ОГЛАВЛЕНИЕ

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>.[1]

– Тим Питерс,
легендарный разработчик ядра и автор сборника поучений «The Zen of Python»

«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 паке-

[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*).

тах, 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. Если вы уже знакомы с Python и хотели бы перейти на версию Python 3.4 или старше, эта книга для вас. Когда я писал ее, большинство профессиональных программистов работали с Python 2, поэтому я специально выделял особенности Python 3, которые для этой аудитории могли оказаться внове.

Однако поскольку книга посвящена, главным образом, тому, как получить максимум от Python 3.4, я не останавливаюсь на исправлениях, которые нужно внести в старый код, чтобы он продолжал работать. Большинство примеров будут работать в Python 2.7 с минимальными изменениями или вообще без оных, но иногда обратный перенос требует значительных усилий.

И все же я полагаю, что эта книга может быть полезна и тем, кто вынужден продолжать писать на Python 2.7, поскольку базовые концепции остались теми же самыми. Python 3 – не новый язык, и большинство различий можно изучить за полдня. Желающие узнать, что нового появилось в Python 3.0, могут начать со страницы *https://docs.python.org/3.0/whatsnew/3.0.html*. Разумеется, с момента выхода версии 3.0 в 2009 году Python не стоял на месте, но все последующие изменения не так существенны, как внесенные в 3.0.

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, появившихся в Python 3.

На кого эта книга не рассчитана

Если вы только начинаете изучать Python, это 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.

Как организована эта книга

Читатели, на которых рассчитана эта книги, без труда смогут начать чтение с любой главы. Но каждая из шести частей образует книгу в книге. Я предполагал, что главы, составляющие одну часть, будут читаться по порядку.

Я старался сначала рассказывать о том, что уже есть, а лишь затем – о том, как создавать что-то свое. Например, в главе 2 из 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 IV, 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, узнав, почему я выбрал такой путь, вам будет проще с этим смириться.

Ниже описаны основные темы, рассматриваемые в каждой части книги.

Часть I

Содержит всего одну главу, посвященную модели данных в Python, где объясняется ключевая роль специальных методов (например, `__repr__`) для обеспечения единообразного поведения объектов любого типа – в языке, заслуженно считающемся образцом единообразия. Осмысление различных граней модели данных – сквозная тема книги, но именно в главе 1 дается общий обзор.

Часть II

В главах из этой части рассматриваются типы коллекций: последовательности, отображения и множества, а также сравниваются типы `str` и `bytes`. Это вещи, которые радостно приветствовали пользователи Python 3 и которых отчаянно не хватает пользователям Python 2, еще не модернизировавшим свой код. Основная цель – напомнить, что уже имеется, и объяснить некоторые особенности поведения, которые могут оказаться неожиданными, например, изменение порядка ключей словаря `dict` в то время, когда в нем никто ничего не ищет, или подводные камни, связанные с зависящей от локали сортировкой строки Unicode. Во имя достижения этой цели изложение временами становится широким и высокоуровневым (например, во время знакомства с многочисленными типами последовательностей и отображений), а временами – углубленным (например, при описании деталей хэш-таблиц, лежащих в основе типов `dict` и `set`).

Часть III

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`.

Часть IV

Теперь наше внимание перемещается на создание классов. В части II несколько раз встречалось объявление `class`, а в части IV представлены многочисленные классы. 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 программирование на основе классов, а какие-то – нет. В главах из этой части объясняется, как работает механизм ссылок, что на самом деле означает изменчивость, как устроен жизненный цикл объектов, как создать свою коллекцию или абстрактный базовый класс, как справиться с множественным наследованием и как реализовать перегрузку операторов (если это имеет смысл).

Часть V

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 конструкцию `yield from`. Часть V заканчивается высокоуровневым введением в современные средства распараллеливания, реализованные в Python в виде модуля `collections.futures` (потоки и процессы, представленные под маской будущих объектов), и событийно-ориентированного ввода-вывода посредством `asyncio` (будущие объекты, надстроенные над сопрограммами и `yield from`).

Часть VI

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, привыкших 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 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 doctest-скрипт 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*). Вы обнаружите, что для проверки правильности большей части кода в книге достаточно ввести команду `python3 -m doctest example_script.py` в оболочке ОС.

Как производился хронометраж

В этой книге иногда приводятся результаты простого хронометража. Тесты производились на одном из двух ноутбуков, которыми я пользовался для написания книги: MacBook Pro 13" 2011 года с процессором Intel Core i7 2.7 ГГц, памятью 8 ГБ и вращающимся жестким диском MacBook Air 13" 2014 года с процессором Intel Core i5 1.4 ГГц, памятью 4 ГБ и SSD-диском. MacBook Air оснащен менее быстрым процессором и располагает меньшим объемом оперативной памяти, зато эта память быстрее (1600 МГц против 1333 МГц), а SSD-диск гораздо быстрее вращающегося. Не могу сказать, какая машина быстрее для повседневной работы.

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.

Терминология Python

Я стремился написать книгу не только о самом языке Python, но и о сложившейся вокруг него культуре. За 20 лет существования сообщество пользователей Python выработало собственный профессиональный жаргон и акронимы. В конце книги есть раздел «Терминология Python», в котором перечислены термины, имеющие специальный смысл для питонистов.

Использованная версия Python

Весь приведенный в книге код тестировался для версии Python 3.4, точнее CPython 3.4, – самой распространенной реализации Python, написанной на C. С одним исключением: в разделе «Новый инфиксный оператор @ в Python 3.5» описывается оператор `@`, который поддерживается только в версии Python 3.5.

Почти весь код должен работать для любого интерпретатора, совместимого с Python 3.x, в том числе PyPy3 2.4.0, совместимого с Python 3.2.5. Из существенных исключений упомяну конструкцию `yield from` и модуль `asyncio`, появившиеся только в версии 3.3.

По большей части, код должен работать и в Python 2.7 с мелкими изменениями за исключением примеров в главе 4, относящихся к Unicode, и уже упомянутой функциональности, отсутствующей в версиях младше 3.3.

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*).

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 by Luciano Ramalho (O'Reilly). Copyright 2015 Luciano Ramalho, 978-1-491-94600-8».

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: *http://bit.ly/fluent-python*.

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

Пролог

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 недотягивающий до теоретической красоты, зато такой, что писать на нем программы в радость.[1]
>
> – Джим Хагьюнин,
> *автор Jython, соавторAspectJ, архитектор .Net DLR*

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 вы тратите большую часть времени на реализацию вызываемых каркасом методов. То же самое справедливо для модели данных Python. 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 (например, `__getitem__`). Так, за синтаксической конструкцией `obj[key]` стоит специальный

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

метод `__getitem__`. Для вычисления выражения `my_collection[key]` интерпретатор вызывает метод `my_collection.__getitem__(key)`.

Благодаря специальным методам объекты могут реализовывать, поддерживать и взаимодействовать с базовыми конструкциями языка, а именно:

- итерирование;
- коллекции;
- доступ к атрибутам;
- перегрузка операторов;
- вызов функций и методов;
- создание и уничтожение объектов;
- представление и форматирование строк;
- управляемые контексты (т. е. блоки `with`).

Магические и dunder-методы

На жаргоне специальные методы обычно называют магическими, но в применении к конкретным методам, например `__getitem__`, некоторые разработчики говорят «подчерк-подчерк-getitem» (under-under-getitem), внося тем самым двусмысленность, потому что у конструкции `__x` имеется другой специальный смысл[2]. Произносить правильно – «подчерк-подчерк-getitem-подчерк-подчерк» – утомительно, поэтому я, следуя своему учителю Стиву Холдену, говорю «dunder-getitem». Все опытные питонисты понимают это сокращение. По этой причине специальные методы иногда называют также dunder-методами[3].

Колода карт на Python

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

В примере 1.1 приведен класс, представляющий колоду игральных карт.

Пример 1.1. Колода как последовательность карт

```
import collections

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

class FrenchDeck:
```

[2] См. раздел «Закрытые и защищенные методы в Python» ниже.

[3] Лично я впервые услышал слово «dunder» от Стива Холдена. Википедия (*http://bit.ly/1Vm72Mf*) приписывает авторство Марку Джонсону и Тиму Хохбергу, которые первыми употребили это слово в письменном ответе на вопрос «Как произнести __ (двойное подчеркивание)?» в списке рассылки python-list 26 сентября 2002 года; ответ Джонсона см. по адресу *https://mail.python.org/pipermail/python-list/2002-September/112991.html*.

```
    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. Начиная с версии Python 2.6, класс `namedtuple` можно использовать 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
```

Получение карты из колоды, например первой или последней, не должно быть сложнее обращения `deck[0]` или `deck[-1]`, и именно это обеспечивает метод `__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 от использования специальных методов для работы с моделью данных.

- Пользователям вашего 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 doctest-скриптах

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 doctest-скрипт 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 21 – туза 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`[4], 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 11 мы исправим это, добавив однострочный метод `__setitem__`.

[4] В Python 2 необходимо было бы явно написать `FrenchDeck(object)`, а в Python 3 это подразумевается по умолчанию.

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` и т. д., интерпретатор поступает проще: реализация функции `len()` в CPython возвращает значение поля `ob_size` C-структуры `PyVarObject`, которой представляется любой встроенный объект в памяти. Это 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.

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, которые 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 «Познакомимся с функцией iter поближе» главы 14.

Старайтесь не создавать собственные атрибуты с именами вида `__foo__`, потому что в будущем подобные имена могут получить специальный смысл, даже если в текущей версии это не так.

Эмуляция числовых типов

Несколько специальных методов позволяют объектам иметь операторы, например `+`. Подробно мы рассмотрим этот вопрос в главе 13, 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 14.

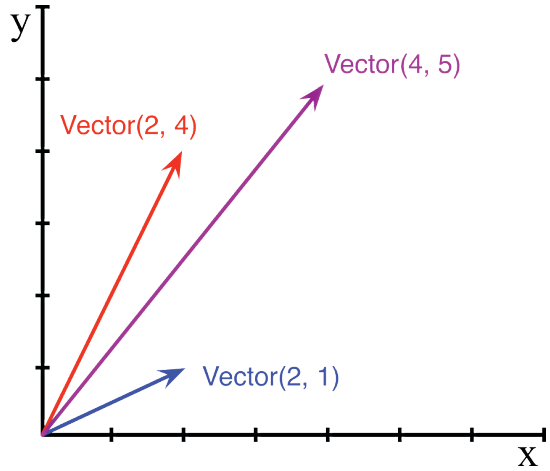

Рис. 1.1. Пример сложения двумерных векторов: Vector(2, 4) + Vector(2, 1) = Vector(4, 5)

Для начала спроектируем API класса, написав имитацию сеанса оболочки, которая впоследствии станет doctest-скриптом. 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 абсолютное значение 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 также 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

```
from math import hypot

class Vector:

   def __init__(self, x=0, y=0):
      self.x = x
      self.y = y

   def __repr__(self):
      return 'Vector(%r, %r)' % (self.x, self.y)

   def __abs__(self):
      return 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)
```

Отметим, что ни один из реализованных нами специальных методов (кроме `__init__`) не вызывается напрямую внутри самого класса или при типичном использовании класса, показанном в листингах сеансов оболочки. Как уже было сказано, чаще всего специальные методы вызывает интерпретатор Python. В следующих разделах мы обсудим, как писать каждый специальный метод.

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 (*http://bit.ly/1Vm7gD1*), применяемом в методе `str.format`.

Я в этой книге использую оператор `%` и метод `str.format`, как и большая часть сообщества Python. Мне очень нравится более мощный метод `str.format`, но я знаю, что многие питонисты предпочитают более простой оператор `%`, так что в обозримом будущем мы, скорее всего, будем встречать в написанных на Python программах оба варианта.

Отметим, что 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 строки.

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`. Метод `__str__` должен возвращать строку, пригодную для показа пользователям.

Если вы реализуете только один из этих двух методов, то пусть это будет `__repr__`, потому что в отсутствие пользовательского метода `__str__` интерпретатор Python вызывает `__repr__`.

На сайте Stack Overflow был задан вопрос «Difference between __str__ and __repr__ in Python» (http://bit.ly/1Vm7j1N), ответ на который содержит прекрасные разъяснения Алекса Мартелли и Мартина Питерса.

Арифметические операторы

В примере 1.2 реализованы два оператора: `+` и `*`, чтобы продемонстрировать принципы работы методов `__add__` и `__mul__`. Отметим, что оба метода создают и возвращают новый экземпляр `Vector`, не модифицируя ни один операнд, – аргументы `self` и `other` только читаются. Это ожидаемое поведение инфиксных операторов: создавать новые объекты, не трогая операнды. Я еще вернусь к этому вопросу в главе 13.

В примере 1.2 реализовано умножение объекта `Vector` на число, но не числа на объект `Vector`, что нарушает свойство коммутативности умножения. В главе 13 мы исправим этот недочет с помощью специального метода `__rmul__`.