает механизм выделения, можно наблюдать, если внести два изменения в исходный код. Во-первых, вместо `+= 1` в методе `SceneObject.__init__()` напишите `+= 500`. А во-вторых, закомментируйте предложение `self.selecting = False` в методе `Window.on_mouse_press()` (он будет рассмотрен в следующем подразделе). Теперь запустите программу и щелкните по любому шарику: то, что обычно перерисовывается во внеэкранном буфере, теперь видно зрителю, а во всех остальных отношениях программа работает нормально.

8.2.3. Обработка взаимодействия с пользователем

В игру Gravitate 3D играют преимущественно с помощью мыши. Однако предусмотрены и клавиши для вращения доски, начала и завершения игры.

```
def on_mouse_press(self, x, y, button, modifiers):
   if self.gameOver:
      self._new_game()
      return
   self.selecting = True
   self.on_draw()
   self.selecting = False
   selectColor = (GLubyte * 3)()
```

```
glReadPixels(x, y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, selectColor)
selectColor = tuple([component for component in selectColor])
self._clicked(selectColor)
```

Pyglet вызывает этот метод, когда пользователь щелкает кнопкой мыши (при условии, что метод определен). Если игра закончена, то щелчок интерпретируется как желание начать новую игру. В противном случае мы предполагаем, что пользователь хотел щелкнуть по шарику. Мы устанавливаем флаг `selecting` в `True` и перерисовываем сцену (поскольку это делается вне экрана, то пользователь ничего не заметит), после чего сбрасываем флаг в `False`.

Функция `glReadPixels()` читает цвета одного или нескольких пикселей; в данном случае мы пользуемся ей, чтобы прочитать пиксель внеэкранного буфера, находящийся в той позиции, по которой пользователь щелкнул, и получить его RGB-значение в виде трех байтов без знака. Затем эти три байта помещаются в 3-кортеж целых чисел, чтобы можно было сравнить цвет с уникальным цветом выделения каждого объекта.

Отметим, что при вызове `glReadPixels()` предполагается, что начало системы координат расположено в левом нижнем углу (как в pyglet). Если бы начало координат находилось в левом верхнем углу, то надо было бы написать еще два предложения: `viewport = (GLint * 4)(); glGetIntegerv(GL_VIEWPORT, viewport)`, а параметр `y` при обращении к `glReadPixels()` нужно было бы заменить на `viewport[3] - y`.

```
def _clicked(self, selectColor):
    for x, y, z in itertools.product(range(self.board_size), repeat=3):
        if selectColor == self.board[x][y][z].selectColor:
            if (x, y, z) == self.selected:
                self._delete() # Второй щелчок удаляет
            else:
                self.selected = (x, y, z)
            return
```

Этот метод вызывается, когда пользователь щелкает мышью, кроме случая, когда щелчок начинает новую игру. Функция `itertools.product()` порождает все возможные тройки координат *x*, *y*, *z* на доске и сравнивает цвет выделения объекта в соответствующей позиции с цветом пикселя в точке щелчка. Если цвета совпадают, то мы нашли объект, по которому щелкнул пользователь. Если этот объект уже выделен, значит, пользователь щелкает по нему второй раз, поэтому мы пытаемся удалить как сам объект, так и примыкающие к

нему объекты того же цвета. В противном случае щелчок сделан, для того чтобы выбрать объект (и снять выделение с ранее выбранного).

```
def _delete(self):
    x, y, z = self.selected
    self.selected = None
    color = self.board[x][y][z].color
    if not self._is_legal(x, y, z, color):
        return
    self._delete_adjoining(x, y, z, color)
    self.label.text = "{:,}".format(self.score)
    pyglet.clock.schedule_once(self._close_up, self.delay)
```

Этот метод удаляет выбранный шарик и примыкающие к нему шарики того же цвета (а также примыкающие к *ним*). Сначала мы снимаем выделение с выбранного шарика, а затем проверяем, можно ли что-то удалять (это можно делать, если к шарику примыкает еще хотя бы один того же цвета). Если удаление допустимо, то мы производим его с помощью метода `_delete_adjoining()` и вызываемых из него (они не показаны). Затем записываем в метку новый счет и планируем вызов метода `self._close_up()` (не показан) через полсекунды. Это дает пользователю возможность увидеть, какие шарики были удалены, прежде чем освободившееся место схлопнется из-за притягивания шариков к центру. (Можно было бы поступить более изощренно – анимировать схлопывание, медленно перемещая каждый шарик в предназначенное ему место).

```
def on_key_press(self, symbol, modifiers):
    if (symbol == pyglet.window.key.ESCAPE or
        ((modifiers & pyglet.window.key.MOD_CTRL or
          modifiers & pyglet.window.key.MOD_COMMAND) and
          symbol == pyglet.window.key.Q)):
        pyglet.app.exit()
    elif ((modifiers & pyglet.window.key.MOD_CTRL or
           modifiers & pyglet.window.key.MOD_COMMAND) and
           symbol == pyglet.window.key.N):
        self._new_game()
    elif (symbol in {pyglet.window.key.DELETE, pyglet.window.key.SPACE,
                     pyglet.window.key.BACKSPACE} and
          self.selected is not None):
        self._delete()
```

Пользователь всегда может выйти из программы, нажав кнопку ×, но мы хотим, чтобы это можно было сделать также клавишами `Esc` или `Ctrl+Q` (или ⌘Q). Чтобы начать новую игру, когда текущая завершилась, пользователю достаточно просто щелкнуть мышью; но мож-

но также сделать это в любой момент с помощью клавиши `Ctrl+N` (или `⌘N`). Мы хотим также, чтобы выбранный шарик (и примыкающие к нему) можно было удалить не только вторым щелчком по нему, но и нажатием клавиш `Del`, `Space` или `Backspace`.

В классе `Window` есть также метод `on_text_motion()`, который обрабатывает нажатия клавиш со стрелками и поворачивает сцену вокруг оси *x* или *y*. Этот метод здесь не приводится, потому что он ничем не отличается от того, что мы видели раньше (в подразделе 8.1.2).

На этом мы завершаем рассмотрение игры Gravitate 3D. За кадром остались только методы, относящиеся к деталям реализации логики игры, в частности те, что занимаются удалением примыкающих шариков (для этого цвет и цвет выделения устанавливаются равными `None`) и сдвигом шариков к центру сцены.

Программное создание трехмерных сцен – вещь нетривиальная, в частности, потому что традиционные интерфейсы OpenGL чисто процедурные (основанные на функциях), а не объектно-ориентированные. Тем не менее, благодаря PyOpenGL и pyglet нетрудно перенести написанный на C код для OpenGL на Python. Кроме того, pyglet предоставляет собственную удобную поддержку для обработки событий и создания окон, а PyOpenGL интегрируется со многими библиотеками ГИП, в том числе Tkinter.

ПРИЛОЖЕНИЕ А. Эпилог

В этой книге мы рассмотрели много интересных приемов и познакомились с рядом полезных библиотек. Хочется надеяться, что по ходу чтения у вас появились идеи, как лучше писать программы на Python 3 (`www.python.org`).

Популярность Python продолжает расти, он все шире используется в разных предметных областях и на всех континентах. Python идеален в качестве первого изучаемого языка программирования, поскольку поддерживает процедурную, объектно-ориентированную и функциональную парадигмы, и при этом его синтаксис понятен, не отягощен излишними «наворотами» и последователен. Но Python еще и отличный язык для профессионального программирования (что Google демонстрирует уже на протяжении многих лет). В немалой степени это объясняется тем, что Python поддерживает быструю разработку, что получающийся код удобен для сопровождения и что очень легко получить доступ к богатейшей функциональности, написанной на C и других компилируемых языках, поддерживающих принятые в C соглашения о вызове.

В программировании на Python нет тупиков: всегда есть чему еще поучиться и куда двигаться дальше. Поэтому хотя Python удовлетворяет потребности начинающих программистов, в нем есть такие продвинутые возможности и такие интеллектуальные глубины, которые придутся по вкусу самым взыскательным специалистам. Python открыт не только в смысле лицензирования и доступности исходного кода, но и для интроспекции – вплоть до байт-кода, если вам это нужно. И разумеется, Python открыт для всех, кто хочет внести свой вклад в его разработку (`docs.python.org/devguide`).

Наверное, существует несколько тысяч компьютерных языков (хотя широко употребляются лишь несколько десятков), и сегодня Python, безусловно, один из самых популярных. Как специалист по информатике, которому за несколько десятков лет довелось работать

с самыми разными языками, я частенько бывал раздосадован тем, какие языки выбирали мои работодатели. И, наверное, как и многие ученые, я не переставал думать, что надо бы создать язык получше – такой, в котором не было бы проблем и неудобств языков, к которым я привык, такой, который включал бы все лучшее из известных мне языков. С годами я обнаружил, что любой идеальный язык, о котором мне мечталось, оборачивается Python'ом, хотя и с кое-какими отсутствующими в нем особенностями – константами, факультативной типизацией и контролем доступа (закрытые атрибуты). Поэтому теперь я не жажду создать собственный идеальный язык программирования – я им пользуюсь. Спасибо тебе, Гвидо ван Россум, и всем остальным разработчикам Python, прошлым и нынешним, за то, что вы подарили миру язык программирования и экосистему невообразимо мощные и полезные, которые работают почти всюду и доставляют радость своим пользователям.

ПРИЛОЖЕНИЕ В. Краткая библиография

C++ Concurrency in Action: Practical Multithreading[1]

Anthony Williams (Manning Publications, Co., 2012, ISBN-13: 978-1-933988-77-1)

Это книга о параллельном программировании на C++, но ценна она тем, что рассказывает о многих проблемах и подводных камнях, подстерегающих разработчика параллельных программ (на любом языке), и объясняет, как их избежать.

Clean Code: A Handbook of Agile Software Craftsmanship[2]

Robert C. Martin (Prentice Hall, 2009, ISBN-13: 978-0-13-235088-4)

В этой книге рассматриваются разнообразные «тактические» вопросы программирования: выбор хороших имен, проектирование функций, рефакторинг и т. п. В книге немало идей, которые должны помочь любому программисту улучшить стиль кодирования и сделать свои программы более удобными для сопровождения. (Примеры в книге написаны на Java.)

Code Complete: A Practical Handbook of Software Construction, Second Edition[3]

Steve McConnell (Microsoft Press, 2004, ISBN-13: 978-0-7356-1967-8)

Это книга о том, как писать надежные, как скала, программы. Автор не ограничивается особенностями конкретного языка, а углубляется в область идей, принципов и практических приемов. Книга изобилует примерами, которые заставят любого программиста задуматься о своей профессии.

---

1 Э. Уильямс «Параллельное программирование на C++ в действии. Практика разработки многопоточных программ», ДМК Пресс, 2012.

2 Р. Мартин «Чистый код: создание, анализ и рефакторинг», Питер, 2013

3 С. Макконнелл «Совершенный код. Практическое руководство по разработке программного обеспечения», Русская Редакция, 2013

Design Patterns: Elements of Reusable Object-Oriented Software[4]

Erich Gamma,Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995, ISBN-13: 978-0-201-63361-0)

Одна из книг по программированию, оказавших наибольшее влияние на современные представления об этом предмете. Паттерны проектирования очаровывают и находят практическое применения в повседневной работе. (Примеры в книге по большей части написаны на C++.)

Domain-Driven Design: Tackling Complexity in the Heart of Software[5]

Eric Evans (Addison-Wesley, 2004, ISBN-13: 978-0-321-12521-7)

Очень интересная книга о проектировании программного обеспечения. Особенно полезна при разработке крупных проектов силами большого коллектива. Посвящена главным образом созданию и уточнению моделей предметной области, для которой проектируется система, а также созданию единого языка, на котором могут выражать свои мысли все причастные к ее созданию, а не только инженеры-программисты.

Don't Make Me Think!: A Common Sense Approach to Web Usability, Second Edition

Steve Krug (New Riders, 2006, ISBN-13: 978-0-321-34475-5)

Короткая, ориентированная на практиков книжка об удобстве пользования веб-приложениями, основанная на исследованиях и опыте автора. Применение доходчиво изложенных в этой книге идей позволит улучшить сайт любого размера.

GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos

Jeff Johnson (Morgan Kaufmann, 2008, ISBN-13: 978-0-12-370643-0)

Пусть вас не вводит в заблуждение шутливое название; это серьезная книга, которую должен прочитать каждый программист, разрабатывающий пользовательские интерфейсы. Наверное, вы согласитесь не с каждым предложением, но наверняка призадумаетесь и почерпнете полезные идеи о проектировании ГИП.

Java Concurrency in Practice

Brian Goetz, et. al. (Addison-Wesley, 2006, ISBN-13: 978-0-321-34960-6)

---

4 Э. Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2013

5 Э. Эванс «Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем», Вильямс, 2010

Отличное изложение вопросов параллелизма на Java. Книга содержит множество советов по поводу параллельного программирования, применимых к любому языку.

The Little Manual of API Design

Jasmin Blanchette (Trolltech/Nokia, 2008)

Очень короткое руководство (доступное бесплатно на странице `www21.in.tum.de/~blanchet/api-design.pdf`), которое содержит мысли по поводу проектирования API на примере библиотеки Qt.

Mastering Regular Expressions, Third Edition[6]

Jeffrey E.F. Friedl (O'Reilly Media, 2006, ISBN-13: 978-0-596-52812-6)

Стандартный учебник по регулярным выражениям. Написан доходчиво, содержит множество практических примеров с подробными объяснениями.

OpenGL SuperBible: Comprehensive Tutorial and Reference, Fourth Edition

Richard S. Wright, Jr., Benjamin Lipchak, and Nicholas Haemel (Addison-Wesley, 2007, ISBN-13: 978-0-321-49882-3)

Хорошее введение в трехмерную графику с применением OpenGL, подходит для программистов, не имеющих опыта в этой области. Примеры написаны на C++, но API OpenGL очень точно повторены в pyglet и других пакетах на Python, предлагающих интерфейс к OpenGL, так что для адаптации примеров не потребуется больших усилий.

Programming in Python 3: A Complete Introduction to the Python Language, Second Edition[7]

Mark Summerfield (Addison-Wesley, 2010, ISBN-13: 978-0-321-68056-3)

Эта книга научит программировать на Python 3 любого, кто умеет писать программы на каком-нибудь традиционном процедурном или объектно-ориентированном языке (включая, разумеется, Python 2).

Python Cookbook, Third Edition

David Beazley and Brian K. Jones (O'Reilly Media, 2013, ISBN-13: 978-1-4493-4037-7)

6 Дж. Фридл «Регулярные выражения», Символ-Плюс, 2008

7 М. Саммерфилд «Программирование на Python 3. Подробное руководство», Символ-Плюс, 2009 (перевод первого издания)

Книга изобилует интересными и практически ориентированными идеями, охватывающими все стороны программирования на Python 3. Отличное дополнение к «Python in Practice».

Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming

Mark Summerfield (Prentice Hall, 2008, ISBN-13: 978-0-13-235418-9)

В этой книге речь идет о программирования ГИП с помощью Python 2 и библиотеки Qt 4. Версии примеров из этой книги на Python 3 имеются на сайте автора, и почти все изложенное применимо не только к PyQt, но и к PySide.

Security in Computing, Fourth Edition

Charles P. Pfleeger and Shari Lawrence Pfleeger (Prentice Hall, 2007, ISBN-13: 978-0-13-239077-4)

Интересное, полезное и практичное изложение широкого круга вопросов, относящихся к компьютерной безопасности. Объясняется, как организуются атаки и как от них защититься.

Tcl and the Tk Toolkit, Second Edition

John K. Ousterhout and Ken Jones (Addison-Wesley, 2010, ISBN-13: 978-0-321-33633-0)

Стандартный учебник по Tcl/Tk 8.5. Tcl – нетрадиционный, почти лишенный синтаксиса язык, но эта книга поможет понять, как читать документацию по Tcl/Tk, что часто бывает необходимо при разработке приложений с помощью Python и Tkinter.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

D

F

G

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. 8**(499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Марк Саммерфилд

Python на практике

Создание качественных программ
с использованием параллелизма, библиотек и паттернов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 $^{1}/_{16}$. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 20,70.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф