ОГЛАВЛЕНИЕ

Предисловие **9**

Введение **11**

Благодарности 14

Глава 1. Порождающие паттерны проектирования в Python **16**

1.1. Паттерн Абстрактная фабрика 16

1.1.1. Классическая Абстрактная фабрика 17

1.1.2. Абстрактная фабрика в духе Python 20

1.2. Паттерн Построитель 22

1.3. Паттерн Фабричный метод 28

1.4. Паттерн Прототип 37

1.5. Паттерн Одиночка 38

Глава 2. Структурные паттерны проектирования в Python **40**

2.1. Паттерн Адаптер 40

2.2. Паттерн Мост 46

2.3. Паттерн Компоновщик 52

2.3.1. Классическая иерархия составных и несоставных объектов 53

2.3.2. Единый класс для составных и несоставных объектов 57

2.4. Паттерн Декоратор 60

2.4.1. Декораторы функций и методов 61

2.4.2. Декораторы классов 67

2.5. Паттерн Фасад 74

2.6. Паттерн Приспособленец 79

2.7. Паттерн Заместитель 82

Глава 3. Поведенческие паттерны проектирования в Python **88**

3.1. Паттерн Цепочка ответственности 88

3.1.1. Традиционная Цепочка 89

3.1.2. Цепочка на основе сопрограмм 91

3.2. Паттерн Команда 95

3.3. Паттерн Интерпретатор ... 99
3.3.1. Вычисление выражения с помощью eval() ... 100
3.3.2. Исполнение кода с помощью exec() ... 104
3.3.3. Исполнение кода в подпроцессе ... 107
3.4. Паттерн Итератор ... 112
3.4.1. Итераторы, следующие протоколу последовательности ... 112
3.4.2. Реализация итераторов с помощью функции iter() с двумя аргументами ... 113
3.4.3. Итераторы на базе протокола итераторов ... 115
3.5. Паттерн Посредник ... 118
3.5.1. Традиционный Посредник ... 119
3.5.2. Посредник на основе сопрограмм ... 123
3.6. Паттерн Хранитель ... 125
3.7. Паттерн Наблюдатель ... 125
3.8. Паттерн Состояние ... 130
3.8.1. Чувствительные к состоянию методы ... 133
3.8.2. Определяемые состоянием методы ... 135
3.9. Паттерн Стратегия ... 136
3.10. Паттерн Шаблонный метод ... 139
3.11. Паттерн Посетитель ... 142
3.12. Пример: пакет обработки изображений ... 144
3.12.1. Общий модуль обработки изображений ... 146
3.12.2. Обзор модуля Xpm ... 156
3.12.3. Модуль-обертка PNG ... 159

Глава 4. Высокоуровневый параллелизм в Python ... 162
4.1. Распараллеливание задач с большим объемом вычислений ... 166
4.1.1. Очереди и многопроцессная обработка ... 169
4.1.2. Будущие объекты и многопроцессная обработка ... 175
4.2. Распараллеливание задач, ограниченных скоростью ввода-вывода ... 178
4.2.1. Очереди и многопоточность ... 180
4.2.2. Будущие объекты и многопоточность ... 185
4.3. Пример: приложение с параллельным ГИП ... 188
4.3.1. Создание ГИП ... 190
4.3.2. Модуль ImageScaleWorker ... 198
4.3.3. Как ГИП обрабатывает продвижение ... 201
4.3.4. Как ГИП обрабатывает выход из программы ... 203

Глава 5. Расширение Python ... 205
5.1. Доступ к написанным на C библиотекам с помощью пакета ctypes ... 207

5.2. Использование Cython ... 215
5.2.1. Доступ к написанным на C библиотекам с помощью Cython ... 215
5.2.2. Создание Cython-модулей для повышения производительности ... 222
5.3. Пример: ускоренная версия пакета Image ... 228

Глава 6. Высокоуровневое сетевое программирование на Python ... 233
6.1. Создание приложений на базе технологии XML-RPC ... 234
6.1.1. Обертка данных ... 235
6.1.2. Разработка сервера XML-RPC ... 239
6.1.3. Разработка клиента XML-RPC ... 241
6.2. Создание приложений на базе технологии RPyC ... 251
6.2.1. Потокобезопасная обертка данных ... 251
6.2.2. Разработка сервера RPyC ... 257
6.2.3. Разработка клиента RPyC ... 260

Глава 7. Графические интерфейсы пользователя на Python и Tkinter ... 264
7.1. Введение в Tkinter ... 267
7.2. Создание диалоговых окон с помощью Tkinter ... 269
7.2.1. Создание диалогового приложения ... 271
7.2.2. Создание диалоговых окон в приложении ... 280
7.3. Создание приложений с главным окном с помощью Tkinter ... 290
7.3.1. Создание главного окна ... 292
7.3.2. Создание меню ... 294
7.3.3. Создание строки состояния с индикаторами ... 297

Глава 8. Трехмерная графика на Python с применением OpenGL ... 301
8.1. Сцена в перспективной проекции ... 303
8.1.1. Создание программы Cylinder с помощью PyOpenGL ... 304
8.1.2. Создание программы Cylinder с помощью pyglet ... 309
8.2. Игра в ортографической проекции ... 311
8.2.1. Рисование сцены с доской ... 314
8.2.2. Обработка выбора объекта на сцене ... 317
8.2.3. Обработка взаимодействия с пользователем ... 319

Приложение A. Эпилог ... 323

Приложение B. Краткая библиография ... 325

Предметный указатель ... 329

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 С-библиотеки можно воспользоваться модулем ctypes из стандартной библиотеки или сторонним пакетом Cython. Использование готовых С-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].

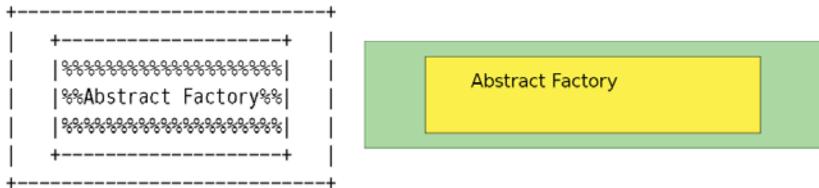

Рис. 1.1. Рисунки в виде простого текста и SVG

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`.

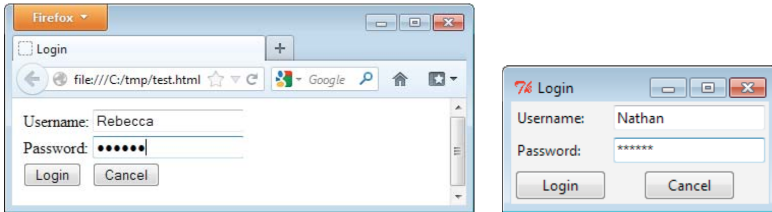

Рис. 1.2. Формы на HTML и Tkinter в Windows

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.

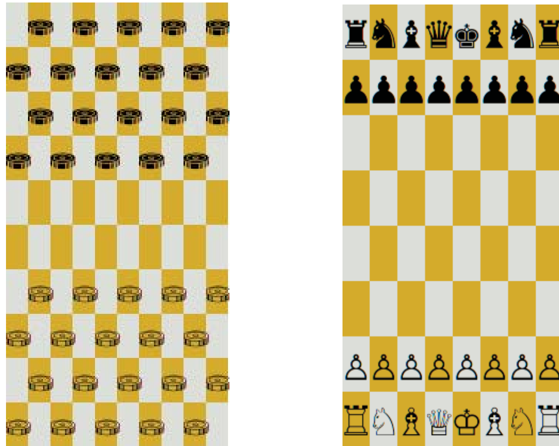

Рис. 1.3. Шашечная и шахматная доска на консоли Linux

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).