Оглавление

Об авторе

Лусиану Рамальо был веб-разработчиком до выхода компании Netscape на IPO в 1995 году, а в 1998 году перешел с Perl на Java, а затем на Python. В 2015 году пришел в компанию Thoughtworks, где работает главным консультантом в отделении в Сан-Паулу. Он выступал с основными докладами, презентациями и пособиями на различных мероприятиях, связанных с Python, в обеих Америках, Европе и Азии. Выступал также на конференциях по Go и Elixir по вопросам проектирования языков. Рамальо – член фонда Python Software Foundation и сооснователь клуба Garoa Hacker Clube, первого места для общения хакеров в Бразилии.

Колофон

На обложке изображена намакская песчаная ящерица (*Pedioplanis namaquensis*), встречающаяся в засушливых саваннах и полупустынях Намибии.

Внешние признаки: туловище черное с четырьмя белыми полосками на спине, лапы коричневые с белыми пятнышками, брюшко белое, длинный розовато-коричневый хвост. Одна из самых быстрых ящериц, активна в течение дня, питается мелкими насекомыми. Обитает на бедных растительностью песчано-каменистых равнинах. Самка откладывает от трех до пяти яиц в ноябре. Остаток зимы ящерицы спят в норах, которые роют в корнях кустов.

В настоящее время охранный статус намакской песчаной ящерицы – «пониженная уязвимость». Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для мира.

Предисловие

> План такой: если кто-то пользуется средством, которое вы не понимаете, просто пристрелите его. Это проще, чем учить что-то новое, и очень скоро в мире останутся только кодировщики, которые используют только всем понятное крохотное подмножество Python 0.9.6 <смешок>.
>
> – Тим Питерс, легендарный разработчик ядра и автор сборника поучений «The Zen of Python»[1]

«Python – простой для изучения и мощный язык программирования». Это первые слова в официальном «Пособии по Python» (*https://docs.python.org/3/tutorial/*). И это правда, но не вся правда: поскольку язык так просто выучить и начать применять на деле, многие практикующие программисты используют лишь малую часть его обширных возможностей.

Опытный программист может написать полезный код на Python уже через несколько часов изучения. Но вот проходят недели, месяцы – и многие разработчики так и продолжают писать на Python код, в котором отчетливо видно влияние языков, которые они учили раньше. И даже если Python – ваш первый язык, все равно авторы академических и вводных учебников зачастую излагают его, тщательно избегая особенностей, характерных только для этого языка.

Будучи преподавателем, который знакомит с Python программистов, знающих другие языки, я нередко сталкиваюсь еще с одной проблемой, которую пытаюсь решить в этой книге: нас интересует только то, о чем мы уже знаем. Любой программист, знакомый с каким-то другим языком, догадывается, что Python поддерживает регулярные выражения, и начинает смотреть, что про них написано в документации. Но если вы никогда раньше не слыхали о распаковке кортежей или о дескрипторах, то, скорее всего, и искать сведения о них не станете, а в результате не будете использовать эти средства лишь потому, что они специфичны для Python.

Эта книга не является полным справочным руководством по Python. Упор в ней сделан на языковые средства, которые либо уникальны для Python, либо отсутствуют во многих других популярных языках. Кроме того, в книге рассматривается в основном ядро языка и немногие библиотеки. Я редко упоминаю о пакетах, не включенных в стандартную библиотеку, хотя нынче количество пакетов для Python уже перевалило за 60 000 и многие из них исключительно полезны.

На кого рассчитана эта книга

Эта книга написана для практикующих программистов на Python, которые хотят усовершенствоваться в Python 3. Я тестировал примеры на Python 3.10,

[1] Сообщение в группе Usenet comp.lang.python от 23 декабря 2002: «Acrimony in c.l.p.» (*https://mail.python.org/pipermail/python-list/2002-December/147293.html*).

а большую их часть также на Python 3.9 и 3.8. Если какой-то пример требует версии 3.10, то это явно оговаривается.

Если вы не уверены в том, достаточно ли хорошо знаете Python, чтобы читать эту книгу, загляните в оглавление официального «Пособия по Python» (*https://docs.python.org/3/tutorial/*). Темы, рассмотренные в пособии, в этой книге не затрагиваются, за исключением некоторых новых средств.

На кого эта книга не рассчитана

Если вы только начинаете изучать Python, эта книга покажется вам сложноватой. Более того, если вы откроете ее на слишком раннем этапе путешествия в мир Python, то может сложиться впечатление, будто в каждом Python-скрипте следует использовать специальные методы и приемы метапрограммирования. Преждевременное абстрагирование ничем не лучше преждевременной оптимизации.

Пять книг в одной

Я рекомендую всем прочитать главу 1 «Модель данных в языке Python». Читатели этой книги в большинстве своем после ознакомления с главой 1, скорее всего, смогут легко перейти к любой части, но я зачастую предполагаю, что главы каждой части читаются по порядку. Части I–V можно рассматривать как отдельные книги внутри книги.

Я старался сначала рассказывать о том, что уже есть, а лишь затем о том, как создавать что-то свое. Например, в главе 2 части II рассматриваются готовые типы последовательностей, в том числе не слишком хорошо известные, например `collections.deque`. О создании пользовательских последовательностей речь пойдет только в части III, где мы также узнаем об использовании абстрактных базовых классов (abstract base classes – ABC) из модуля `collections.abc`. Создание собственного ABC обсуждается еще позже, поскольку я считаю, что сначала нужно освоиться с использованием ABC, а уж потом писать свои.

У такого подхода несколько достоинств. Прежде всего, зная, что есть в вашем распоряжении, вы не станете заново изобретать велосипед. Мы пользуемся готовыми классами коллекций чаще, чем реализуем собственные, и можем уделить больше внимания нетривиальным способам работы с имеющимися средствами, отложив на потом разговор о разработке новых. И мы скорее унаследуем существующему абстрактному базовому классу, чем будем создавать новый с нуля. Наконец, я полагаю, что понять абстракцию проще после того, как видел ее в действии.

Недостаток же такой стратегии в том, что главы изобилуют ссылками на более поздние материалы. Надеюсь, что теперь, когда вы узнали, почему я избрал такой путь, вам будет проще смириться с этим.

Как организована эта книга

Ниже описаны основные темы, рассматриваемые в каждой части книги.

Часть I «Структуры данных»

В главе I, посвященной модели данных в Python, объясняется ключевая роль специальных методов (например, `__repr__`) для обеспечения единообразного поведения объектов любого типа. Специальные методы более подробно обсуждаются на протяжении всей книги. В остальных главах этой части рассматривается использование типов коллекций: последовательностей, отображений и множеств, а также различие между типами `str` и `bytes` – то, что радостно приветствовали пользователи Python 3 и чего отчаянно не хватает пользователям Python 2, еще не модернизировавшим свой код. Также рассматриваются высокоуровневые построители классов, имеющиеся в стандартной библиотеке: фабрики именованных кортежей и декоратор `@dataclass`. Сопоставление с образцом – новая возможность, появившаяся в Python 3.10, – рассматривается в разделах глав 2, 3 и 5, где обсуждаются паттерны последовательностей, отображений и классов. Последняя глава части I посвящена жизненному циклу объектов: ссылкам, изменяемости и сборке мусора.

Часть II «Функции как объекты»

Здесь речь пойдет о функциях как полноправных объектах языка: что под этим понимается, как это отражается на некоторых популярных паттернах проектирования и как реализовать декораторы функций с помощью замыканий. Рассматриваются также следующие вопросы: общая идея вызываемых объектов, атрибуты функций, интроспекция, аннотации параметров и появившееся в Python 3 объявление `nonlocal`. Глава 8 содержит введение в новую важную тему – аннотации типов в сигнатурах функций.

Часть III «Классы и протоколы»

Теперь наше внимание перемещается на создание классов «вручную» – в отличие от использования построителей классов, рассмотренных в главе 5. Как и в любом объектно-ориентированном (ОО) языке, в Python имеется свой набор средств; какие-то из них, возможно, присутствовали в языке, с которого вы и я начинали изучение программирования на основе классов, а какие-то – нет. В главах из этой части объясняется, как создать свою коллекцию, абстрактный базовый класс (ABC) и протокол, как работать со множественным наследованием и как реализовать перегрузку операторов (если это имеет смысл). В главе 15 мы продолжим обсуждать аннотации типов.

Часть IV «Поток управления»

Эта часть посвящена языковым конструкциям и библиотекам, выходящим за рамки последовательного потока управления с его условными выражениями, циклами и подпрограммами. Сначала мы рассматриваем генераторы, затем – контекстные менеджеры и сопрограммы, в том числе трудную для понимания, но исключительно полезную новую конструкцию `yield from`. В главу 18 включен важный пример использования сопоставления с образцом в простом, но функциональном интерпретаторе языка. Глава 19 «Модели конкурентности в Python» новая, она посвящена обзору различных

видов конкурентной и параллельной обработки в Python, их ограничений и вопросу о том, как архитектура программы позволяет использовать Python в приложениях масштаба веба. Я переписал главу об *асинхронном программировании*, стремясь уделить больше внимания базовым средствам языка – `await`, `async dev`, `async for` и `async with` – и показать, как они используются совместно с библиотекой *asyncio* и другими каркасами.

Часть V «Метапрограммирование»

Эта часть начинается с обзора способов построения классов с динамически создаваемыми атрибутами для обработки слабоструктурированных данных, например в формате JSON. Затем мы рассматриваем знакомый механизм свойств, после чего переходим к низкоуровневым деталям доступа к атрибутам объекта с помощью дескрипторов. Объясняется связь между функциями, методами и дескрипторами. На примере приведенной здесь пошаговой реализации библиотеки контроля полей мы вскрываем тонкие нюансы, которые делают необходимым применение рассмотренных в этой главе продвинутых инструментов: декораторов классов и метаклассов.

Практикум

Часто для исследования языка и библиотек мы будем пользоваться интерактивной оболочкой Python. Я считаю важным всячески подчеркивать удобство этого средства для обучения. Особенно это относится к читателям, привыкшим к статическим компилируемым языкам, в которых нет цикла чтения-вычисления-печати (read-eval-print#loop – REPL).

Один из стандартных пакетов тестирования для Python, `doctest` (*https://docs.python.org/3/library/doctest.html*), работает следующим образом: имитирует сеансы оболочки и проверяет, что результат вычисления выражения совпадает с заданным. Я использовал `doctest` для проверки большей части приведенного в книге кода, включая листинги сеансов оболочки. Для чтения книги ни применять, ни даже знать о пакете `doctest` не обязательно: основная характеристика doctest-скриптов (или просто тестов) состоит в том, что они выглядят как копии интерактивных сеансов оболочки Python, поэтому вы можете сами выполнить весь демонстрационный код.

Иногда я буду объяснять, чего мы хотим добиться, демонстрируя тест раньше кода, который заставляет его выполниться успешно. Если сначала отчетливо представить себе, что необходимо сделать, а только потом задумываться о том, как это сделать, то структура кода заметно улучшится. Написание тестов раньше кода – основа методологии разработки через тестирование (TDD); мне кажется, что и для преподавания это полезно. Если вы незнакомы с `doctest`, загляните в документацию (*https://docs.python.org/3/library/doctest.html*) и в репозиторий исходного кода к этой книге (*https://github.com/fluentpython/example-code-2e*).

Я также написал автономные тесты для нескольких крупных примеров, воспользовавшись модулем `pytest`, который, как мне кажется, проще и имеет больше возможностей, чем модуль `unittest` из стандартной библиотеки. Вы увидите, что для проверки правильности большей части кода в книге до-

статочно ввести команду `python3 -m doctest example_script.py` или `pytest` в оболочке ОС. Конфигурационный файл *pytest.ini* в корне репозитория исходного кода (*https://github.com/fluentpython/example-code-2e*) гарантирует, что команда `pytest` найдет и выполнит тесты.

Поговорим: мое личное мнение

Я использую, преподаю и принимаю участие в обсуждениях Python с 1998 года и обожаю изучать и сравнивать разные языки программирования, их дизайн и теоретические основания. В конце некоторых глав имеются врезки «Поговорим», где излагается моя личная точка зрения на Python и другие языки. Если вас такие обсуждения не интересуют, можете спокойно пропускать их. Приведенные в них сведения всегда факультативны.

Сопроводительный сайт: fluentpython.com

Из-за включения новых средств – аннотаций типов, классов данных, сопоставления с образцом и других – это издание оказалось почти на 30 % больше первого. Чтобы книга была подъемной, я перенес часть материалов на сайт *fluentpython.com*. В нескольких главах имеются ссылки на опубликованные там статьи. Там же есть текст нескольких выборочных глав. Полный текст доступен онлайн (*https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/*) и по подписке O'Reilly Learning *(https://www.oreilly.com/online-learning/try-now.html*). Код примеров имеется в репозитории на GitHub (*https://github.com/fluentpython/example-code-2e*).

Графические выделения

В книге применяются следующие графические выделения.

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный`

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Отмечу, что когда внутри элемента, набранного моноширинным шрифтом, оказывается разрыв строки, дефис не добавляется, поскольку он мог бы быть ошибочно принят за часть элемента.

`Моноширинный полужирный`

Команды или иной текст, который пользователь должен вводить буквально.

`Моноширинный курсив`

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

Так обозначается совет или рекомендация.

Так обозначается замечание общего характера.

Так обозначается предупреждение или предостережение.

О примерах кода

Все скрипты и большая часть приведенных в книге фрагментов кода имеются в репозитории на GitHub по адресу (*https://github.com/fluentpython/example-code-2e*).

Вопросы технического характера, а также замечания по примерам кода следует отправлять по адресу *bookquestions@oreilly.com*.

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Fluent Python, 2nd ed., by Luciano Ramalho (O'Reilly). Copyright 2022 Luciano Ramalho, 978-1-492-05635-5».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)

707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Часть I

Структуры данных

Глава 1

Модель данных в языке Python

> У Гвидо поразительное эстетическое чувство дизайна языка. Я встречал многих замечательных проектировщиков языков программирования, создававших теоретически красивые языки, которыми никто никогда не пользовался, а Гвидо – один из тех редких людей, которые могут создать язык, немного не дотягивающий до теоретической красоты, зато такой, что писать на нем программы в радость.
>
> – Джим Хагьюнин, автор Jython, соавтор AspectJ, архитектор .Net DLR[1]

Одно из лучших качеств Python – его согласованность. Немного поработав с этим языком, вы уже сможете строить обоснованные и правильные предположения о еще незнакомых средствах.

Однако тем, кто раньше учил другой объектно-ориентированный язык, может показаться странным синтаксис `len(collection)` вместо `collection.len()`. Это кажущаяся несообразность – лишь верхушка айсберга, и если ее правильно понять, то она станет ключом к тому, что мы называем «питонизмами». А сам айсберг называется моделью данных в Python и описывает API, следуя которому, можно согласовать свои объекты с самыми идиоматичными средствами языка.

Можно считать, что модель данных описывает Python как каркас. Она формализует различные структурные блоки языка, в частности последовательности, итераторы, функции, классы, контекстные менеджеры и т. д.

При программировании в любом каркасе мы тратим большую часть времени на реализацию вызываемых каркасом методов. Это справедливо и при использовании модели данных Python для построения новых классов. Интерпретатор Python вызывает специальные методы для выполнения базовых операций над объектами, часто такие вызовы происходят, когда встречается некая синтаксическая конструкция. Имена специальных методов начинаются и заканчиваются двумя знаками подчеркивания. Так, за синтаксической конструкцией `obj[key]` стоит специальный метод `__getitem__`. Для вычисления выражения `my_collection[key]` интерпретатор вызывает метод `my_collection.__getitem__(key)`.

Мы реализуем специальные методы, когда хотим, чтобы наши объекты могли поддерживать и взаимодействовать с базовыми конструкциями языка, а именно:

---

[1] История Jython (*http://hugunin.net/story_of_jython.html*), изложенная в предисловии к книге Samuele Pedroni and Noel Rappin «Jython Essentials» (O'Reilly).

- коллекции;
- доступ к атрибутам;
- итерирование (включая асинхронное итерирование с помощью `async for`);
- перегрузку операторов;
- вызов функций и методов;
- представление и форматирование строк;
- асинхронное программирование с использованием `await`;
- создание и уничтожение объектов;
- управляемые контексты (т. е. блоки `with` и `async with`).

Магические и dunder-методы

На жаргоне специальные методы называют *магическими*, но как мы в разговоре произносим имя конкретного метода, например `__getitem__`? Выражение «dunder-getitem» я услышал от автора и преподавателя Стива Холдена. «Dunder» – это сокращенная форма «двойной подчерк до и после». Поэтому специальные методы называются также *dunder-методами*. В главе «Лексический анализ» *Справочного руководства по Python* имеется предупреждение: «*Любое* использование имен вида `__*__` в любом контексте, отличающееся от явно документированного, может привести к ошибке без какого-либо предупреждения».

Что нового в этой главе

В этой главе немного отличий от первого издания, потому что она представляет собой введение в модель данных в Python, которая давно стабилизировалась. Перечислим наиболее существенные изменения:

- в таблицы в разделе «Сводка специальных методов» добавлены методы, поддерживающие асинхронное программирование и другие новые средства;
- на рис. 1.2 показано использование специальных методов API коллекций, включая абстрактный базовый класс `collections.abc.Collection`, появившийся в версии Python 3.6.

Кроме того, здесь и далее я использую синтаксис *f-строк*, введенный в Python 3.6, который проще читать и зачастую удобнее, чем прежние способы форматирования: метод `str.format()` и оператор `%`.

Использовать нотацию `my_fmt.format()` все еще необходимо, когда `my_fmt` определено не там, где выполняется операция форматирования. Например, если `my_fmt` состоит из нескольких строчек и определено в виде константы или читается из конфигурационного файла либо из базы данных. Тут по-другому не сделаешь, но такие ситуации встречаются нечасто.

Колода карт на Python

Следующий пример очень прост, однако демонстрирует выгоды от реализации двух специальных методов: `__getitem__` и `__len__`.

Пример 1.1. Колода как последовательность карт

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

Прежде всего отметим использование `collections.namedtuple` для конструирования простого класса, представляющего одну карту. Мы используем класс `namedtuple` для построения классов, содержащих только атрибуты и никаких методов, как, например, запись базы данных. В данном примере мы воспользовались им для создания простого представления игральной карты, что продемонстрировано в следующем сеансе оболочки:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

Но изюминка примера – класс `FrenchDeck`. Совсем короткий, он таит в себе немало интересного. Во-первых, как и для любой стандартной коллекции в Python, для колоды можно вызвать функцию `len()`, которая вернет количество карт в ней:

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Получить карту из колоды, например первую или последнюю, просто благодаря методу `__getitem__`:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Нужно ли создавать метод для выбора случайной карты? Необязательно. В Python уже есть функция выборки случайного элемента последовательности: `random.choice`. Достаточно вызвать ее для экземпляра колоды:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
```

```
>>> choice(deck)
Card(rank='2', suit='clubs')
```

Мы только что видели два преимущества использования специальных методов для работы с моделью данных.

- Пользователям нашего класса нет нужды запоминать нестандартные имена методов для выполнения стандартных операций («Как мне получить количество элементов? То ли `.size()`, то ли `.length()`, то ли еще как-то»).
- Проще воспользоваться богатством стандартной библиотеки Python (например, функцией `random.choice`), чем изобретать велосипед.

Но это еще не все.

Поскольку метод `__getitem__` делегирует выполнение оператору `[]` объекта `self._cards`, колода автоматически поддерживает срезы. Вот как можно посмотреть три верхние карты в неперетасованной колоде, а затем выбрать только тузы, начав с элемента, имеющего индекс 12, и пропуская по 13 карт:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Стоило нам реализовать специальный метод `__getitem__`, как колода стала допускать итерирование:

```
>>> for card in deck: # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

Итерировать можно и в обратном порядке:

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```

Многоточие в тестах

Всюду, где возможно, листинги сеансов оболочки извлекались из doctest-скриптов, чтобы гарантировать точность. Если вывод слишком длинный, то опущенная часть помечается многоточием, как в последней строке показанного выше кода. В таких случаях мы используем директиву `# doctest: +ELLIPSIS`, чтобы тест завершился успешно. Если вы будете вводить эти примеры в интерактивной оболочке, можете вообще опускать директивы doctest.

Итерирование часто подразумевается неявно. Если в коллекции отсутствует метод `__contains__`, то оператор `in` производит последовательный просмотр.

Конкретный пример – в классе `FrenchDeck` оператор `in` работает, потому что этот класс итерируемый. Проверим:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

А как насчет сортировки? Обычно карты ранжируются по достоинству (тузы – самые старшие), а затем по масти в порядке пики (старшая масть), черви, бубны и трефы (младшая масть). Приведенная ниже функция ранжирует карты, следуя этому правилу: `0` означает двойку треф, а `51` – туз пик.

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

С помощью функции `spades_high` мы теперь можем расположить колоду в порядке возрастания:

```
>>> for card in sorted(deck, key=spades_high): # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 карт опущено)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Хотя класс `FrenchDeck` неявно наследует `object`, его функциональность не наследуется, а является следствием использования модели данных и композиции. Вследствие реализации специальных методов `__len__` и `__getitem__` класс `FrenchDeck` ведет себя как стандартная последовательность и позволяет использовать базовые средства языка (например, итерирование и получение среза), а также функции `reversed` и `sorted`. Благодаря композиции реализации методов `__len__` и `__getitem__` могут перепоручать работу объекту `self._cards` класса `list`.

А как насчет тасования?

В текущей реализации объект класса `FrenchDeck` нельзя перетасовать, потому что он неизменяемый: ни карты, ни их позиции невозможно изменить, не нарушая инкапсуляцию (т. е. манипулируя атрибутом `_cards` непосредственно). В главе 13 мы исправим это, добавив однострочный метод `__setitem__`.

Как используются специальные методы

Говоря о специальных методах, нужно все время помнить, что они предназначены для вызова интерпретатором, а не вами. Вы пишете не `my_object.__len__()`, а `len(my_object)`, и если `my_object` – экземпляр определенного пользователем класса, то Python вызовет реализованный вами метод экземпляра `__len__`.

Однако для встроенных классов, например `list`, `str`, `bytearray`, или расширений типа массивов NumPy интерпретатор поступает проще. Коллекции переменного размера, написанные на C, включают структуру[1] `PyVarObject`, в которой имеется поле `ob_size`, содержащее число элементов в коллекции. Поэтому если `my_object` – экземпляр одного из таких встроенных типов, то `len(my_object)` возвращает значение поля `ob_size`, что гораздо быстрее, чем вызов метода.

Как правило, специальный метод вызывается неявно. Например, предложение `for i in x:` подразумевает вызов функции `iter(x)`, которая, в свою очередь, может вызывать метод `x.__iter__()`, если он реализован, или использовать `x.__getitem__()`, как в примере класса `FrenchDeck`.

Обычно в вашей программе не должно быть много прямых обращений к специальным методам. Если вы не пользуетесь метапрограммированием, то чаще будете реализовывать специальные методы, чем явно вызывать их. Единственный специальный метод, который регулярно вызывается из пользовательского кода напрямую, – `__init__`, он служит для инициализации суперкласса из вашей реализации `__init__`.

Если необходимо обратиться к специальному методу, то обычно лучше вызвать соответствующую встроенную функцию (например, `len`, `iter`, `str` и т. д.). Она вызывает нужный специальный метод и нередко предоставляет дополнительный сервис. К тому же для встроенных типов это быстрее, чем вызов метода. См. раздел «Использование iter совместно с вызываемым объектом» главы 17.

В следующих разделах мы рассмотрим некоторые из наиболее важных применений специальных методов:

- эмуляция числовых типов;
- строковое представление объектов;
- булево значение объекта;
- реализация коллекций.

Эмуляция числовых типов

Несколько специальных методов позволяют объектам иметь операторы, например `+`. Подробно мы рассмотрим этот вопрос в главе 16, а пока проиллюстрируем использование таких методов на еще одном простом примере.

Мы реализуем класс для представления двумерных векторов, обычных евклидовых векторов, применяемых в математике и физике (рис. 1.1).

Для представления двумерных векторов можно использовать встроенный класс `complex`, но наш класс допускает обобщение на *n*-мерные векторы. Мы займемся этим в главе 17.

[1] В языке C структура – это тип записи с именованными полями.

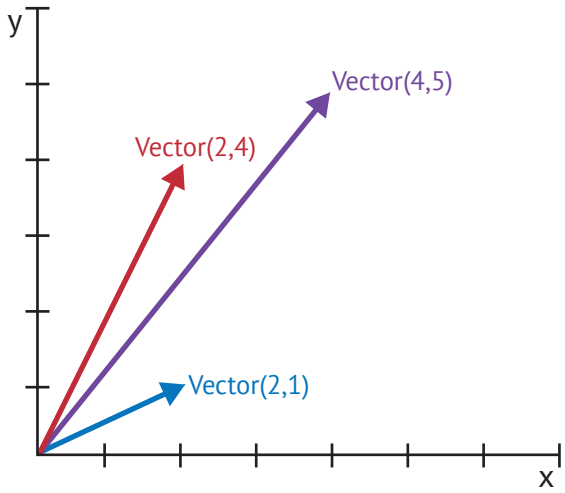

Рис. 1.1. Пример сложения двумерных векторов: Vector(2, 4) + Vector(2, 1) = Vector(4, 5)

Для начала спроектируем API класса, написав имитацию сеанса оболочки, которая впоследствии станет тестом. В следующем фрагменте тестируется сложение векторов, изображенное на рис. 1.1.

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Отметим, что оператор `+` порождает результат типа `Vector`, который отображается в оболочке интуитивно понятным образом.

Встроенная функция `abs` возвращает абсолютную величину вещественного числа – целого или с плавающей точкой – и модуль числа типа `complex`, поэтому для единообразия наш API также использует функцию `abs` для вычисления модуля вектора:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

Мы можем еще реализовать оператор `*`, выполняющий умножение на скаляр (т. е. умножение вектора на число, в результате которого получается новый вектор с тем же направлением и умноженным на данное число модулем):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

В примере 1.2 приведен класс `Vector`, реализующий описанные операции с помощью специальных методов `__repr__`, `__abs__`, `__add__` и `__mul__`.

Пример 1.2. Простой класс двумерного вектора

```
«»»
vector2d.py: упрощенный класс, демонстрирующий некоторые специальные методы.

Упрощен из дидактических соображений. Классу не хватает правильной
обработки ошибок, особенно в методах ``__add__`` and ``__mul__``.

Далее в книге этот пример будет существенно расширен.

Сложение::

    >>> v1 = Vector(2, 4)
    >>> v2 = Vector(2, 1)
    >>> v1 + v2
    Vector(4, 5)

Абсолютная величина::

    >>> v = Vector(3, 4)
    >>> abs(v)
    5.0

Умножение на скаляр::

    >>> v * 3
    Vector(9, 12)
    >>> abs(v * 3)
    15.0

"""

import math
class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

Мы реализовали пять специальных методов, помимо хорошо знакомого `__init__`. Отметим, что ни один из них не вызывается напрямую внутри само-

го класса или при типичном использовании класса, показанном в листингах сеансов оболочки. Как уже было сказано, чаще всего специальные методы вызывает интерпретатор Python.

В примере 1.2 реализовано два оператора: `+` и `*`, чтобы продемонстрировать использование методов `__add__` и `__mul__`. В обоих случаях метод создает и возвращает новый экземпляр класса `Vector`, не модифицируя ни один операнд – аргументы `self` и `other` только читаются. Именно такого поведения ожидают от инфиксных операторов: создавать новые объекты, не трогая операндов. Мы еще вернемся к этому вопросу в главе 16.

Реализация в примере 1.2 позволяет умножать `Vector` на число, но не число на `Vector`. Это нарушает свойство коммутативности операции умножения на скаляр. В главе 16 мы исправим данный недостаток, реализовав специальный метод `__rmul__`.

В следующих разделах мы обсудим другие специальные методы класса `Vector`.

Строковое представление

Специальный метод `__repr__` вызывается встроенной функцией `repr` для получения строкового представления объекта. Если бы мы не реализовали метод `__repr__`, то объект класса `Vector` был бы представлен в оболочке строкой вида `<Vector object at 0x10e100070>`.

Интерактивная оболочка и отладчик вызывают функцию `repr`, передавая ей результат вычисления выражения. То же самое происходит при обработке спецификатора `%r` в случае классического форматирования с помощью оператора `%` и при обработке поля преобразования `!r` в новом синтаксисе форматной строки (*https://docs.python.org/3.10/library/string.html#format-string-syntax*), применяемом в *f-строках* в методе `str.format`.

Отметим, что в *f-строке* в нашей реализации метода `__repr__` мы использовали `!r` для получения стандартного представления отображаемых атрибутов. Это разумный подход, потому что в нем отчетливо проявляется существенное различие между `Vector(1, 2)` и `Vector('1', '2')` – второй вариант в контексте этого примера не заработал бы, потому что аргументами конструктора должны быть числа, а не объекты `str`.

Строка, возвращаемая методом `__repr__`, должна быть однозначно определена и по возможности соответствовать коду, необходимому для восстановления объекта. Именно поэтому мы выбрали представление, напоминающее вызов конструктора класса (например, `Vector(3, 4)`).

В отличие от `__repr__`, метод `__str__` вызывается конструктором `str()` и неявно используется в функции `print`. Он должен возвращать строку, пригодную для показа пользователям.

Иногда строка, возвращенная методом `__repr__`, уже пригодна для показа пользователям, тогда нет нужды писать метод `__str__`, т. к. реализация, унаследованная от класса `object`, вызывает `__repr__`, если нет альтернативы. Пример 5.2 – один из немногих в этой книге, где используется пользовательский метод `__str__`.

Программисты, имеющие опыт работы с языками, где имеется метод `toString`, по привычке реализуют метод `__str__`, а не `__repr__`. Если вы реализуете только один из этих двух методов, то пусть это будет `__repr__`.

На сайте Stack Overflow был задан вопрос «What is the difference between __str__ and __repr__ in Python» (*https://stackoverflow.com/questions/1436703/what-is-the-difference-between-str-and-repr*), ответ на который содержит прекрасные разъяснения Алекса Мартелли и Мартина Питерса.

Булево значение пользовательского типа

Хотя в Python есть тип `bool`, интерпретатор принимает любой объект в булевом контексте, например в условии `if`, в управляющем выражении цикла `while` или в качестве операнда операторов `and`, `or` и `not`. Чтобы определить, является ли выражение истинным или ложным, применяется функция `bool(x)`, которая возвращает `True` или `False`.

По умолчанию любой экземпляр пользовательского класса считается истинным, но положение меняется, если реализован хотя бы один из методов `__bool__` или `__len__`. Функция `bool(x)`, по существу, вызывает `x.__bool__()` и использует полученный результат. Если метод `__bool__` не реализован, то Python пытается вызвать `x.__len__()` и при получении нуля функция `bool` возвращает `False`. В противном случае `bool` возвращает `True`.

Наша реализация `__bool__` концептуально проста: метод возвращает `False`, если модуль вектора равен 0, и `True` в противном случае. Для преобразования модуля в булеву величину мы вызываем `bool(abs(self))`, поскольку ожидается, что метод `__bool__` возвращает булево значение. Вне метода `__bool__` редко возникает надобность вызывать `bool()` явно, потому что любой объект можно использовать в булевом контексте.

Обратите внимание на то, как специальный метод `__bool__` обеспечивает согласованность пользовательских объектов с правилами проверки значения истинности, определенными в главе «Встроенные типы» документации по стандартной библиотеке Python (*http://docs.python.org/3/library/stdtypes.html#truth*).

Можно было бы написать более быструю реализацию метода `Vector.__bool__`:

```
    def __bool__(self):
        return bool(self.x or self.y)
```

Она сложнее воспринимается, зато позволяет избежать обращений к `abs` и `__abs__`, возведения в квадрат и извлечения корня. Явное преобразование в тип `bool` необходимо, потому что метод `__bool__` должен возвращать булево значение, а оператор `or` возвращает один из двух операндов: результат вычисления `x or y` равен `x`, если `x` истинно, иначе равен `y` вне зависимости от его значения.

API коллекций

На рис. 1.2 описаны интерфейсы основных типов коллекций, поддерживаемых языком. Все классы на этой диаграмме являются *абстрактными базо-*

выми классами, ABC. Такие классы и модуль `collections.abc` рассматриваются в главе 13. Цель этого краткого раздела – дать общее представление о самых важных интерфейсах коллекций в Python и показать, как они строятся с помощью специальных методов.

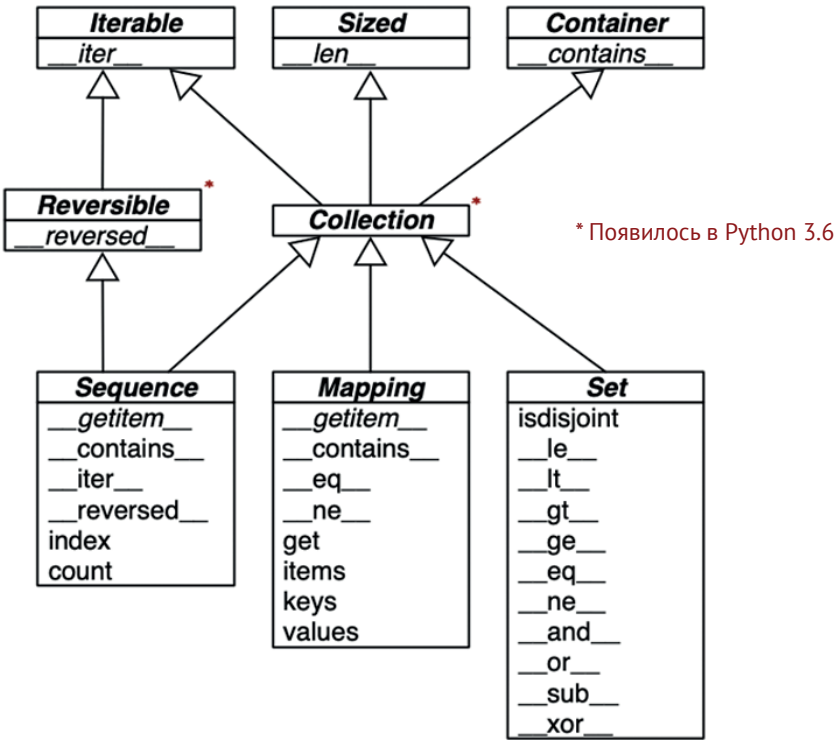

Рис. 1.2. UML-диаграмма классов, иллюстрирующая наиболее важные типы коллекций. Методы, имена которых набраны курсивом, абстрактные, поэтому должны быть реализованы в конкретных подклассах, например `list` и `dict`. У остальных методов имеются конкретные реализации, которые подклассы могут унаследовать

У каждого из ABC в верхнем ряду есть всего один специальный метод. Абстрактный базовый класс `Collection` (появился в версии Python 3.6) унифицирует все три основных интерфейса, который должна реализовать любая коллекция:

- `Iterable` для поддержки `for`, распаковки и других видов итерирования;
- `Sized` для поддержки встроенной функции `len`;
- `Container` для поддержки оператора `in`.

Python не требует, чтобы конкретные классы наследовали какому-то из этих ABC. Любой класс, реализующий метод `__len__`, удовлетворяет требованиям интерфейса `Sized`.

Перечислим три важнейшие специализации `Collection`:

- `Sequence`, формализует интерфейс встроенных классов, в частности `list` и `str`;
- `Mapping`, реализован классами `dict`, `collections.defaultdict` и др.;
- `Set`, интерфейс встроенных типов `set` и `frozenset`.

Только `Sequence` реализует интерфейс `Reversible`, потому что последовательности поддерживают произвольное упорядочение элементов, тогда как отображения и множества таким свойством не обладают.

Начиная с версии Python 3.7 тип `dict` официально считается «упорядоченным», но это лишь означает, что порядок вставки ключей сохраняется. Переупорядочить ключи словаря `dict` так, как вам хочется, невозможно.

Все специальные методы ABC `Set` предназначены для реализации инфиксных операторов. Например, выражение `a & b` вычисляет пересечение множеств `a` и `b` и реализовано специальным методом `__and__`.

В следующих двух главах мы подробно рассмотрим стандартные библиотечные последовательности, отображения и множества.

А пока перейдем к основным категориям специальных методов, определенным в модели данных Python.

Сводка специальных методов

В главе «Модель данных» (*http://docs.python.org/3/reference/datamodel.html*) справочного руководства по языку Python перечислено более 80 специальных методов. Больше половины из них используются для реализации операторов: арифметических, поразрядных и сравнения. Следующие таблицы дают представление о том, что имеется в нашем распоряжении.

В табл. 1.1 показаны имена специальных методов, за исключением тех, что используются для реализации инфиксных операторов и базовых математических функций, например `abs`. Большинство этих методов будут рассмотрены на протяжении книги, в т. ч. недавние добавления: асинхронные специальные методы, в частности `__anext__` (добавлен в Python 3.5), и точка подключения для настройки класса `__init_subclass__` (добавлен в Python 3.6).

Таблица 1.1. Имена специальных методов (операторы не включены)

Категория	Имена методов
Представление в виде строк и байтов	`__repr__`, `__str__`, `__format__`, `__bytes__`, `__fspath__`
Преобразование в число	`__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Эмуляция коллекций	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Итерирование	`__iter__`, `__aiter__`, `__next__`, `__anext__`, `__reversed__`
Выполнение объектов, допускающих вызов, или сопрограмм	`__call__`, `__await__`
Управление контекстом	`__enter__`, `__exit__`, `__aenter__`, `__aexit__`
Создание и уничтожение объектов	`__new__`, `__init__`, `__del__`

Окончание табл. 1.1

Категория	Имена методов
Управление атрибутами	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
Дескрипторы атрибутов	`__get__`, `__set__`, `__delete__`, `__set_name__`
Абстрактные базовые классы	`__instancecheck__`, `__subclasscheck__`
Метапрограммирование классов	`__prepare__`, `_init_subclass__`, `__class_getitem__`, `__mro_entries__`

Инфиксные и числовые операторы поддерживаются специальными методами, перечисленными в табл. 1.2. В версии Python 3.5 были добавлены методы `__matmul__`, `__rmatmul__` и `__imatmul__` для поддержки `@` в роли инфиксного оператора умножения матриц (см. главу 16).

Таблица 1.2. Имена специальных методов для операторов

Категория операторов	Символы	Имена методов
Унарные числовые операторы	`- + abs()`	`__neg__ __pos__ __abs__`
Операторы сравнения	`< <= == !- > >=`	`__lt__ __le__ __eq__ __ne__ __gt__ __ge__`
Арифметические операторы	`+ - * / // % @ divmod() round() ** pow()`	`__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__`
Инверсные арифметические операторы	(арифметические операторы с переставленными операндами)	`__radd__ __rsub__ __rmul__ __rtruediv__ __rfloordiv__ __rmod__ __rmatmul__ __rdivmod__ __rpow__`
Арифметические операторы составного присваивания	`+= -= *= /= //= %= @= **=`	`__iadd__ __isub__ __imul__ __itruediv____ifloordiv__ __imod__ __imatmul__ __ipow__`
Поразрядные операторы	`& | ^ << >> ~`	`__and__ __or__ __xor__ __lshift__ __rshift__ __invert__`
Инверсные поразрядные операторы	(поразрядные операторы с переставленными операндами)	`__rand__ __ror__ __rxor__ __rlshift__ __rrshift__`
Поразрядные операторы составного присваивания	`&= |= ^= <<= >>=`	`__iand__ __ior__ __ixor__ __ilshift__ __irshift__`

Python вызывает инверсный специальный метод от имени второго операнда, если нельзя использовать соответственный специальный метод от имени первого операнда. Операторы составного присваивания – сокращенный способ вызвать инфиксный оператор с последующим присваиванием переменной, например `a += b`.

В главе 16 инверсные операторы и составное присваивание рассматриваются подробнее.

Почему len – не метод

Я задавал этот вопрос разработчику ядра Раймонду Хэттингеру в 2013 году, смысл его ответа содержится в цитате из «Дзен Python»: «практичность важнее чистоты» (*https://www.python.org/doc/humor/#thezen-of-python*). В разделе «Как используются специальные методы» выше я писал, что функция `len(x)` работает очень быстро, если `x` – объект встроенного типа. Для встроенных объектов интерпретатор CPython вообще не вызывает никаких методов: длина просто читается из поля C-структуры. Получение количества элементов в коллекции – распространенная операция, которая должна работать эффективно для таких разных типов, как `str`, `list`, `memoryview` и т. п.

Иначе говоря, `len` не вызывается как метод, потому что играет особую роль в модели данных Python, равно как и `abs`. Но благодаря специальному методу `__len__` можно заставить функцию `len` работать и для пользовательских объектов. Это разумный компромисс между желанием обеспечить как эффективность встроенных объектов, так и согласованность языка. Вот еще цитата из «Дзен Python»: «особые случаи не настолько особые, чтобы из-за них нарушать правила».

Если рассматривать `abs` и `len` как унарные операторы, то, возможно, вы простите их сходство с функциями, а не с вызовами метода, чего следовало бы ожидать от ОО-языка. На самом деле в языке ABC – непосредственном предшественнике Python, в котором впервые были реализованы многие его средства, – существовал оператор `#`, эквивалентный `len` (следовало писать `#s`). При использовании в качестве инфиксного оператора – `x#s` – он подсчитывал количество вхождений `x` и `s`; в Python для этого нужно вызвать `s.count(x)`, где `s` – произвольная последовательность.

Резюме

Благодаря реализации специальных методов пользовательские объекты могут вести себя как встроенные типы. Это позволяет добиться выразительного стиля кодирования, который сообщество считает «питоническим».

Важное требование к объекту Python – обеспечить полезные строковые представления себя: одно – для отладки и протоколирования, другое – для показа пользователям. Именно для этой цели предназначены специальные методы `__repr__` и `__str__`.

Эмуляция последовательностей, продемонстрированная на примере класса `FrenchDeck`, – одно из самых распространенных применений специальных методов. Устройство большинства типов последовательностей – тема главы 2, а реализация собственных последовательностей будет рассмотрена в главе 12 в контексте создания многомерного обобщения класса `Vector`.

Благодаря перегрузке операторов Python предлагает богатый набор числовых типов, от встроенных до `decimal.Decimal` и `fractions.Fraction`, причем все они поддерживают инфиксные арифметические операторы. Библиотека анализа данных NumPy поддерживает инфиксные операторы для матриц и тензоров. Реализация операторов, в том числе инверсных и составного присваивания, будет продемонстрирована в главе 16 в процессе расширения класса `Vector`.

Использование и реализация большинства других специальных методов, входящих в состав модели данных Python, рассматривается в разных частях книги.

Дополнительная литература

Глава «Модель данных» (*http://docs.python.org/3/reference/datamodel.html*) справочного руководства по языку Python – канонический источник информации по теме этой главы и значительной части изложенного в книге материала.

В книге Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», третье издание (O'Reilly), прекрасно объясняется модель данных. Данное ими описание механизма доступа к атрибутам – самое полное из всех, что я видел, если не считать самого исходного кода CPython на C. Мартелли также очень активен на сайте Stack Overflow, ему принадлежат более 6200 ответов. С его профилем можно ознакомиться по адресу *http://stackoverflow.com/users/95810/alex-martelli*.

Дэвид Бизли написал две книги, в которых подробно описывается модель данных в контексте Python 3: «Python Essential Reference», издание 4 (Addison-Wesley Professional), и «*Python Cookbook*»[1], издание 3 (O'Reilly), в соавторстве с Брайаном Л. Джонсом.

В книге Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow «The Art of the Metaobject Protocol» (MIT Press) объясняется протокол метаобъектов, одним из примеров которого является модель данных в Python.

Поговорим

Модель данных или объектная модель?

То, что в документации по Python называется «моделью данных», большинство авторов назвали бы «объектной моделью Python». В книгах Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», издание 3, и David Beazley «Python Essential Reference», издание 4, – лучших книгах по «модели данных Python» – употребляется термин «объектная модель». В Википедии самое первое определение модели данных (*http://en.wikipedia.org/wiki/Object_model*) звучит так: «Общие свойства объектов в конкретном языке программирования». Именно в этом и заключается смысл «модели данных Python». В этой книге я употребляю термин «модель данных», потому что его предпочитают авторы документации и потому что так называется глава в справочном руководстве по языку Python (*https://docs.python.org/3/reference/datamodel.html*), имеющая прямое касательство к нашему обсуждению.

Магические методы

В словаре «The Original Hacker's Dictionary» (*https://www.dourish.com/goodies/jargon.html*) термин *магический* определяется как «еще не объясненный или слишком сложный для объяснения» или как «не раскрываемый публично механизм, позволяющий делать то, что иначе было бы невозможно».

В сообществе Ruby эквиваленты специальных методов называют магическими. Многие пользователи из сообщества Python также восприняли этот термин. Лично я считаю, что специальные методы – прямая противоположность магии. В этом отношении языки Python и Ruby одинаковы: тот и другой предоставляют развитый протокол метаобъектов, отнюдь не магический, но позволяющий пользователям применять те же средства, что доступны разра-

[1] *Бизли Д., Джонс Б. К.* Python. Книга рецептов. М.: ДМК Пресс, 2019 // *https://dmkpress.com/catalog/computer/programming/python/978-5-97060-751-0/*

ботчикам ядра, которые пишут интерпретаторы этих языков.

Сравним это с Go. В этом языке у некоторых объектов есть действительно магические возможности, т. е. такие, которые невозможно имитировать в пользовательских типах. Например, массивы, строки и отображения в Go поддерживают использование квадратных скобок для доступа к элементам: `a[i]`. Но не существует способа приспособить нотацию `[]` к новым, определенным пользователем типам. Хуже того, в Go нет ни понятия интерфейса итерируемости, ни объекта итератора на пользовательском уровне, поэтому синтаксическая конструкция `for/range` ограничена поддержкой пяти «магических» встроенных типов, в т. ч. массивов, строк и отображений.

Быть может, в будущем проектировщики Go расширят протокол метаобъектов. А пока в нем куда больше ограничений, чем в Python и Ruby.

Метаобъекты

«The Art of the Metaobject Protocol» (AMOP) – моя любимая книга по компьютерам. Но и отбросив в сторону субъективизм, термин «протокол метаобъектов» полезен для размышления о модели данных в Python и о похожих средствах в других языках. Слово «метаобъект» относится к объектам, являющимся структурными элементами самого языка. А «протокол» в этом контексте – синоним слова «интерфейс». Таким образом, протокол метаобъектов – это причудливый синоним «объектной модели»: API для доступа к базовым конструкциям языка.

Развитый протокол метаобъектов позволяет расширять язык для поддержки новых парадигм программирования. Грегор Кикзалес, первый автор книги AMOP, впоследствии стал первопроходцем аспектно-ориентированного программирования и первоначальным автором AspectJ, расширения Java для реализации этой парадигмы. В динамическом языке типа Python реализовать аспектно-ориентированное программирование гораздо проще, и существует несколько каркасов, в которых это сделано. Самым известным из них является каркас *zope.interface* (*http://docs.zope.org/zope.interface/*), на базе которого построена система управления контентом Plone (*https://plone.org/*).