Уэс Маккини

Python и анализ данных

SECOND EDITION

Python for Data Analysis

Data Wrangling with Pandas, NumPy, and IPython

Wes McKinney

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

ВТОРОЕ ИЗДАНИЕ

Python и анализ данных

Первичная обработка данных с применением pandas, NumPy и IPython

Уэс Маккини

ЛАНЬ®

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Москва, 2020

УДК 004.438Python:004.6
ББК 32.973.22
М15

Маккини У.

М15 Python и анализ данных / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 540 с.: ил.

ISBN 978-5-94074-590-5

Второе издание этой книги дает современное практическое введение в разработку научных приложений на Python, ориентированных на обработку данных. Код переписан под версию Python 3.6, добавлены сведения о последних версиях библиотек pandas, NumPy, IPython и Jupyter.

Описаны те части языка Python и библиотеки для него, которые необходимы для эффективного решения широкого круга аналитических задач: интерактивная оболочка IPython и Jupyter-блокноты, библиотеки NumPy и pandas, библиотека для визуализации данных matplotlib и др.

Издание подойдет как аналитикам, только начинающим осваивать обработку данных, так и опытным программистам на Python, еще не знакомым с научными приложениями.

УДК 004.438Python:004.6
ББК 32.973.22

ISBN 978-1-491-95766-0 (анг.)
ISBN 978-5-94074-590-5 (рус.)

Содержание

ЛАНЬ®

Предисловие

Что нового во втором издании?

Первое издание этой книги вышло в 2012 году, когда Python-библиотеки для анализа данных с открытым исходным кодом (в частности, pandas) были еще внове и быстро развивались. В этом исправленном и дополненном издании я переработал главы, стремясь отразить как несовместимые изменения и устаревшие возможности, так и новые средства, появившиеся за прошедшие пять лет. Я также добавил новый материал с описанием инструментов, которые либо еще не существовали в 2012 году, либо были недостаточно зрелыми. Наконец, я старался не писать о новых или активно разрабатываемых проектах с открытым кодом, которым, возможно, не суждено дожить до зрелости. Хотелось бы представить читателям этого издания средства, которые не утратят актуальности ни в 2020, ни в 2021 году.

Ниже перечислены основные отличия второго издания.

- Весь код, включая пособие по Python, обновлен и доведен до уровня версии 3.6 (в первом издании использовалась версия Python 2.7).
- Обновлены инструкции по установке Python из дистрибутива Anaconda Python Distribution, а также всех дополнительных Python-пакетов.
- Внесены исправления, соответствующие последним версиям библиотеки pandas, существовавшим в 2017 году.
- Добавлена глава о дополнительных средствах pandas, дающая также ряд других советов по работе с библиотекой.
- Размещено краткое введение в библиотеки statsmodels и scikit-learn.

Кроме того, материал, вошедший в первое издание, подвергнут реорганизации, чтобы книгу было проще читать начинающим пользователям.

Графические выделения

В книге применяются следующие графические выделения.

Курсив
Новые термины, имена и расширения имен 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.

`Моноширинный полужирный`
Команды или иной текст, который должен быть введен пользователем 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 предостережение или предупреждение.

О примерах кода

Файлы данных и прочие материалы, организованные по главам, можно найти в репозитории книги на GitHub: http://github.com/wesm/pydata-book.

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, не возбраняется включить в свою программу несколько фрагментов кода из книги, однако для продажи или распространения примеров из книг издательства O'Reilly на компакт-диске 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 позволение.

Мы высоко ценим (хотя и не требуем их размещения) ссылки на наши издания. В ссылке обычно указывается название книги, имя автора, издатель-

ство и ISBN, например: «Python for Data Analysis by Wes McKinney (O'Reilly). Copyright 2017 Wes McKinney, 978-1-491-95766-0».

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

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты

авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Эта работа – плод многолетних плодотворных обсуждений и совместной работы с многочисленными людьми со всего света. Хочу поблагодарить некоторых из них.

Памяти Джона Д. Хантера (1968–2012)

В августе 2012 года после многолетней борьбы с раком толстой кишки ушел из жизни наш дорогой друг и коллега Джон Д. Хантер. Это произошло почти сразу после того, как я закончил рукопись первого издания книги.

Роль и влияние Джона на сообщества, специализирующиеся на применении Python в научных приложениях и обработке данных, трудно переоценить. Помимо разработки библиотеки matplotlib в начале 2000-х годов (время, когда Python был далеко не так популярен, как сейчас) он помогал формировать культуру целого поколения разработчиков открытого кода, ставших впоследствии столпами экосистемы Python, которую мы часто считаем само собой разумеющейся.

Мне повезло познакомиться с Джоном в начале своей работы над открытым кодом в январе 2010 года, сразу после выхода версии pandas 0.1. Его вдохновляющее руководство помогало мне даже в самые тяжелые моменты не отказываться от своего видения pandas и Python как полноправного языка для анализа данных.

Джон был очень близок с Фернандо Пересом (Fernando Perez) и Брайаном Грейнджером (Brian Granger), которые заложили основы IPython и Jupyter и были авторами многих других инициатив в сообществе Python. Мы надеялись работать над книгой вчетвером, но в итоге только у меня оказалось достаточно свободного времени. Я уверен, что он гордился бы тем, чего мы достигли, порознь и совместно, за прошедшие пять лет.

Благодарности ко второму изданию (2017)

Прошло почти пять лет с того времени, как я закончил рукопись первого издания книги. Случилось это в июле 2012 года. С тех пор многое изменилось. Сообщество Python неизмеримо выросло, а сложившаяся вокруг него экосистема программных продуктов с открытым исходным кодом процветает.

Новое издание не появилось бы на свет без неустанных усилий разработчиков ядра pandas, благодаря которым этот проект и сложившееся вокруг него сообщество превратились в один из краеугольных камней экосистемы Python в области науки о данных. Назову лишь некоторых: Том Аугспургер (Tom Augspurger), Йорис ван ден Боше (Joris van den Bossche), Крис Бартак (Chris Bartak), Филлип Клауд (Phillip Cloud), gfyoung, Энди Хэйден (Andy Hayden), Масааки Хорикоши (Masaaki Horikoshi), Стивен Хойер (Stephan Hoyer), Адам Клейн (Adam Klein), Воутер Овермейр (Wouter Overmeire), Джэфф Ребэк (Jeff Reback), Чань Ши (Chang She), Скиппер Сиболд (Skipper Seabold), Джефф Трэтнер (Jeff Tratner) и др.

Что касается собственно подготовки издания, то я благодарю сотрудников издательства O'Reilly, которые терпеливо помогали мне на протяжении всего процесса работы над книгой, а именно Мари Божуро (Marie Beaugureau), Бена Лорика (Ben Lorica) и Коллин Топорек (Colleen Toporek). В очередной раз у меня были замечательные технические редакторы: Том Аугспургер, Пол Бэрри (Paul Barry), Хью Браун (Hugh Brown), Джонатан Коу (Jonathan Coe) и Андреас Маллер (Andreas Muller). Спасибо вам.

Первое издание книги переведено на ряд иностранных языков, в том числе на китайский, французский, немецкий, японский, корейский и русский. Перевод этого текста с целью сделать его доступным более широкой аудитории – трудное и зачастую неблагодарное занятие. Благодарю вас за то, что помогаете людям во всем мире учиться программировать и использовать средства анализа данных.

Мне также повезло пользоваться на протяжении нескольких последних лет поддержкой своих трудов по разработке ПО с открытым исходным кодом со стороны сайта Cloudera и фонда Two Sigma Investments. В то время как открытые проекты получают все меньший объем ресурсов, несопоставимый с количеством пользователей, очень важно, чтобы коммерческие компании поддерживали разработку ключевых программных проектов. Это было бы правильно.

Благодарности к первому изданию (2012)

Мне было бы трудно написать эту книгу без поддержки многих людей.

Из сотрудников издательства O'Reilly я крайне признателен редакторам – Меган Бланшетт (Meghan Blanchette) и Джулии Стил (Julie Steele), которые направляли меня на протяжении всего процесса. Майк Лоукидес (Mike Loukides) также работал со мной на стадии подачи предложения и помогал с выпуском книги в свет.

В техническом рецензировании книги многие принимали участие. Мартин Лас (Martin Blais) и Хью Браун (Hugh Brown) оказали неоценимую помощь в повышении качества примеров, ясности изложения и улучшении организации книги в целом. Джеймс Лонг (James Long), Дрю Конвей (Drew Conway), Фернандо Перес, Брайан Грейнджер, Томас Клюйвер (Thomas Kluyver), Адам

Клейн, Джон Клейн, Чань Ши и Стефан ван дер Вальт (Stefan van der Walt) отрецензировали по одной или по нескольку глав и сделали ценные замечания с разных точек зрения.

Я почерпнул немало отличных идей для примеров и наборов данных в беседах с друзьями и коллегами, в том числе с Майком Дьюаром (Mike Dewar), Джеффом Хаммербахером (Jeff Hammerbacher), Джеймсом Джондроу (James Johndrow), Кристианом Ламом (Kristian Lum), Адамом Клейном, Хилари Мейсон (Hilary Mason), Чань Ши и Эшли Вильямсом (Ashley Williams).

Конечно, я в долгу у многих лидеров сообщества, применяющего открытое ПО на Python в научных приложениях, поскольку именно они заложили фундамент моей работы и воодушевляли меня, пока я писал книгу. Это те, кто разрабатывал ядро IPython (Фернандо Перес, Брайан Грейнджер, Мин Рэган-Келли, Томас Клюйвер и др.), Джон Хантер, Скиппер Сиболд, Трэвис Олифант (Travis Oliphant), Питер Вонг (Peter Wang), Эрик Джонс (Eric Jones), Робер Керн (Robert Kern), Джозеф Перктольд (Josef Perktold), Франческ Альтед (Francesc Alted), Крис Фоннесбек (Chris Fonnesbeck) и многие, многие другие. Еще несколько человек оказывали мне значительную поддержку, делились идеями и подбадривали на протяжении всего пути: Дрю Конвей, Шон Тэйлор (Sean Taylor), Джузеппе Палеолого (Giuseppe Paleologo), Джаред Дандер (Jared Lander), Дэвид Эпштейн (David Epstein), Джон Кроуос (John Krowas), Джошуа Блум (Joshua Bloom), Дэн Пилсуорт (Den Pilsworth), Джон Майлз-Уайт (John Myles-White) и многие другие, о которых я забыл.

Я также благодарен всем, кто повлиял на становление меня как ученого. В первую очередь это мои бывшие коллеги по компании AQR, которые поддерживали мою работу над pandas в течение многих лет: Алекс Рейфман (Alex Reyfman), Майкл Вонг (Michael Wong), Тим Сарджен (Tim Sargen), Октай Курбанов (Oktay Kurbanov), Мэтью Щанц (Matthew Tschantz), Рони Израэлов (Roni Israelov), Майкл Кац (Michael Katz), Крис Уга (Chris Uga), Прасад Раманан (Prasad Ramanan), Тэд Сквэр (Ted Square) и Хун Ким (Hoon Kim). И наконец, благодарю моих университетских наставников Хэйнса Миллера (МТИ) и Майка Уэста (университет Дьюк).

Если говорить о личной жизни, то я благодарен Кэйси Динкин (Casey Dinkin), чью каждодневную поддержку невозможно переоценить. Спасибо той, кто терпел перепады моего настроения, когда я пытался собрать окончательный вариант рукописи, несмотря на свой и так уже перегруженный график. Благодарю и моих родителей, Билла и Ким, которые учили меня никогда не отступать от мечты и не соглашаться на меньшее.

Об авторе

Уэс Маккини – разработчик программного обеспечения и предприниматель из Нью-Йорка. После получения степени бакалавра математики в МТИ в 2007 году его приняли на работу в компанию AQR Capital Management в Гринвиче, где занимался финансовой математикой. Неудовлетворенный малопригодными средствами анализа данных, Уэс изучил язык Python и приступил к созданию того, что в будущем стало проектом pandas. Сейчас он активный член сообщества обработки данных на Python и агитирует за использование Python в анализе данных, финансовых задачах и математической статистике.

Впоследствии Уэс стал сооснователем и генеральным директором компании DataPad, технологические активы и коллектив которой в 2014 году приобрела компания Cloudera. С тех пор он занимается технологиями больших данных и является членом комитетов по управлению проектами Apache Arrow и Apache Parquet, курируемыми фондом Apache Software Foundation. В 2016 году автор перешел в компанию Two Sigma Investments из Нью-Йорка, где продолжает трудиться над тем, чтобы средствами ПО с открытым исходным кодом сделать анализ данных быстрее и проще.

Об иллюстрации на обложке

На обложке книги изображена перохвостая тупайя (*Ptilocercus lowii*). Это единственный представитель своего вида в семействе *Ptilocercidae* рода *Ptilocercus*; остальные тупайи принадлежат семейству *Tupaiidae*. Тупайи отличаются длинным хвостом и мягким буро-желтым мехом. У перохвостой тупайи хвост напоминает птичье перо, за что она и получила свое название. Тупайи всеядны, питаются преимущественно насекомыми, фруктами, семенами и небольшими позвоночными животными.

Эти дикие млекопитающие, обитающие в основном в Индонезии, Малайзии и Таиланде, известны хроническим потреблением алкоголя. Как выяснилось, малайские тупайи несколько часов в сутки пьют естественно ферментированный нектар пальмы *Eugeissona tristis*, что эквивалентно употреблению 10–12 стаканов вина, содержащего 3,8 % алкоголя. Но это не приводит к интоксикации их организма благодаря развитой способности расщеплять этиловый спирт, включая его в обмен веществ способами, недоступными человеку. Кроме того, поражает отношение массы мозга к массе тела – оно больше, чем у всех прочих млекопитающих, в том числе у человека.

Несмотря на название, перохвостая тупайя не является настоящей тупайей, а ближе к приматам. Вследствие такого родства перохвостые тупайи стали альтернативой приматам в медицинских экспериментах по изучению миопии, психосоциального стресса и гепатита.

Глава 1. Предварительные сведения

1.1. О чем эта книга?

Книга посвящена вопросам преобразования, обработки, очистки данных и вычислениям на языке Python. Моя цель – предложить руководство по тем частям языка программирования Python и экосистемы его библиотек и инструментов, относящихся к обработке данных, которые помогут вам стать хорошим аналитиком данных. Хотя в названии книги фигурируют слова «анализ данных», основной упор сделан на программировании на Python, на библиотеках и инструментах, а не на методологии анализа данных как таковой. Речь идет о программировании на Python, необходимом для анализа данных.

Какого рода данные?

Говоря «данные», я имею в виду прежде всего *структурированные данные*. Это намеренно расплывчатый термин, охватывающий различные часто встречающиеся виды данных, как то:

- табличные данные – в разных столбцах они могут иметь разный тип (строки, числа, даты или еще что-то). Сюда относятся данные, которые обычно хранятся в реляционных базах или в файлах с запятой в качестве разделителя;
- многомерные списки (матрицы);
- данные, представленные в виде нескольких таблиц, связанных между собой по ключевым столбцам (то, что в SQL называется первичными и внешними ключами);
- равноотстоящие и неравноотстоящие временные ряды.

Это далеко не полный список. Значительную часть наборов данных можно привести к структурированному виду, более подходящему для анализа и моделирования, хотя сразу не всегда очевидно, как это сделать. В тех случаях, когда это не удается, есть возможность извлечь из набора данных структурированное множество признаков. Например, подборку новостных статей можно преобразовать в таблицу частот слов, к которой затем применить анализ эмоциональной окраски.

Большинству пользователей электронных таблиц типа Microsoft Excel, пожалуй, самого широко распространенного средства анализа данных, такие виды данных хорошо знакомы.

1.2. Почему именно Python?

Для многих людей (и для меня в том числе) Python – язык, в который нельзя не влюбиться. С момента появления в 1991 году Python стал одним из самых популярных динамических языков программирования наряду с Perl, Ruby и др. Относительно недавно Python и Ruby приобрели особую популярность как средства создания веб-сайтов в многочисленных каркасах, например Rails (Ruby) и Django (Python). Такие языки часто называют *скриптовыми*, потому что они используются для быстрого написания небольших программ – *скриптов*. Лично мне термин «скриптовый язык» не нравится, потому что он наводит на мысль, будто для создания ответственного программного обеспечения язык не годится. Из всех интерпретируемых языков Python выделяется большим и активным сообществом научных расчетов и анализа данных. За последние десять лет Python превратился из ультрасовременного языка научных расчетов, которым пользуются на свой страх и риск, в один из самых важных языков, применяемых в науке о данных, в машинном обучении и разработке ПО общего назначения в академических учреждениях и промышленности.

При анализе данных и интерактивных научно-исследовательских расчетов с визуализацией результатов Python неизбежно приходится сравнивать со многими предметно-ориентированными языками программирования и инструментами – с открытым исходным кодом и коммерческими, такими как R, MATLAB, SAS, Stata и др. Сравнительно недавно появились улучшенные библиотеки для Python (прежде всего pandas), и он стал серьезным конкурентом в решении задач манипулирования данными. А так как Python еще – и универсальный язык программирования, то это отличный выбор для создания приложений обработки данных.

Python как клей

Своим успехом в области научных расчетов Python отчасти обязан простоте интеграции с кодом на C, C++ и FORTRAN. Во многих современных вычислительных средах применяется общий набор унаследованных библиотек,

написанных на FORTRAN и C, содержащих реализации алгоритмов линейной алгебры, оптимизации, интегрирования, быстрого преобразования Фурье и др. Поэтому многочисленные компании и национальные лаборатории используют Python как клей для объединения написанных за много лет программ.

В значительном количестве программ содержатся небольшие участки кода, на выполнение которых уходит много времени, и внушительные куски склеивающего кода, который выполняют нечасто. В большинстве случаев время выполнения склеивающего кода несущественно, реальную отдачу дает оптимизация узких мест, которые иногда имеет смысл переписать на низкоуровневом языке типа C.

Решение проблемы «двух языков»

Во многих организациях принято для научных исследований, создания опытных образцов и проверки новых идей использовать предметно-ориентированные языки типа MATLAB или R, а затем переносить удачные разработки в производственную систему, написанную на Java, C# или C++. Но все чаще люди приходят к выводу, что Python подходит не только для исследования и создания прототипа, но и для построения самих производственных систем. Полагаю, что компании большей частью будут выбирать этот путь, потому что использование учеными и технологами одного и того же набора программных средств, несомненно, выгодно для организации.

Недостатки Python

Python – великолепная среда для создания приложений для научных расчетов и большинства систем общего назначения, но тем не менее существуют задачи, которым Python не очень подходит.

Поскольку Python – интерпретируемый язык программирования, в общем случае написанный на нем код работает значительно медленнее, чем эквивалентный код на компилируемом языке типа Java или C++. Но поскольку *время программиста* обычно стоит гораздо дороже *времени процессора*, многих такой компромисс устраивает. Однако в приложениях, где задержка должна быть очень мала (например, в торговых системах с большим количеством транзакций), время, потраченное на программирование на низкоуровневом и не обеспечивающем максимальную продуктивность языке типа C++ во имя достижения максимальной производительности, будет потрачено не зря.

Python не идеальный язык для программирования многопоточных приложений с высокой степенью параллелизма, особенно при наличии многих потоков, активно использующих процессор. Проблема связана с наличием *глобальной блокировки интерпретатора* (GIL) – механизма, который не дает интерпретатору исполнять более одной команды байт-кода Python в каждый

момент времени. Объяснение технических причин существования GIL выходит за рамки этой книги, но на данный момент представляется, что GIL вряд ли скоро исчезнет. И хотя во многих приложениях обработки больших объектов данных, для того чтобы сократить срок работы, приходится организовывать кластер машин, встречаются все же ситуации, когда более желательна однопроцессная многопоточная система.

Я не хочу сказать, что Python вообще непригоден для исполнения многопоточного параллельного кода. Написанные на C или C++ расширения Python, пользующиеся платформенной многопоточностью, могут исполнять код параллельно и не ограничены механизмом GIL, при условии что им не нужно регулярно взаимодействовать с Python-объектами.

1.3. Необходимые библиотеки для Python

Для читателей, плохо знакомых с экосистемой Python и используемыми в книге библиотеками, я сделаю краткий обзор библиотек.

NumPy

NumPy, сокращение от «Numerical Python», – основной пакет для выполнения научных расчетов на Python. Большая часть этой книги базируется на NumPy и построенных поверх него библиотеках. В числе прочего он предоставляет:

- быстрый и эффективный объект многомерного массива *ndarray*;
- функции для выполнения вычислений над элементами одного массива или математических операций с несколькими массивами;
- средства для чтения и записи на диски наборов данных, представленных в виде массивов;
- операции линейной алгебры, преобразование Фурье и генератор случайных чисел;
- зрелый C API, позволяющий обращаться к структурам данных и вычислительным средствам NumPy из расширений Python и кода на C или C++.

Помимо ускорения работы с массивами, одной из основных целей NumPy в части анализа данных является организация контейнера для передачи данных между алгоритмами. Как средство хранения и манипуляции данными массивы NumPy куда эффективнее встроенных в Python структур данных. Кроме того, библиотеки, написанные на низкоуровневом языке типа C или Fortran, могут работать с данными, хранящимися в массиве NumPy, вообще без копирования в другое представление. Таким образом, многие средства вычислений, ориентированные на Python, либо используют массивы NumPy в качестве основной структуры данных, либо каким-то иным способом организуют интеграцию с NumPy.

pandas

Библиотека pandas предоставляет структуры данных и функции, призванные сделать работу со структурированными данными простой, быстрой и выразительной. С момента появления в 2010 году она способствовала превращению Python в мощную и продуктивную среду анализа данных. Основные объекты pandas, используемые в книге, – это `DataFrame` – двумерная таблица, в которой строки и столбцы имеют метки, и `Series` – объект одномерного массива с метками.

В библиотеке pandas сочетаются высокая производительность средств работы с массивами, присущая NumPy, и гибкие возможности манипулирования данными, свойственные электронным таблицам и реляционным базам данных (например, на основе SQL). Она предоставляет развитые средства индексирования, позволяющие без труда изменять форму наборов данных, формировать продольные и поперечные срезы, выполнять агрегирование и выбирать подмножества. Поскольку манипулирование данными, их подготовка и очистка играют огромную роль в анализе данных, в этой книге библиотека pandas будет одним из основных инструментов.

Если кому интересно, я начал разрабатывать pandas в начале 2008 года, когда работал в компании AQR Capital Management, занимающейся управлением инвестициями. Тогда я сформулировал специфический набор требований, которым не удовлетворял ни один отдельно взятый инструмент, имевшийся в моем распоряжении:

- структуры данных с помеченными осями, которые поддерживали бы автоматическое или явное выравнивание данных, – это предотвратило бы появление типичных ошибок при работе с невыровненными данными и данными из разных источников, которые по-разному индексированы;
- встроенная функциональность временных рядов;
- одни и те же структуры данных должны поддерживать как временные ряды, так и данные других видов;
- арифметические операции и упрощения должны сохранять метаданные;
- гибкая обработка отсутствующих данных;
- поддержка соединения и других реляционных операций, имеющихся в популярных базах данных (например, на основе SQL).

Я хотел, чтобы вся эта функциональность находилась в одном месте и предпочтительно была реализована на языке, хорошо приспособленном для разработки ПО общего назначения. Python выглядел хорошим кандидатом на эту роль, но в то время в нем не было подходящих встроенных структур данных и средств. Поскольку изначально библиотека pandas создавалась для решения финансовых задач и задач бизнес-аналитики, в ней особенно глубоко проработаны средства работы с временными рядами, ориентированные на обработку данных с временными метками, которые порождаются бизнес-процессами.

Пользователям языка статистических расчетов R название DataFrame покажется знакомым, потому что оно выбрано по аналогии с объектом `data.frame` в R. В отличие от Python, фреймы данных уже встроены в язык R и его стандартную библиотеку, поэтому многие средства, присутствующие в pandas, либо являются частью ядра R, либо предоставляются дополнительными пакетами.

Само название pandas образовано как от *panel data* (панельные данные), применяемого в эконометрике термина для обозначения многомерных структурированных наборов данных, так и от фразы *Python data analysis*.

matplotlib

Библиотека matplotlib – самый популярный в Python инструмент для создания графиков и других способов визуализации двумерных данных. Первоначально она была написана Джоном Д. Хантером (John D. Hunter), а теперь сопровождается большой группой разработчиков. Она отлично подходит для создания графиков, пригодных для публикации. Хотя программистам на Python доступны и другие библиотеки визуализации, matplotlib используется чаще всего и потому хорошо интегрирована с другими частями экосистемы. На мой взгляд, если вам нужно средство визуализации, то это самый безопасный выбор.

IPython и Jupyter

Проект IPython (http://ipython.org/) начал реализовывать в 2001 году Фернандо Перес (Fernando Perez) в качестве побочного, имеющего целью создать более удобный интерактивный интерпретатор Python. За прошедшие с тех пор 16 лет он стал одним из самых важных элементов современного инструментария Python. Хотя IPython сам по себе не содержит никаких средств вычислений или анализа данных, он изначально спроектирован, чтобы обеспечивать максимальную продуктивность интерактивных вычислений и разработки ПО. Он поощряет цикл *выполнить – исследовать* вместо привычного цикла *редактировать – компилировать – выполнить*, свойственного многим другим языкам программирования. Кроме того, он позволяет легко обращаться к оболочке и файловой системе операционной системы. Поскольку написание кода анализа данных часто подразумевает исследование методом проб и ошибок и опробование разных подходов, то благодаря IPython работу удается выполнить быстрее.

В 2014 году Фернандо и команда разработки IPython анонсировали проект Jupyter (https://jupyter.org/) – широкую инициативу проектирования языково-независимых средств интерактивных вычислений. Веб-блокнот IPython превратился в Jupyter-блокнот, который ныне поддерживает более 40 языков программирования. Систему IPython теперь можно использовать как *ядро* (языковой режим) для совместной работы Python и Jupyter.

Сам IPython стал компонентом более широкого проекта Jupyter с открытым исходным кодом, предоставляющего продуктивную среду для интерактивных исследовательских вычислений. В своем самом старом и простом режиме это улучшенная оболочка Python, имеющая целью ускорить написание, тестирование и отладку кода на Python. Систему IPython можно использовать также через Jupyter-блокнот, интерактивный веб-блокнот, поддерживающий десятки языков программирования. Оболочка IPython и Jupyter-блокноты особенно полезны для исследования и визуализации данных.

Кроме того, Jupyter-блокноты позволяют создавать контент на языках разметки Markdown и HTML, т. е. готовить комбинированные документы, содержащие код и текст. На других языках программирования также реализованы ядра для Jupyter, благодаря чему их можно использовать вместо Python.

Лично я при работе с Python использую в основном IPython – для выполнения, отладки и тестирования кода.

В сопроводительных материалах к книге (https://github.com/wesm/pydata-book) вы найдете Jupyter-блокноты, содержащие примеры кода к каждой главе.

SciPy

SciPy – собрание пакетов, предназначенных для решения различных стандартных вычислительных задач. Вот некоторые из них:

- `scipy.integrate` – подпрограммы численного интегрирования и решения дифференциальных уравнений;
- `scipy.linalg` – подпрограммы линейной алгебры и разложения матриц, дополняющие те, что включены в `numpy.linalg`;
- `scipy.optimize` – алгоритмы оптимизации функций (нахождения минимумов) и поиска корней;
- `scipy.signal` – средства обработки сигналов;
- `scipy.sparse` – алгоритмы работы с разреженными матрицами и решения разреженных систем линейных уравнений;
- `scipy.special` – обертка вокруг SPECFUN, написанной на Fortran-библиотеке, содержащей реализации многих стандартных математических функций, в том числе гамма-функции;
- `scipy.stats` – стандартные непрерывные и дискретные распределения вероятностей (функции плотности вероятности, формирования выборки, функции непрерывного распределения вероятности), различные статистические критерии и дополнительные описательные статистики.

Совместно NumPy и SciPy достаточно полно заменяют значительную часть системы MATLAB и многочисленные дополнения к ней.

scikit-learn

Проект scikit-learn (https://scikit-learn.org/stable/), запущенный в 2010 году, с самого начала стал основным инструментарием машинного обучения про-

граммистов на Python. Всего за семь лет к нему присоединилось 1500 разработчиков со всего мира. В нем имеются подмодули для следующих моделей:

- классификация: метод опорных векторов, метод ближайших соседей, случайные леса, логистическая регрессия и т. д.;
- регрессия: Lasso, гребневая регрессия и т. д.;
- кластеризация: метод *k* средних, спектральная кластеризация и т. д.;
- понижение размерности: метод главных компонент, отбор признаков, матричная факторизация и т. д.;
- выбор модели: поиск на сетке, перекрестный контроль, метрики;
- предобработка: выделение признаков, нормировка.

Наряду с pandas, statsmodels и IPython библиотека scikit-learn сыграла важнейшую роль для превращения Python в продуктивный язык программирования для науки о данных. Я не смогу включить в эту книгу полное руководство по scikit-learn, но все же предложу краткое введение в некоторые используемые в ней модели и объясню, как их использовать совместно с другими средствами.

statsmodels

Пакет статистического анализа statsmodels (http://www.statsmodels.org/stable/index.html) начал разрабатываться по инициативе профессора статистики из Стэнфордского университета Джонатана Тэйлора (Jonathan Taylor), который реализовал ряд моделей регрессионного анализа, популярных в языке программирования R. Скиппер Сиболд (Skipper Seabold) и Джозеф Перктольд (Josef Perktold) формально создали новый проект statsmodels в 2010 году, и с тех пор он набрал критическую массу заинтересованных пользователей и соразработчиков. Натаниэль Смит (Nathaniel Smith) разработал проект Patsy, который предоставляет средства для задания формул и моделей для statsmodels по образцу системы формул в R.

По сравнению со scikit-learn, пакет statsmodels содержит алгоритмы классической (прежде всего частотной) статистики и эконометрики. В него входят следующие подмодули:

- регрессионные модели: линейная регрессия, обобщенные линейные модели, робастные линейные модели, линейные модели со смешанными эффектами и т. д.;
- дисперсионный анализ (ANOVA);
- анализ временных рядов: AR, ARMA, ARIMA, VAR и другие модели;
- непараметрические методы: ядерная оценка плотности, ядерная регрессия;
- визуализация результатов статистического моделирования.

Пакет statsmodels ориентирован в большей степени на статистический вывод, он дает оценки неопределенности и *p*-значения параметров. Напротив, scikit-learn ориентирован главным образом на предсказание.

Как и для scikit-learn, я создам краткое введение в statsmodels и объясню, как им пользоваться в сочетании с NumPy и pandas.

1.4. Установка и настройка

Поскольку Python используется в самых разных приложениях, не существует единственно верной процедуры установки Python и необходимых дополнительных пакетов. У многих читателей, скорее всего, нет среды, подходящей для научных применений Python и проработки этой книги, поэтому я дам подробные инструкции для разных операционных систем. Рекомендую использовать бесплатный дистрибутив Anaconda. На момент написания книги Anaconda предлагается для версий Python 2.7 и 3.6, хотя в будущем это может измениться. В книге используется Python 3.6, и я настоятельно рекомендую работать именно с этой или более старшей версией.

Windows

В Windows начните со скачивания установщика Anaconda (https://www.anaconda.com/distribution/). Рекомендую следовать инструкциям, опубликованным на странице скачивания Anaconda, при этом надо понимать, что после выхода книги из печати они могли измениться.

По завершении установки проверьте, все ли сконфигурировано правильно. Откройте окно командной строки (это приложение называется *cmd.exe*), для чего щелкните правой кнопкой мыши по меню **Пуск** и выберите пункт **Командная строка**[1]. Попробуйте запустить интерпретатор Python, набрав команду **python**. Должно появиться сообщение, соответствующее установленной версии Anaconda:

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13)
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

Чтобы выйти из оболочки, нажмите **Ctrl**+**Z** или введите команду **exit()** и щелкните по **Enter**.

Apple OS X

Скачайте установщик Anaconda для OS X Anaconda. Он должен называться как-то вроде *Anaconda3-4.1.0-MacOSX-x86_64.pkg*. Дважды щелкните по *pkg*-файлу для запуска установщика. Установщик автоматически добавит путь к исполняемому файлу Anaconda в ваш файл *.bash_profile*, полный путь к которому имеет вид */Users/$USER/.bash_profile*.

---

[1] В Windows 10 этого пункта в меню **Пуск** по умолчанию нет. Чтобы открыть окно командной строки, найдите программу cmd.exe с помощью средства поиска. – *Прим. перев.*

Для проверки работоспособности попробуйте запустить IPython из оболочки системы (для получения командной строки откройте приложение Terminal):

```
$ ipython
```

Чтобы выйти из оболочки, нажмите **Ctrl**+**D** или введите команду **exit()** и щелкните по **Enter**.

GNU/Linux

Детали установки в Linux варьируются в зависимости от дистрибутива. Я опишу процедуру, работающую в дистрибутивах Debian, Ubuntu, CentOS и Fedora. Установка в основных чертах производится так же, как для OS X, отличается только порядок установки Anaconda. Установщик представляет собой скрипт оболочки, запускаемый из терминала. В зависимости от разрядности системы нужно выбрать установщик типа x86 (32-разрядный) или x86_64 (64-разрядный). Имя соответствующего файла имеет вид *Anaconda3-4.1.0-Linux-x86_64.sh*. Для установки нужно выполнить такую команду:

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

В некоторых дистрибутивах Linux менеджеры пакетов, например apt, располагают всеми необходимыми Python-пакетами. Ниже описывается установка с помощью Anaconda, поскольку она одинакова во всех дистрибутивах и упрощает обновление пакетов до последней версии.

После подтверждения согласия с лицензией вам будет предложено указать место установки файлов Anaconda. Я рекомендую устанавливать их в свой домашний каталог, например */home/$USER/anaconda* (вместо `$USER` подставьте свое имя пользователя).

Установщик Anaconda может спросить, хотите ли вы добавить каталог *bin/* в начало переменной `$PATH`. Если после установки возникнут проблемы, это можно сделать вручную, модифицировав файл *.bashrc* (или *.zshrc*, если вы пользуетесь оболочкой zsh) примерно таким образом:

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

Затем запустите новый процесс терминала или повторно выполните файл *.bashrc* командой `source ~/.bashrc`.

Установка или обновление Python-пакетов

Рано или поздно вам могут понадобиться дополнительные Python-пакеты, не включенные в дистрибутив Anaconda. В общем случае они устанавливаются такой командой:

```
conda install package_name
```

Если это не работает, то, возможно, удастся установить пакет с помощью менеджера пакетов pip:

```
pip install package_name
```

Для обновления пакетов служит команда `conda update`:

```
conda update package_name
```

pip также поддерживает обновление, нужно только задать флаг `--upgrade`:

```
pip install --upgrade package_name
```

В этой книге вам не раз представится возможность попробовать эти команды в деле.

Если для установки пакетов вы используете и conda, и pip, то не следует пытаться обновлять пакеты conda с помощью pip, поскольку из-за этого может повредиться среда. При работе с Anaconda или Miniconda всегда рекомендуется сначала попробовать обновить пакет командой conda.

Python 2 и Python 3

Первая версия в линейке интерпретаторов Python 3.x была выпущена в конце 2008 года. В нее включены изменения, несовместимые с ранее написанным кодом для версий Python 2.x. Поскольку с момента выхода первой версии Python в 1991 году прошло уже 17 лет, создание несовместимой версии Python 3 рассматривалось как благо – принимая во внимание все уроки прошлого.

В 2012 году разработчики и пользователи приложений Python для научных расчетов и анализа данных работали в основном с версиями 2.x, поскольку многие пакеты еще не были переведены на Python 3. Поэтому в первом издании книги использовалась версия Python 2.7. Но теперь пользователи могут выбирать между Python 2.x и 3.x, так как большинство библиотек поддерживают обе ветви.

Однако поддержка Python 2.x прекратится в 2020 году (это относится и к критическим исправлениям безопасности), так что начинать новые проекты на Python 2.7 не стоит. Поэтому в книге используется Python 3.6 – стабильная, хорошо поддерживаемая и широко распространенная версия. Все уже начали называть Python 2.x унаследованным Python, а Python 3.x – просто Python. Призываю и вас последовать этому примеру.

Я взял за основу версию Python 3.6. У вас может быть более свежая версия, но все примеры кода должны быть совместимы с ней. В Python 2.7 некоторые примеры могут работать иначе или не работать вовсе.

Интегрированные среды разработки (IDE)

Когда меня спрашивают о том, какой средой разработки я пользуюсь, я почти всегда отвечаю: «IPython плюс текстовый редактор». Обычно я пишу программу и итеративно тестирую и отлаживаю ее по частям в IPython или Jupyter-блокнотах. Полезно также иметь возможность интерактивно экс-

периментировать с данными и визуально проверять, получается ли в результате определенных манипуляций ожидаемый результат. Библиотеки pandas и NumPy спроектированы с учетом простоты использования в оболочке.

Однако некоторые пользователи предпочитают разрабатывать программы в полноценной IDE, а не в сравнительно примитивном текстовом редакторе типа Emacs или Vim. Вот некоторые доступные варианты:

- PyDev (бесплатная) – IDE, построенная на платформе Eclipse;
- PyCharm от компании JetBrains (на основе подписки для коммерческих компаний, бесплатна для разработчиков ПО с открытым исходным кодом);
- Python Tools для Visual Studio (для работающих в Windows);
- Spyder (бесплатная) – IDE, которая в настоящий момент поставляется в составе Anaconda;
- Komodo IDE (коммерческая).

1.5. Сообщество и конференции

Помимо поиска в интернете можно пользоваться полезными рассылками, посвященными применению Python в научных расчетах и для обработки данных. Их участники быстро отвечают на вопросы. Вот некоторые из таких ресурсов:

- pydata: группа Google по вопросам, относящимся к использованию Python для анализа данных и pandas;
- pystatsmodels: вопросы, касающиеся statsmodels и pandas;
- numpy-discussion: вопросы, касающиеся NumPy;
- рассылка по scikit-learn (scikit-learn@python.org) и машинному обучению на Python вообще;
- scipy-user: общие вопросы использования SciPy и Python для научных расчетов.

Я сознательно не публикую URL-адреса, потому что они часто меняются. Поиск в интернете вам в помощь.

Ежегодно в разных странах проходят конференции для программистов на Python. Если вы захотите пообщаться с другими программистами, которые разделяют ваши интересы, то имеет смысл посетить какое-нибудь мероприятие (если есть такая возможность). Многие конференции оказывают финансовую поддержку тем, кто не может позволить себе вступительный взнос или транспортные расходы. Приведу неполный перечень конференций:

- PyCon and EuroPython: две самые крупные, проходящие соответственно в Северной Америке и в Европе;
- SciPy и EuroSciPy: конференции, ориентированные на научные применения Python, проходящие соответственно в Северной Америке и в Европе;

- PyData: мировая серия региональных конференций, посвященных науке о данных и анализу данных;
- международные и региональные конференции PyCon (полный список см. на сайте http://pycon.org).

1.6. Структура книги

Если вы раньше никогда не программировали на Python, то имеет смысл потратить время на знакомство с главами 2 и 3, где я поместил очень краткое руководство по языковым средствам Python, а также по оболочке IPython и Jupyter-блокнотам. Эти знания необходимы для чтения книги. Если у вас уже есть опыт работы с Python, то можете вообще пропустить эти главы или пролистать их по диагонали.

Далее дается краткое введение в основные возможности NumPy, а более подробное изложение имеется в приложении A. Затем мы познакомимся с pandas и посвятим оставшуюся часть книги анализу данных с применением pandas, NumPy и matplotlib (для визуализации). Я старался построить изложение по возможности поступательно, хотя иногда главы немного пересекаются, и есть несколько случаев, когда используются еще не описанные концепции.

У разных читателей могут быть разные цели, но, вообще говоря, можно предложить следующую классификацию задач.

- Взаимодействие с внешним миром – чтение и запись в файлы и хранилища данных различных форматов.
- Подготовка – очистка, переформатирование, комбинирование, нормализация, изменение формы, получение продольных и поперечных срезов, трансформация данных для анализа.
- Преобразование – применение математических и статистических операций к группам наборов данных для получения новых наборов (например, агрегирование большой таблицы по некоторым переменным).
- Моделирование и вычисления – связывание данных со статистическими моделями, алгоритмами машинного обучения и иными вычислительными средствами.
- Презентация – создание интерактивных или статических графических визуализаций или текстовых сводных отчетов.

Примеры кода

Примеры кода в большинстве случаев показаны так, как выглядят в оболочке IPython или Jupyter-блокнотах: ввод и вывод.

```
In [5]: КОД
Out[5]: РЕЗУЛЬТАТ
```

Это означает, что вы должны ввести код в блоке `In` в своей рабочей среде и выполнить его, нажав клавишу **Enter** (или **Shift**-**Enter** в Jupyter). Результат должен быть таким, как показано в блоке `Out`.

Данные для примеров

Наборы данных для примеров из каждой главы находятся в репозитории на сайте GitHub: https://github.com/wesm/pydata-book. Вы можете получить их либо с помощью командной утилиты системы управления версиями git, либо скачав zip-файл репозитория с сайта. Если возникнут проблемы, заходите на мой сайт (https://wesmckinney.com/), где выложены актуальные инструкции по получению материалов к книге.

Я стремился сделать так, чтобы в репозиторий попало все необходимое для воспроизведения примеров, но мог где-то ошибиться или что-то пропустить. В таком случае пишите мне на адрес book@wesmckinney.com. Самый лучший способ сообщить об ошибках, найденных в книге, – описать их на странице опечаток на сайте издательства O'Reilly (https://www.oreilly.com/catalog/errata.csp?isbn=0636920050896).

Соглашения об импорте

В сообществе Python принят ряд соглашений об именовании наиболее употребительных модулей:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

Это означает, что `np.arange` – ссылка на функцию `arange` в пакете NumPy. Так делается, потому что импорт всех имен из большого пакета, каким является NumPy (`from numpy import *`), считается среди разработчиков на Python дурным тоном.

Жаргон

Я употребляю некоторые термины, встречающиеся как в программировании, так и в науке о данных, с которыми вы, возможно, незнакомы. Поэтому приведу краткие определения.

- *Переформатирование* (Munge/Munging/Wrangling) – процесс приведения неструктурированных и (или) замусоренных данных к структурированной или чистой форме. Слово вошло в лексикон многих современных специалистов по анализу данных.
- *Псевдокод* – описание алгоритма или процесса в форме, напоминающей код, хотя фактически это не есть корректный исходный код на каком-то языке программирования.
- *Синтаксический сахар* – синтаксическая конструкция, которая не добавляет новую функциональность, а лишь вносит дополнительное удобство или позволяет сделать код короче.

Глава 2. Основы языка Python, IPython и Jupyter-блокноты

В 2011 и 2012 годах, когда я писал первое издание книги, ресурсов для изучения анализа данных с применением Python было гораздо меньше. Тут мы имеем что-то похожее на проблему яйца и курицы: многие библиотеки, наличие которых мы сейчас считаем само собой разумеющимся, в том числе pandas, scikit-learn и statsmodels, тогда были еще относительно незрелыми. В 2017 году количество литературы по науке о данных, по анализу данных и машинному обучению неуклонно растет, дополняя прежние работы по научным расчетам, предназначенные для специалистов по информатике, физике и другим дисциплинам. Есть также замечательные книги о самом языке программирования Python и о том, как стать эффективным программистом.

Поскольку книга задумана как введение в работу с данными на Python, считаю полезным дать замкнутый обзор некоторых наиболее важных особенностей встроенных в Python структур данных и библиотек с точки зрения манипулирования данными. Поэтому в этой и следующей главах приводится лишь информация, необходимая для чтения книги.

На мой взгляд, для продуктивного анализа данных вовсе *необязательно* профессионально заниматься разработкой ПО на Python. Я призываю вас экспериментировать с примерами кода и изучать документацию по различным типам, функциям и методам, используя оболочку IPython и Jupyter-блокноты. Хотя я изо всех сил старался излагать материал поступательно, иногда могут встретиться вещи, которые еще не объяснялись.

Книга посвящена в основном инструментам табличного анализа и подготовки данных для работы с большими наборами. Чтобы применить эти инструменты, зачастую необходимо сначала преобразовать беспорядочные

данные низкого качества в более удобную табличную (или *структурную*) форму. К счастью, Python – идеальный язык для быстрого приведения данных к нужному виду. Чем свободнее вы владеете языком, тем проще будет подготовить новый набор данных для анализа.

Некоторые описанные в книге инструменты лучше изучать в интерактивном сеансе IPython или Jupyter. После того как научитесь запускать IPython и Jupyter, я рекомендую проработать примеры, экспериментируя и пробуя разные подходы. Как и в любом окружении, ориентированном на работу с клавиатурой, полезно запомнить наиболее употребительные команды на подсознательном уровне.

Некоторые базовые понятия Python, например классы и объектно-ориентированное программирование, в этой главе не рассматриваются, хотя их полезно включить в арсенал средств для анализа данных. Желающим углубить свои знания рекомендую дополнить эту главу официальным пособием по Python (https://docs.python.org/3/) и, возможно, одной из многих замечательных книг по программированию на Python вообще. Начать можно, например, с таких книг:

- *Python Cookbook*, Third Edition, by David Beazley and Brian K. Jones (O'Reilly);
- *Fluent Python* by Luciano Ramalho (O'Reilly)[1];
- *Effective Python* by Brett Slatkin (Pearson)[2].

2.1. Интерпретатор Python

Python – *интерпретируемый* язык. Интерпретатор Python исполняет программу по одному предложению за раз. Стандартный интерактивный интерпретатор Python запускается из командной строки командой `python`:

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

Строка `>>>` – это приглашение к вводу выражения. Для выхода из интерпретатора Python и возврата в командную строку нужно либо ввести команду `exit()`, либо нажать **Ctrl**+**D**.

Для выполнения Python-программы нужно просто набрать команду `python`, указав в качестве первого аргумента имя файла с расширением `.py`. Допустим, вы создали файл `hello_world.py` с таким содержимым:

```
print 'Hello world'
```

[1] *Рамальо Л.* Python. К вершинам мастерства. М.: ДМК Пресс, 2016.

[2] *Сламкин Б.* Секреты Python. М.: Вильямс, 2017.

Чтобы выполнить его, достаточно ввести следующую команду (файл *hello_world.py* должен находиться в текущем каталоге):

```
$ python hello_world.py
Hello world
```

Многие программисты выполняют свой код на Python именно таким образом, но в мире *научных* приложений и анализа данных принято использовать IPython, улучшенный и дополненный интерпретатор Python, или веб-блокноты Jupyter, первоначально разработанные как часть проекта IPython. Введение в IPython и Jupyter будет дано в этой главе, а углубленное описание возможностей IPython – в приложении 3. С помощью команды `%run` IPython исполняет код в указанном файле в том же процессе, что позволяет интерактивно изучать результаты по завершении выполнения.

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

По умолчанию приглашение IPython содержит не стандартную строку `>>>`, а строку вида `In [2]:`, включающую порядковый номер предложения.

2.2. Основы IPython

В этом разделе мы научимся запускать оболочку IPython и Jupyter-блокнот, а также познакомимся с некоторыми важнейшими понятиями.

Запуск оболочки IPython

IPython можно запустить из командной строки, как и стандартный интерпретатор Python, только для этого служит команда `ipython`:

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
```

```
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

In [2]: a
Out[2]: 5
```

Чтобы выполнить произвольное предложение Python, нужно ввести его и нажать клавишу **Enter**. Если ввести только имя переменной, то IPython выведет строковое представление объекта:

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(7)}

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1:  0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4:  1.9657805725027142,
 5:  1.3934058329729904,
 6:  0.09290787674371767}
```

Здесь первые две строки содержат код на Python; во второй строке создается переменная **data**, ссылающаяся на только что созданный словарь Python. В последней строке значение **data** выводится на консоль.

Многие объекты Python форматируются для удобства чтения; такая *красивая печать* отличается от обычного представления методом `print`. Тот же словарь, напечатанный в стандартном интерпретаторе Python, выглядел бы куда менее презентабельно:

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print data
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
6: 0.3308507317325902}
```

IPython предоставляет также средства для исполнения произвольных блоков кода (путем копирования и вставки) и целых Python-скриптов. Эти вопросы будут рассмотрены чуть ниже.

Запуск Jupyter-блокнота

Одним из основных компонентов Jupyter-проекта является *блокнот* – интерактивный документ, содержащий код, текст (простой или размеченный), визуализации и другие результаты выполнения кода. Jupyter-блокнот взаимодействует с *ядрами* – реализациями протокола интерактивных вычислений на

различных языках программирования. В ядре Jupyter для Python в качестве основы используется IPython.

Для запуска Jupyter выполните в терминале команду `jupyter notebook`:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

На многих платформах Jupyter автоматически открывается в браузере по умолчанию (если только при запуске не был указан флаг `--no-browser`). Если это не так, то можете сами ввести URL при запуске блокнота, в данном случае `http://localhost:8888/`. На рис. 12.1 показано, как выглядит блокнот в браузере Google Chrome.

Многие используют Jupyter в качестве локальной среды вычислений, но его можно также развернуть на сервере и работать с ним удаленно. Я не буду вдаваться в детали, при необходимости вы сможете найти информацию в интернете.

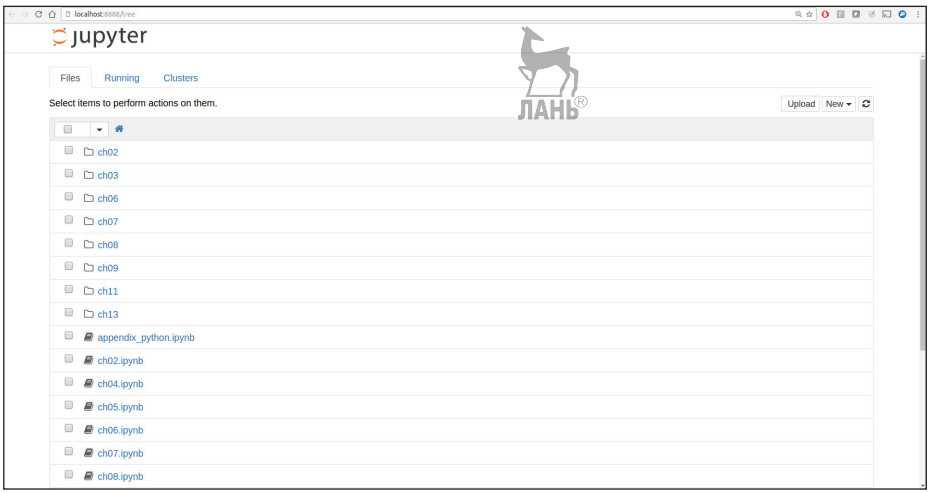


Рис. 2.1. Начальная страница Jupyter-блокнота

Для создания нового блокнота нажмите кнопку **New** и выберите «Python 3» или «conda [default]». На экране появится окно, показанное на рис. 2.2. Если вы здесь впервые, попробуйте щелкнуть по пустой ячейке кода и ввести строку кода на Python. Для выполнения нажмите **Shift**-**Enter**.

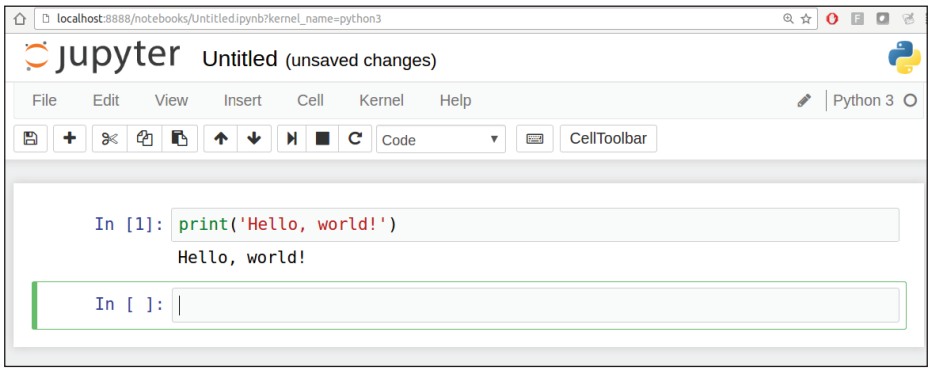


Рис. 2.2. Так выглядит новый Jupyter-блокнот

После сохранения блокнота (команда **Save and Checkpoint** в меню **File**) будет создан файл с расширением *.ipynb*. В нем содержится все, что сейчас находится в блокноте (включая все результаты выполнения кода). Чтобы загрузить имеющийся блокнот, поместите файл в тот каталог, из которого был запущен блокнот, или в его подкаталог и дважды щелкните по имени файла на начальной странице. Можете попробовать проделать это с моими блокнотами, находящимися в репозитории *wesm/pydata-book* на GitHub (рис. 2.3).

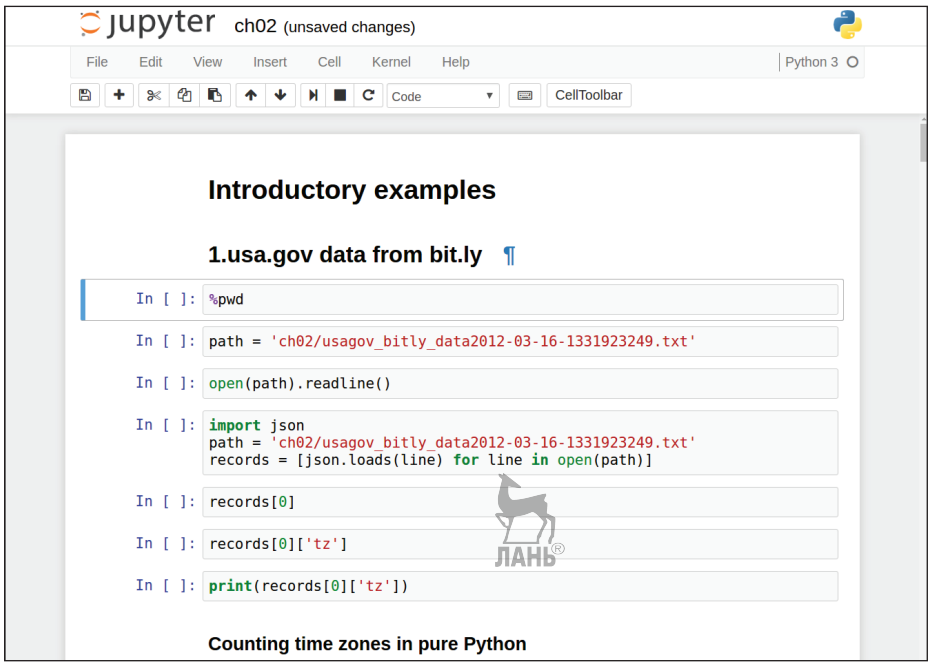


Рис. 2.3. Пример существующего Jupyter-блокнота

Хотя может показаться, что работа с Jupyter-блокнотами отличается от работы в оболочке IPython, на самом деле почти все описанные в этой главе команды и инструменты можно использовать в обеих средах.

Завершение по нажатии клавиши Tab

На первый взгляд оболочка IPython очень похожа на стандартный интерпретатор Python (вызываемый командой **python**) с мелкими косметическими изменениями. Одно из существенных преимуществ над стандартной оболочкой Python – *завершение по нажатии клавиши **Tab***, реализованное в большинстве IDE и других средах интерактивных вычислений. Если во время ввода выражения нажать <Tab>, то оболочка произведет поиск в пространстве имен всех переменных (объектов, функций и т. д.), имена которых начинаются с введенной к этому моменту строки:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple and an_example any
```

Обратите внимание, что IPython вывел обе определенные выше переменные, а также ключевое слово Python and и встроенную функцию any. Естественно, можно также завершать имена методов и атрибутов любого объекта, если предварительно ввести точку:

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append b.extend b.insert b.remove b.sort
b.count b.index b.pop b.reverse
```

То же самое относится и к модулям:

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date           datetime.MAXYEAR      datetime.timedelta
datetime.datetime       datetime.MINYEAR      datetime.tzinfo
datetime.datetime_CAPI  datetime.time
```

В Jupyter-блокноте и новых версиях IPython (5.0 и старше) варианты автозавершения отображаются не в выпадающем окне, а в текстовом виде.

Отметим, что IPython по умолчанию скрывает методы и атрибуты, начинающиеся знаком подчеркивания, например магические методы и внутренние «закрытые» методы и атрибуты, чтобы не загромождать экран (и не смущать неопытных пользователей). На них автозавершение также распространяется, нужно только сначала набрать знак подчеркивания. Если вы предпочитаете всегда видеть такие методы при автозавершении, измените соответствующий режим в конфигурационном файле IPython. О том, как это сделать, см. документацию по IPython.

Завершение по нажатии **Tab** работает во многих контекстах, помимо поиска в интерактивном пространстве имен и завершения атрибутов объекта или модуля. Если нажать <Tab> при вводе чего-то, похожего на путь к файлу (даже внутри строки Python), то будет произведен поиск в файловой системе:

```
In [7]: datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/README
datasets/movielens/ratings.dat   datasets/movielens/users.dat

In [7]: path = 'datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/README
datasets/movielens/ratings.dat   datasets/movielens/users.dat
```

В сочетании с командой %run (см. ниже) эта функция несомненно позволит вам меньше лупить по клавиатуре.

Автозавершение позволяет также сэкономить время при вводе именованных аргументов функции (в том числе самого знака =). См. рис. 2.4.

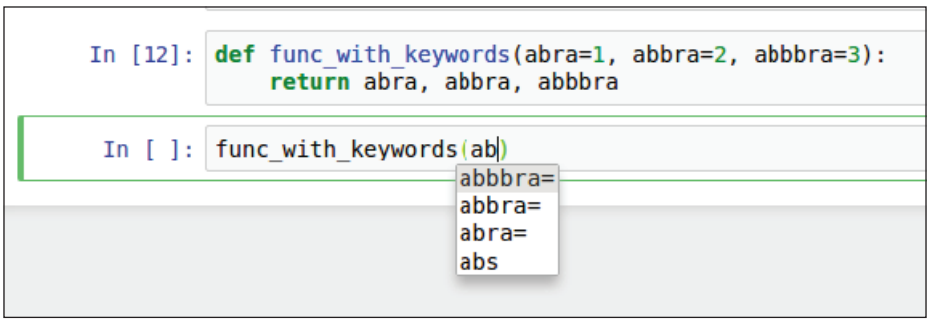


Рис. 2.4. Автозавершение именованных аргументов функции в Jupyter-блокноте

Ниже мы еще поговорим о функциях.

Интроспекция

Если ввести вопросительный знак (?) до или после имени переменной, то будет напечатана общая информация об объекте:

```
In [8]: b = [1, 2, 3]

In [9]: b?
Type:       list
String Form:[1, 2, 3]
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

In [10]: print?
```

```
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:  builtin_function_or_method
```

Это называется *интроспекцией объекта*. Если объект представляет собой функцию или метод экземпляра, то будет показана строка документации, если она существует. Допустим, мы написали такую функцию (этот код можно ввести в IPython или Jupyter):

```
def add_numbers(a, b):
    """
    Сложить два числа

    Возвращает
    ----------
    the_sum : тип аргументов
    """
    return a + b
```

Тогда при вводе знака ? мы увидим строку документации:

```
In [11]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Сложить два числа

Возвращает
----------
the_sum : type of arguments
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

Два вопросительных знака ?? покажут также исходный код функции, если это возможно:

```
In [12]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Сложить два числа

    Возвращает
    ----------
    the_sum : тип аргументов
```

```
    """
    return a + b
File:        book_scripts/<ipython-input-546-5473012eeb65>
Type:        function
```

И последнее применение ? – поиск в пространстве имен IPython по аналогии со стандартной командной строкой UNIX или Windows. Если ввести несколько символов в сочетании с метасимволом *, то будут показаны все имена по указанной маске. Например, вот как можно получить список всех функций в пространстве имен верхнего уровня NumPy, имена которых содержат строку load:

```
In [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

Команда %run

Команда %run позволяет выполнить любой файл как Python-программу в контексте текущего сеанса IPython. Предположим, что в файле *ipython_script_test.py* хранится такой простенький скрипт:

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

Этот скрипт можно выполнить, передав имя файла команде %run:

```
In [14]: %run ipython_script_test.py
```

Скрипт выполняется в *пустом пространстве имен* (в которое ничего не импортировано и в котором не определены никакие переменные), поэтому его поведение должно быть идентично тому, что получается при запуске программы из командной строки командой python script.py. Все переменные (импортированные, функции, глобальные объекты), определенные в файле (до момента исключения, если таковое произойдет), будут доступны оболочке IPython:

```
In [15]: c
Out[15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

Если Python-скрипт ожидает передачи аргументов из командной строки (которые должны попасть в массив `sys.argv`), то их можно перечислить после пути к файлу, как в командной строке.

Если вы хотите дать скрипту доступ к переменным, уже определенным в интерактивном пространстве имен IPython, используйте команду `%run -i`, а не просто `%run`.

В Jupyter-блокноте можно также использовать магическую функцию `%load`, которая импортирует скрипт в ячейку кода:

```
>>> %load ipython_script_test.py

    def f(x, y, z):
        return (x + y) / z

    a = 5
    b = 6
    c = 7.5
    result = f(a, b, c)
```

Прерывание выполняемой программы

Нажатие **Ctrl**+**C** во время выполнения кода, запущенного с помощью `%run` или просто долго работающей программы, приводит к возбуждению исключения `KeyboardInterrupt`. В этом случае почти все Python-программы немедленно прекращают работу, если только не возникло очень редкое стечение обстоятельств.

Если Python-код вызвал откомпилированный модуль расширения, то нажатие **Ctrl**+**C** не всегда приводит к немедленному завершению. В таких случаях нужно либо дождаться возврата управления интерпретатору Python, либо (если случилось что-то ужасное) принудительно снять процесс Python.

Исполнение кода из буфера обмена

В Jupyter-блокноте можно скопировать код в любую ячейку и выполнить его. В оболочке IPython также можно выполнять код, находящийся в буфере обмена. Предположим, что в каком-то другом приложении имеется такой код:

```
x = 5
y = 7
if x > 5:
   x += 1

   y = 8
```

Проще всего воспользоваться магическими функциями `%paste` и `%cpaste`. Функция `%paste` берет текст, находящийся в буфере обмена, и выполняет его в оболочке как единый блок:

```
In [17]: %paste
x = 5
y = 7
```

```
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

Функция %cpaste аналогична, но выводит специальное приглашение для вставки кода:

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:   x += 1
:
:   y = 8
:--
```

При использовании %cpaste вы можете вставить сколько угодно кода, перед тем как начать его выполнение. Например, %cpaste может пригодиться, если вы хотите посмотреть на вставленный код до выполнения. Если окажется, что случайно вставлен не тот код, то из %cpaste можно выйти нажатием **Ctrl**+**C**.

Комбинации клавиш

В IPython есть много комбинаций клавиш для навигации по командной строке (они знакомы пользователям текстового редактора Emacs или оболочки UNIX bash) и взаимодействия с историей команд (см. следующий раздел). В табл. 2.1 перечислены наиболее употребительные комбинации, а на рис. 2.5 некоторые из них, например перемещение курсора, проиллюстрированы.

Таблица 2.1. Стандартные комбинации клавиш IPython

Комбинация клавиш	Описание
Ctrl+P или ↑	Просматривать историю команд назад в поисках команд, начинающихся с введенной строки
Ctrl+N или ↓	Просматривать историю команд вперед в поисках команд, начинающихся с введенной строки
Ctrl+R	Обратный поиск в истории в духе Readline (частичное соответствие)
Ctrl+Shift+V	Вставить текст из буфера обмена
Ctrl+C	Прервать исполнение программы
Ctrl+A	Переместить курсор в начало строки
Ctrl+E	Переместить курсор в конец строки
Ctrl+K	Удалить текст от курсора до конца строки
Ctrl+U	Отбросить весь текст в текущей строке
Ctrl+F	Переместить курсор на один символ вперед
Ctrl+B	Переместить курсор на один символ назад
Ctrl+L	Очистить экран

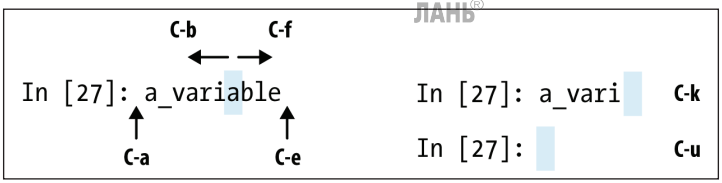


Рис. 2.5. Иллюстрация некоторых комбинаций клавиш IPython

Отметим, что в Jupyter-блокнотах для навигации и редактирования применяются совершенно другие комбинации клавиш. Поскольку изменения происходят быстрее, чем в IPython, я рекомендую воспользоваться встроенной в Jupyter справкой.

О магических командах

В IPython есть много специальных команд, называемых магическими, цель которых – упростить решение типичных задач и облегчить контроль над поведением всей системы IPython. Магической называется команда, которой предшествует знак процента `%`. Например, магическая функция `%timeit` (мы подробно рассмотрим ее ниже) позволяет замерить время выполнения любого предложения Python, например умножения матриц:

```
In [20]: a = np.random.randn(100, 100)

In [20]: %timeit np.dot(a, a)
10000 loops, best of 3: 20.9 us per loop
```

Магические команды можно рассматривать как командные утилиты, исполняемые внутри IPython. У многих из них имеются дополнительные параметры командной строки, список которых можно распечатать с помощью `?` (вы ведь так и думали, правда?)[1]:

```
In [21]: %debug?
Docstring:
::

    %debug [--breakpoint FILE:LINE] [statement [statement ...]]

Активировать интерактивный отладчик.

Эта магическая команда поддерживает два способа активации отладчика.
Первый - активировать отладчик до выполнения кода. Тогда вы сможете
поставить точку прерывания и пошагово выполнять код, начиная с этой
точки. В данном режиме команде передаются подлежащие выполнению
предложения и необязательная точка прерывания.
```

[1] Сообщения выводятся на английском языке, но для удобства читателя переведены. – *Прим. перев.*

```
Второй способ – активировать отладчик в постоперационном режиме, для
чего нужно просто выполнить команду %debug без аргументов. В случае
исключения это позволит интерактивно просматривать кадры стека. Отметим,
что в таком случае всегда используется последняя трасса стека, так что
анализировать ее надо сразу после возникновения исключения, поскольку
следующее исключение затрет предыдущее.

Если вы хотите, чтобы IPython автоматически делал это при каждом
исключении, то обратитесь к документации по магической команде %pdb.

позиционные аргументы:
  statement         Код, подлежащий выполнению в отладчике. При работе
                    в режиме ячейки можно опускать.

факультативные аргументы:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                    Установить точку прерывания на строке LINE файла
                    FILE.
```

Магические функции по умолчанию можно использовать и без знака процента, если только нигде не определена переменная с таким же именем, как у магической функции. Этот режим называется *автомагическим*, его можно включить или выключить с помощью функции `%automagic`.

Некоторые магические функции ведут себя как функции Python, и их результат можно присваивать переменной:

```
In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book'

In [23]: foo = %pwd

In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'
```

Поскольку к документации по IPython легко можно обратиться из системы, я рекомендую изучить все имеющиеся специальные команды, набрав `%quickref` или `%magic`. В табл. 2.2 описаны некоторые команды, наиболее важные для продуктивной работы в области интерактивных вычислений и разработки в среде IPython.

Таблица 2.2. Часто используемые магические команды IPython

Команда	Описание
`%quickref`	Вывести краткую справку по IPython
`%magic`	Вывести подробную документацию по всем имеющимся магическим командам
`%debug`	Войти в интерактивный отладчик в точке последнего вызова, показанного в обратной трассировке исключения
`%hist`	Напечатать историю введенных команд (по желанию вместе с результатами)
`%pdb`	Автоматически входить в отладчик после любого исключения
`%paste`	Выполнить отформатированный Python-код, находящийся в буфере обмена

Таблица 2.2 (окончание)

Команда	Описание
`%cpaste`	Открыть специальное приглашение для ручной вставки Python-кода, подлежащего выполнению
`%reset`	Удалить все переменные и прочие имена, определенные в интерактивном пространстве имен
`%page` *OBJECT*	Сформировать красиво отформатированное представление объекта и вывести его постранично
`%run` *script.py*	Выполнить Python-скрипт из IPython
`%prun` *предложение*	Выполнить *предложение* под управлением cProfile и вывести результаты профилирования
`%time` *предложение*	Показать время выполнения одного предложения
`%timeit` *предложение*	Выполнить предложение несколько раз и усреднить время выполнения. Полезно для хронометража кода, который выполняется очень быстро
`%who`, `%who_ls`, `%whos`	Вывести переменные, определенные в интерактивном пространстве имен, с различной степенью детализации
`%xdel` *переменная*	Удалить переменную и попытаться очистить все ссылки на объект во внутренних структурах данных IPython

Интеграция с matplotlib

IPython так популярен в сообществе анализа данных отчасти потому, что он хорошо интегрируется с библиотеками визуализации данных и организации графического интерфейса типа matplotlib. Если вы раньше никогда не работали с matplotlib, ничего страшного; ниже мы обсудим данную библиотеку во всех подробностях. Магическая функция `%matplotlib` задает способ ее интеграции с оболочкой IPython или Jupyter-блокнотом. Это важно, поскольку в противном случае созданные вами графики либо вообще не будут видны (блокнот), либо захватят управление сеансом до его завершения (оболочка).

В оболочке IPython команда `%matplotlib` настраивает интеграцию, так чтобы можно было создавать несколько окон графиков, не мешая консольному сеансу:

```
In [26]: %matplotlib
Using matplotlib backend: Qt4Agg
```

В Jupyter команда выглядит немного иначе (рис. 2.6):

```
In [26]: %matplotlib inline
```

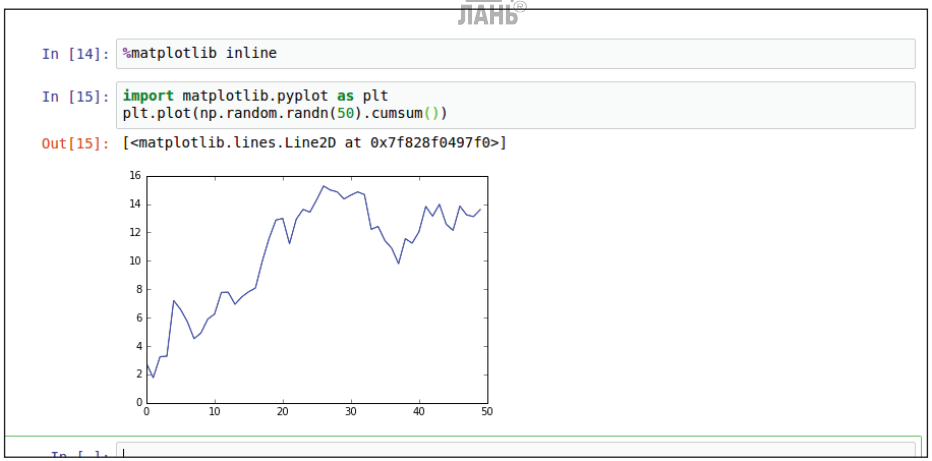


Рис. 2.6. Графики matplotlib, встроенные в окно Jupyter

2.3. Основы языка Python

В этом разделе я сделаю обзор наиболее важных концепций программирования на Python и механизмов языка. В следующей главе более подробно рассмотрим структуры данных, функции и другие средства Python.

Семантика языка

Язык Python отличается удобочитаемостью, простотой и ясностью. Некоторые даже называют написанный на Python код исполняемым псевдокодом.

Отступы вместо скобок

В Python для структурирования кода используются пробелы (или знаки табуляции), а не фигурные скобки, как во многих других языках, например R, C++, Java и Perl. Вот как выглядит цикл в алгоритме быстрой сортировки:

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

Двоеточием обозначается начало блока кода с отступом, весь последующий код до конца блока должен быть набран с точно таким же отступом.

Нравится вам это или нет, но синтаксически значимые пробелы – факт, с которым программисты на Python должны смириться, однако по собственному опыту могу сказать, что благодаря им код на Python выглядит гораздо более удобочитаемым, чем на других знакомых мне языках. Поначалу такой стиль может показаться чужеродным, но со временем вы привыкнете и полюбите его.

Я настоятельно рекомендую использовать *четыре пробела* в качестве величины отступа по умолчанию и настроить редактор так, чтобы он заменял знаки табуляции четырьмя пробелами. Некоторые используют знаки табуляции непосредственно или задают другое число пробелов – например, довольно часто встречаются отступы шириной в два пробела. Но четыре пробела – соглашение, принятое подавляющим большинством программистов на Python, и я советую его придерживаться, если нет каких-то противопоказаний.

Вы уже поняли, что предложения в Python не обязаны завершаться точкой с запятой. Но ее можно использовать, чтобы отделить друг от друга предложения, находящиеся в одной строчке:

```
a = 5; b = 6; c = 7
```


Впрочем, писать несколько предложений в одной строчке не рекомендуется, потому что код из-за этого труднее читается.

Всё является объектом

Важная характеристика языка Python – последовательность его *объектной модели*. Все числа, строки, структуры данных, функции, классы, модули и т. д. в интерпретаторе заключены в «ящики», которые называются *объектами Python*. С каждым объектом ассоциирован *тип* (например, *строка* или *функция*) и внутренние данные. На практике это делает язык более гибким, потому что даже функции можно рассматривать как объекты.

Комментарии

Интерпретатор Python игнорирует текст, которому предшествует знак решетки #. Часто этим пользуются, чтобы включить в код комментарии. Иногда желательно исключить какие-то блоки кода, не удаляя их. Самое простое решение – *закомментировать* такой код:


```
results = []
for line in file_handle:
    # пока оставляем пустые строки
    # if len(line) == 0:
    #   continue
    results.append(line.replace('foo', 'bar'))
```

Комментарий может располагаться также в конце строки с исполняемым кодом. Иногда это полезно, хотя некоторые программисты предпочитают помещать комментарий в строке, предшествующей исполняемому коду:

```
print("Дошли до этой строки") # простой отчет о состоянии
```

Вызов функции и метода объекта

После имени функции ставятся круглые скобки, внутри которых размещено ноль или более параметров. Возвращенное значение может быть присвоено переменной:

```
result = f(x, y, z)
g()
```

Почти со всеми объектами в Python ассоциированы функции, которые имеют доступ к внутреннему состоянию объекта и называются *методами*. Синтаксически вызов методов выглядит так:

```
obj.some_method(x, y, z)
```

Функции могут принимать *позиционные* и *именованные* аргументы:

```
result = f(a, b, c, d=5, e='foo')
```

Подробнее об этом ниже.

Переменные и передача аргументов

Присваивание значения переменной (или *имени*) в Python приводит к созданию *ссылки* на объект, стоящий в правой части присваивания. Рассмотрим список целых чисел:

```
In [8]: a = [1, 2, 3]
```

Предположим, что мы присвоили значение `a` новой переменной `b`:

```
In [8]: b = a
```

В некоторых языках такое присваивание приводит к копированию данных `[1, 2, 3]`. В Python `a` и `b` указывают на один и тот же объект – исходный список `[1, 2, 3]` (схематически изображено на рис. 2.7). Чтобы убедиться в этом, добавим в список `a` еще один элемент и проверим список `b`:

```
In [10]: a.append(4)

In [11]: b
Out[11]: [1, 2, 3, 4]
```

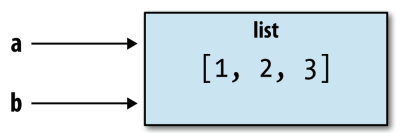


Рис. 2.7. Две ссылки на один объект

Понимать семантику ссылок в Python и знать, когда, как и почему данные копируются, особенно важно при работе с большими наборами данных.

Операцию присваивания называют также *связыванием*, потому что мы связываем имя с объектом. Имена переменных, которым присвоено значение, иногда называют связанными переменными.

Когда объекты передаются функции в качестве аргументов, создаются новые локальные переменные, ссылающиеся на исходные объекты, – копирова-

ние не производится. Если новый объект связывается с переменной внутри функции, его изменение никак не отражается на родительской области видимости. Следовательно, функция может модифицировать внутреннее содержимое изменяемого аргумента. Пусть имеется такая функция:

```
def append_element(some_list, element):
    some_list.append(element)
```

Тогда:

```
In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[30]: [1, 2, 3, 4]
```

Динамические ссылки, строгие типы

В отличие от многих компилируемых языков, в частности Java и C++, со *ссылкой* на объект в Python не связан никакой тип. Следующий код не приведет к ошибке:


```
In [12]: a = 5

In [13]: type(a)
Out[13]: int

In [14]: a = 'foo'

In [15]: type(a)
Out[15]: str
```

Переменные – это имена объектов в некотором пространстве имен; информация о типе хранится в самом объекте. Таким образом, некоторые делают поспешный вывод, будто Python не является «типизированным языком». Это не так, рассмотрим следующий пример:

```
In [16]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-249-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: must be str, not int
```


В некоторых языках, например в Visual Basic, строка `'5'` могла бы быть неявно преобразована (*приведена*) в целое число, и это выражение было бы вычислено как 10. А бывают и такие языки, например JavaScript, где целое число `5` преобразуется в строку, после чего производится конкатенация – `'55'`. В этом отношении Python считается *строго типизированным* языком, т. е. у любого объекта есть конкретный тип (или *класс*), а неявные преобразования разрешены только в некоторых не вызывающих сомнений случаях, например:

```
In [17]: a = 4.5

In [18]: b = 2

# Форматирование строки, см. ниже
In [19]: print 'a is %s, b is %s' % (type(a), type(b))
a is <type 'float'>, b is <type 'int'>

In [20]: a / b
Out[20]: 2.25
```

Знать тип объекта важно, и полезно также уметь писать функции, способные обрабатывать входные параметры различных типов. Проверить, является ли объект экземпляром определенного типа, позволяет функция `isinstance`:

```
In [21]: a = 5
In [22]: isinstance(a, int)
Out[22]: True
```

Функция `isinstance` может также принимать кортеж типов. Тогда она проверяет, присутствует ли в кортеже тип переданного объекта:

```
In [23]: a = 5; b = 4.5

In [24]: isinstance(a, (int, float))
Out[24]: True

In [25]: isinstance(b, (int, float))
Out[25]: True
```

Атрибуты и методы

Объекты в Python обычно имеют атрибуты – другие объекты, хранящиеся «внутри» данного, и методы – ассоциированные с объектом функции, имеющие доступ к внутреннему состоянию объекта. Обращение к тем и другим синтаксически выглядит как *obj.attribute_name*:

```
In [1]: a = 'foo'

In [2]: a.<Tab>
a.capitalize  a.format    a.isupper     a.rindex       a.strip
a.center      a.index     a.join        a.rjust        a.swapcase
a.count       a.isalnum   a.ljust       a.rpartition   a.title
a.decode      a.isalpha   a.lower       a.rsplit       a.translate
a.encode      a.isdigit   a.lstrip      a.rstrip       a.upper
a.endswith    a.islower   a.partition   a.split        a.zfill
a.expandtabs  a.isspace   a.replace     a.splitlines
a.find        a.istitle   a.rfind       a.startswith
```

К атрибутам и методам можно обращаться также с помощью функции `getattr`:

```
In [27]: getattr(a, 'split')
Out[27]: <function str.split>
```

В других языках доступ к объектам по имени иногда называется отражением. Хотя в этой книге мы почти не используем функцию `getattr`, а также родственные ей `hasattr` и `setattr`, они весьма эффективны для написания обобщенного, повторно используемого кода.

Динамическая типизация

Часто нас интересует не тип объекта, а лишь наличие у него определенных методов или поведения. Иногда это называют динамической, или «утиной», типизацией, имея в виду поговорку «если кто-то ходит как утка и крякает как утка, то это утка и есть». Например, объект поддерживает итерирование, если он реализует *протокол итератора*. Для многих объектов это означает, что имеется «магический метод» `__iter__`, хотя есть другой – и лучший – способ проверки: попробовать воспользоваться функцией `iter`:


```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # не является итерируемым
        return False
```

Эта функция возвращает `True` для строк, а также для большинства типов коллекций в Python:

```
In [29]: isiterable('a string')
Out[29]: True

In [30]: isiterable([1, 2, 3])
Out[30]: True

In [31]: isiterable(5)
Out[31]: False
```

Эту функциональность я постоянно использую для написания функций, принимающих параметры разных типов. Типичный случай – функция, принимающая любую последовательность (список, кортеж, ndarray) или даже итератор. Можно сначала проверить, является ли объект списком (или массивом NumPy), и если нет, то преобразовать его в таковой:


```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

Импорт

В Python *модуль* – это просто файл с расширением `.py`, который содержит функции и различные определения, в том числе импортированные из других `py`-файлов. Пусть имеется следующий модуль:

```
# some_module.py
PI = 3.14159
```

```
def f(x):
    return x + 2

def g(a, b):
    return a + b
```

Если бы мы захотели обратиться к переменным или функциям, определенным в *some_module.py*, из другого файла в том же каталоге, то должны были бы написать:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Или эквивалентно:

```
from some_module import f, g, PI
result = g(5, PI)
```

Ключевое слово as позволяет переименовать импортированные сущности:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Бинарные операторы и операции сравнения

Большинство математических операций и операций сравнения именно таково, как мы и ожидаем:

```
In [32]: 5 - 7
Out[32]: -2

In [33]: 12 + 21.5
Out[33]: 33.5

In [34]: 5 <= 2
Out[34]: False
```

Список всех бинарных операторов приведен в табл. 2.3.

Для проверки, ведут ли две ссылки на один и тот же объект, служит оператор is. Оператор is not тоже существует, он позволяет убедиться, что два объекта различаются.

```
In [35]: a = [1, 2, 3]

In [36]: b = a

In [37]: c = list(a)

In [38]: a is b
Out[38]: True

In [39]: a is not c
Out[39]: True
```

Таблица 2.3. Бинарные операторы

Операция	Описание
a + b	Сложить a и b
a - b	Вычесть b из a
a * b	Умножить a на b
a / b	Разделить a на b
a // b	Разделить a на b нацело, отбросив дробный остаток
a ** b	Возвести a в степень b
a & b	True, если и a, и b равны True. Для целых чисел вычисляет поразрядное И
a | b	True, либо a, либо b равно True. Для целых чисел вычисляет поразрядное ИЛИ
a ^ b	Для булевых величин True, если либо a, либо b, но не обе одновременно равны True. Для целых чисел вычисляет поразрядное ИСКЛЮЧИТЕЛЬНОЕ ИЛИ
a == b	True, если a равно b
a != b	True, если a не равно b
a < b, a <= b	True, если a меньше (меньше или равно) b
a > b, a >= b	True, если a больше (больше или равно) b
a is b	True, если a и b ссылаются на один и тот же объект Python
a is not b	True, если a и b ссылаются на разные объекты Python

Поскольку функция list всегда создает новый список Python (т. е. копию исходного), то имеется уверенность, что c и a – различные объекты. Сравнение с помощью оператора is не то же самое, что с помощью оператора ==, потому что в данном случае мы получим:

```
In [40]: a == c
Out[40]: True
```

Операторы is и is not очень часто употребляются, чтобы проверить, равна ли некоторая переменная None, потому что существует ровно один экземпляр None:

```
In [41]: a = None

In [42]: a is None
Out[42]: True
```

Изменяемые и неизменяемые объекты

Большинство объектов в Python: списки, словари, массивы NumPy и почти все определенные пользователем типы (классы) – изменяемо. Это означает, что объект или значения, которые в нем хранятся, можно модифицировать.

```
In [43]: a_list = ['foo', 2, [4, 5]]

In [44]: a_list[2] = (3, 4)

In [45]: a_list
Out[45]: ['foo', 2, (3, 4)]
```

Но некоторые объекты, например строки и кортежи, неизменяемы:

```
In [46]: a_tuple = (3, 5, (4, 5))
In [47]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-47-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Помните, что *возможность* изменять объект не означает *необходимости* это делать. Подобные действия в программировании называются *побочными эффектами*. Когда вы пишете функцию, обо всех ее побочных эффектах следует сообщать пользователю в комментариях или в документации. Я рекомендую по возможности избегать побочных эффектов, *отдавая предпочтение неизменяемости*, даже если вы работаете с изменяемыми объектами.

Скалярные типы

В Python есть небольшой набор встроенных типов для работы с числовыми данными, строками, булевыми значениями (`True` и `False`), датами и временем. Эти типы «с одним значением» иногда называются *скалярными*, а мы будем называть их просто скалярами. Перечень основных скалярных типов приведен в табл. 2.4. Работа с датами и временем будет рассмотрена отдельно, потому что эти типы определены в стандартном модуле `datetime`.

Таблица 2.4. Стандартные скалярные типы в Python

Тип	Описание
`None`	Значение «null» в Python (существует только один экземпляр объекта `None`)
`str`	Тип строки. Может содержать любые символы Unicode
`bytes`	Неинтерпретируемые ASCII-байты (или символы Unicode, закодированные последовательностями байтов)
`float`	Число с плавающей точкой двойной точности (64-разрядное). Отдельный тип `double` не предусмотрен
`bool`	Значение `True` или `False`
`int`	Целое со знаком, максимальное значение зависит от платформы

Числовые типы

Основные числовые типы в Python – `int` и `float`. Тип `int` способен представить сколь угодно большое целое число.

```
In [48]: ival = 17239871

In [49]: ival ** 6
Out[49]: 26254519291092456596965462913230729701102721
```

Числа с плавающей точкой представляются типом Python `float`, который реализован в виде значения двойной точности (64-разрядного). Такие числа можно записывать и в научной нотации:

```
In [50]: fval = 7.243

In [51]: fval2 = 6.78e-5
```

Деление целых чисел, результатом которого не является целое число, всегда дает число с плавающей точкой:

```
In [52]: 3 / 2
Out[52]: 1.5
```

Для выполнения целочисленного деления в духе языка C (когда дробная часть результата отбрасывается) служит оператор деления с отбрасыванием //:


```
In [53]: 3 // 2
Out[53]: 1
```

Строки

Многие любят Python за его мощные и гибкие средства работы со строками. *Строковый литерал* записывается в одиночных (') или двойных (") кавычках:

```
a = 'one way of writing a string'
b = "another way"
```

Для записи многострочных строк, содержащих разрывы, используются тройные кавычки – ''' или """:

```
c = """
Это длинная строка,
занимающая несколько строчек
"""
```

Возможно, вы удивитесь, узнав, что строка c в действительности содержит четыре строчки текста: разрывы строки после """ и после слова строчек являются частью строки. Для подсчета количества знаков новой строки можно воспользоваться методом count объекта c:

```
In [55]: c.count('\n')
Out[55]: 3
```

Строки в Python неизменяемы, при любой модификации создается новая строка:


```
In [56]: a = 'this is a string'
In [57]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [58]: b = a.replace('string', 'longer string')
```

```
In [59]: b
Out[59]: 'this is a longer string'
```

После этой операции переменная a не изменилась:

```
In [60]: a
Out[60]: 'this is a string'
```

Многие объекты Python можно преобразовать в строку с помощью функции str:

```
In [61]: a = 5.6

In [62]: s = str(a)

In [63]: print(s)
5.6
```

Строки – это последовательности символов Unicode и потому могут рассматриваться как любые другие последовательности, например списки или кортежи (которые будут подробно рассмотрены в следующей главе):

```
In [64]: s = 'python'

In [65]: list(s)
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']

In [66]: s[:3]
Out[66]: 'pyt'
```

Синтаксическая конструкция s[:3] называется *срезом* и реализована для многих типов последовательностей в Python. Позже мы подробно объясним, как она работает, поскольку будем часто использовать ее в книге.

Знак обратной косой черты \ играет роль *управляющего символа*, он предшествует специальным символам, например знаку новой строки или символам Unicode. Чтобы записать строковый литерал, содержащий знак обратной косой черты, этот знак необходимо повторить дважды:

```
In [67]: s = '12\\34'

In [68]: print s
12\34
```

Если строка содержит много знаков обратной косой черты и ни одного специального символа, то при такой записи она становится совершенно неразборчивой. По счастью, есть другой способ: поставить перед начальной кавычкой букву r, которая означает, что все символы должны интерпретироваться буквально:

```
In [69]: s = r'this\has\no\special\characters'

In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

Буква r здесь сокращение от *raw*.

Сложение двух строк означает конкатенацию, при этом создается новая строка:

```
In [71]: a = 'this is the first half '

In [72]: b = 'and this is the second half'

In [73]: a + b
Out[73]: 'this is the first half and this is the second half'
```

Еще одна важная тема – форматирование строк. С появлением Python 3 диапазон возможностей в этом плане расширился, здесь я лишь вкратце опишу один из основных интерфейсов. У строковых объектов имеется метод `format`, который можно использовать для подстановки в строку отформатированных аргументов, в результате чего рождается новая строка:

```
In [74]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

Здесь:

- `{0:.2f}` означает, что первый аргумент нужно отформатировать как число с плавающей точкой с двумя знаками после точки;
- `{1:s}` означает, что второй аргумент нужно отформатировать как строку;
- `{2:d}` означает, что третий аргумент нужно отформатировать как целое число.

Для подстановки значений вместо спецификаторов формата мы передаем методу `format` последовательность аргументов:

```
In [75]: template % (4.5560, 'Argentine Pesos', 1)
Out[75]: '4.56 Argentine Pesos are worth US$1'
```

Форматирование строк – обширная тема. Существует несколько методов и многочисленные параметры и ухищрения, призванные контролировать, как именно форматируются значения, подставляемые в результирующую строку. Подробные сведения можно найти в официальной документации по Python (https://docs.python.org/3.6/library/string.html).

В главе 8 обсуждаются общие вопросы работы со строками в контексте анализа данных.

Байты и Unicode

В современном Python (т. е. Python 3.0 и выше) Unicode стал полноправным типом строки, обеспечивающим единообразную обработку любых текстов, а не только в кодировке ASCII. В прежних версиях строка рассматривалась как совокупность байтов без явного предположения о кодировке Unicode. Строку можно было преобразовать в Unicode, если была известна кодировка символов. Рассмотрим пример:

```
In [76]: val = "español"

In [77]: val
Out[77]: 'español'
```

Мы можем преобразовать эту Unicode-строку в последовательность байтов в кодировке UTF-8, вызвав метод encode:

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

В предположении, что известна Unicode-кодировка объекта bytes, мы можем обратить эту операцию методом decode:

```
In [81]: val_utf8.decode('utf-8')
Out[81]: 'español'
```

Хотя в наши дни обычно используют кодировку UTF-8 для любых текстов, в силу исторических причин иногда можно встретить данные и в других кодировках:

```
In [82]: val.encode('latin1')
Out[82]: b'espa\xf1ol'

In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

Чаще всего объекты типа bytes встречаются при работе с файлами, когда неявно перекодировать все данные в Unicode-строки нежелательно.

Несмотря на то что нужда в этом возникает редко, при необходимости можно определить байтовые литералы, добавив к строке префикс b:

```
In [85]: bytes_val = b'this is bytes'

In [86]: bytes_val
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded # теперь это объект типа str (Unicode)
Out[88]: 'this is bytes'
```

Булевы значения

Два булевых значения записываются в Python как True и False. Результатом сравнения и вычисления условных выражений является True или False. Булевы значения объединяются с помощью ключевых слов and и or:

```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

Приведение типов

Типы str, bool, int и float являются также функциями, которые можно использовать для приведения значения к соответствующему типу:

```
In [91]: s = '3.14159'

In [92]: fval = float(s)

In [93]: type(fval)
Out[93]: float

In [94]: int(fval)
Out[94]: 3

In [95]: bool(fval)
Out[95]: True

In [96]: bool(0)
Out[96]: False
```

Тип None

None – это тип значения null в Python. Если функция явно не возвращает никакого значения, то неявно она возвращает None.

```
In [97]: a = None

In [98]: a is None
Out[98]: True

In [99]: b = 5

In [100]: b is not None
Out[100]: True
```

None также часто применяется в качестве значения по умолчанию для необязательных аргументов функции:

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b

    if c is not None:
        result = result * c

    return result
```

Хотя это вопрос чисто технический, стоит иметь в виду, что None не зарезервированное слово языка, а единственный экземпляр класса NoneType.

```
In [101]: type(None)
Out[101]: NoneType
```

Дата и время

Стандартный модуль Python datetime предоставляет типы datetime, date и time. Тип datetime, как нетрудно сообразить, объединяет информацию, хранящуюся в date и time. Именно он чаще всего и используется:

```
In [102]: from datetime import datetime, date, time

In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [104]: dt.day
Out[104]: 29

In [105]: dt.minute
Out[105]: 30
```

Имея экземпляр datetime, можно получить из него объекты date и time путем вызова одноименных методов:

```
In [106]: dt.date()
Out[106]: datetime.date(2011, 10, 29)

In [107]: dt.time()
Out[107]: datetime.time(20, 30, 21)
```

Метод strftime форматирует объект datetime, представляя его в виде строки:

```
In [108]: dt.strftime('%m/%d/%Y %H:%M')

Out[108]: '10/29/2011 20:30'
```


Чтобы разобрать строку и представить ее в виде объекта datetime, нужно вызвать функцию strptime:

```
In [109]: datetime.strptime('20091031', '%Y%m%d')
Out[109]: datetime.datetime(2009, 10, 31, 0, 0)
```

В табл. 2.5 приведен полный перечень спецификаций формата.

Таблица 2.5. Спецификации формата даты в классе datetime (совместима со стандартом ISO C89)

Спецификатор	Описание
%Y	Год с четырьмя цифрами
%y	Год с двумя цифрами
%m	Номер месяца с двумя цифрами [01, 12]
%d	Номер дня с двумя цифрами [01, 31]
%H	Час (в 24-часовом формате) [00, 23]
%I	Час (в 12-часовом формате) [01, 12]
%M	Минута с двумя цифрами [01, 59]
%S	Секунда [00, 61] (секунды 60 и 61 високосные)
%w	День недели в виде целого числа [0 (воскресенье), 6]
%U	Номер недели в году [00, 53]. Первым днем недели считается воскресенье, а дни, предшествующие первому воскресенью, относятся к неделе 0
%W	Номер недели в году [00, 53]. Первым днем недели считается понедельник, а дни, предшествующие первому понедельнику, относятся к неделе 0
%z	Часовой пояс UTC в виде +HHMM или -HHMM; пустая строка, если часовой пояс не учитывается
%F	Сокращение для %Y-%m-%d, например 2012-4-18
%D	Сокращение для %m/%d/%y, например 04/18/2012

При агрегировании или еще какой-то группировке временных рядов иногда бывает полезно заменить некоторые компоненты даты или времени, например обнулить минуты и секунды, создав новый объект:

```
In [110]: dt.replace(minute=0, second=0)
Out[110]: datetime.datetime(2011, 10, 29, 20, 0)
```

Поскольку тип `datetime.datetime` неизменяемый, эти и другие подобные методы порождают новые объекты.

Вычитание объектов `datetime` дает объект типа `datetime.timedelta`:

```
In [111]: dt2 = datetime(2011, 11, 15, 22, 30)

In [112]: delta = dt2 - dt

In [113]: delta
Out[113]: datetime.timedelta(17, 7179)

In [114]: type(delta)
Out[114]: datetime.timedelta
```

Сложение объектов `timedelta` и `datetime` дает новый объект `datetime`, отстоящий от исходного на указанный промежуток времени:

```
In [115]: dt
Out[115]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [116]: dt + delta
Out[116]: datetime.datetime(2011, 11, 15, 22, 30)
```

Поток управления

В Python имеется несколько ключевых слов для реализации условного выполнения, циклов и других стандартных конструкций *потока управления*, имеющихся в других языках.

if, elif, else

Предложение `if` – одно из самых хорошо известных предложений управления потоком выполнения. Оно вычисляет условие и, если получилось `True`, исполняет код в следующем далее блоке:

```
if x < 0:
    print 'Отрицательно'
```

После предложения `if` может находиться один или несколько блоков `elif` и блок `else`, который выполняется, если все остальные условия оказались равны `False`:

```
if x < 0:
    print 'Отрицательно'
elif x == 0:
    print 'Равно нулю'
```

```
elif 0 < x < 5:
    print 'Положительно, но меньше 5'
else:
    print 'Положительно и больше или равно 5'
```

Если какое-то условие равно `True`, последующие блоки `elif` и `else` даже не рассматриваются. В случае составного условия, в котором есть операторы `and` или `or`, частичные условия вычисляются слева направо с закорачиванием:

```
In [117]: a = 5; b = 7

In [118]: c = 8; d = 4

In [119]: if a < b or c > d:
   .....:    print 'Сделано'
Сделано
```

В этом примере условие `c > d` не вычисляется, потому что уже первое сравнение `a < b` равно `True`.

Можно также строить цепочки сравнений:


```
In [120]: 4 > 3 > 2 > 1
Out[120]: True
```

Циклы for

Циклы `for` предназначены для обхода коллекции (например, списка или кортежа) или итератора. Стандартный синтаксис выглядит так:

```
for value in collection:
    # что-то сделать с value
```

Ключевое слово `continue` позволяет сразу перейти к следующей итерации цикла, не доходя до конца блока. Рассмотрим следующий код, который суммирует целые числа из списка, пропуская значения `None`:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

Ключевое слово `break` осуществляет выход из самого внутреннего цикла, объемлющие циклы продолжают работать:

```
In [121]: for i in range(4):
   .....:     for j in range(4):
   .....:         if j > i:
   .....:             break
   .....:         print((i, j))
   .....:
(0, 0)
(1, 0)
```

```
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

Как мы вскоре увидим, если элементы коллекции или итераторы являются последовательностями (например, кортежем или списком), то их можно *распаковать* в переменные, воспользовавшись циклом `for`:

```
for a, b, c in iterator:
    # что-то сделать
```

Циклы while

Цикл `while` состоит из условия и блока кода, который выполняется до тех пор, пока условие не окажется равным `False` или не произойдет выход из цикла в результате предложения `break`:


```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

Ключевое слово pass

Предложение `pass` Python является пустышкой. Его можно использовать в тех блоках, где не требуется никакого действия; нужно оно только потому, что в Python ограничителем блока выступает пробел:


```
if x < 0:
    print 'отрицательно!'
elif x == 0:
    # TODO: сделать тут что-нибудь разумное
    pass
else:
    print 'положительно!'
```

Функция range

Функция `range` порождает список равноотстоящих целых чисел:

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Можно задать начало и конец диапазона и шаг (который может быть отрицательным):

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

Как видите, конечная точка в порождаемый диапазон `range` не включается. Типичное применение `range` – обход последовательности по индексу:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

Для хранения всех целых чисел, сгенерированных функцией `range`, можно использовать список `list`, но часто задачу проще решить с помощью итератора по умолчанию. Следующий код вычисляет сумму тех чисел от 0 до 9999, которые кратны 3 или 5:


```
sum = 0
for i in xrange(10000):
    # % – оператор деления по модулю
    if x % 3 == 0 or x % 5 == 0:
        sum += i
```

Хотя сгенерированный диапазон может быть сколь угодно большим, в каждый момент времени потребление памяти очень мало.

Тернарные выражения

Тернарное выражение в Python позволяет записать блок `if-else`, порождающий единственное значение, в виде однострочного выражения. Синтаксически это выглядит так:

```
value = true-expr  if condition else false-expr
```

Здесь *true-expr* и *false-expr* могут быть произвольными выражениями Python. Результат такой же, как при выполнении более пространной конструкции:


```
if condition:
    value = true-expr
else:
    value = false-expr
```

Вот более конкретный пример:

```
In [126]: x = 5

In [127]: 'Неотрицательно' if x >= 0 else 'Отрицательно'
Out[127]: 'Неотрицательно'
```

Как и в случае блоков `if-else`, вычисляется только одно из двух подвыражений. То есть в обеих частях («if» и «else») тернарного выражения могут находиться сложные выражения, но вычисляется только одно из них – в «истинной» ветви.

И хотя возникает искушение использовать тернарные выражения всегда, чтобы сократить длину программы, нужно понимать, что если подвыражения очень сложны, то таким образом вы приносите в жертву понятность кода.

Глава 3. Встроенные структуры данных, функции и файлы

В этой главе мы обсудим встроенные в язык Python средства, которыми постоянно будем пользоваться в книге. Библиотеки типа pandas и NumPy добавляют функциональность для работы с большими наборами данных, но опираются они на уже имеющиеся в Python инструменты манипуляции данными.

Мы начнем с базовых структур данных: кортежей, списков, словарей и множеств, затем обсудим, как создавать на Python собственные функции, допускающие повторное использование, и наконец, рассмотрим механизмы работы с файлами и взаимодействия с локальным диском.

3.1. Структуры данных и последовательности

Структуры данных в Python просты, но эффективны. Чтобы стать хорошим программистом на Python, необходимо овладеть ими в совершенстве.

Кортеж

Кортеж – это одномерная неизменяемая последовательность объектов Python фиксированной длины. Проще всего создать кортеж, записав последовательность значений через запятую:

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```

Когда кортеж определяется в более сложном выражении, часто бывает необходимо заключать значения в скобки, как в следующем примере, где создается кортеж кортежей:

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

Любую последовательность или итератор можно преобразовать в кортеж с помощью функции `tuple`:

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [5]: tup = tuple('string')

In [5]: tup
Out[5]: ('s', 't', 'r', 'i', 'n', 'g')
```

К элементам кортежа можно обращаться с помощью квадратных скобок `[]`, как и для большинства других типов последовательностей. Как и в C, C++, Java и многих других языках, нумерация элементов последовательностей в Python начинается с нуля:

```
In [6]: tup[0]
Out[6]: 's'
```

Хотя объекты, хранящиеся в кортеже, могут быть изменяемыми, сам кортеж после создания изменить (т. е. записать что-то другое в существующую позицию) невозможно:

```
In [9]: tup = tuple(['foo', [1, 2], True])

In [10]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

Если какой-то объект кортежа изменяемый, например является списком, то его можно модифицировать на месте:

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

Кортежи можно конкатенировать с помощью оператора `+`, получая в результате более длинный кортеж:

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

Умножение кортежа на целое число, как и в случае списка, приводит к конкатенации нескольких копий кортежа.

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Отметим, что копируются не сами объекты, а только ссылки на них.

Распаковка кортежей

При попытке *присвоить* значение похожему на кортеж выражению, состоящему из нескольких переменных, интерпретатор пытается *распаковать* значение в правой части оператора присваивания:

```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

Распаковать можно даже вложенный кортеж:

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

Эта функциональность позволяет без труда решить задачу обмена значений переменных, которая во многих других языках решается так:

```
tmp = a
a = b
b = tmp
```

Однако в Python обменять значения можно и так:

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

Одно из распространенных применений распаковки переменных – обход последовательности кортежей или списков:

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
```

```
   ....:     print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

Другое применение – возврат нескольких значений из функции. Подробнее об этом ниже.

Недавно в Python были включены дополнительные средства распаковки, на случай когда требуется «отщепить» несколько элементов из начала кортежа. Для этого применяется специальный синтаксис `*rest`, используемый также в сигнатурах функций, чтобы обозначить сколь угодно длинный список позиционных аргументов:

```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```


Часть `rest` иногда требуется отбросить; в самом имени `rest` нет ничего специального, оно может быть любым. По соглашению многие программисты используют для обозначения ненужных переменных знак подчеркивания (_):

```
In [33]: a, b, *_ = values
```

Методы кортежа

Поскольку ни размер, ни содержимое кортежа нельзя модифицировать, методов экземпляра у него совсем немного. Пожалуй, наиболее полезен метод `count` (имеется также у списков), возвращающий количество вхождений значения:

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```

Список

В отличие от кортежей, списки имеют переменную длину, а их содержимое можно модифицировать. Список определяется с помощью квадратных скобок `[]` или конструктора типа `list`:

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)
```

```
In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

Семантически списки и кортежи схожи, поскольку те и другие являются одномерными последовательностями объектов. Следовательно, во многих функциях они взаимозаменяемы.

Функция `list` часто используется при обработке данных, чтобы материализовать итератор или генераторное выражение:

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```


Добавление и удаление элементов

Для добавления элемента в конец списка служит метод `append`:

```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Метод `insert` позволяет вставить элемент в указанную позицию списка:

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```


Индекс позиции вставки должен принадлежать диапазону от 0 до длины списка включительно.

Метод `insert` вычислительно сложнее, чем `append`, так как, чтобы освободить место для нового элемента, приходится сдвигать ссылки на элементы, следующие за ним. Если необходимо вставлять элементы как в начало, так и в конец последовательности, то лучше использовать объект `collections.deque`, специально предназначенный для этой цели.

Операцией, обратной к `insert`, является `pop`, она удаляет из списка элемент, находившийся в указанной позиции, и возвращает его:

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

Элементы можно удалять также по значению методом remove, который находит и удаляет из списка первый элемент с указанным значением:

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

Если снижение производительности из-за использования методов append и remove не составляет проблемы, то список Python вполне можно использовать в качестве структуры данных «мультимножество».

Чтобы проверить, содержит ли список некоторое значение, используется ключевое слово in:

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

Проверка вхождения значения в случае списка занимает гораздо больше времени, чем в случае словаря или множества, потому что Python должен просматривать список от начала до конца, а это требует линейного времени, тогда как поиск в других структурах (основанных на хеш-таблицах) занимает постоянное время.

Конкатенация и комбинирование списков

Как и в случае с кортежами, операция сложения конкатенирует списки:

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

Если уже имеется список, то добавить в его конец несколько элементов позволяет метод extend:

```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

Отметим, что конкатенация – сравнительно дорогая операция, потому что нужно создать новый список и скопировать в него все объекты. Обычно предпочтительнее использовать extend для добавления элементов в имеющийся список, особенно если строится длинный перечень. Таким образом,

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

быстрее, чем эквивалентная конкатенация:

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

Сортировка

Список можно отсортировать на месте (без создания нового объекта), вызвав его метод `sort`:

```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```

У метода `sort` есть несколько удобных возможностей. Одна из них – возможность передать *ключ сортировки*, т. е. функцию, порождающую значение, по которому должны сортироваться объекты. Например, вот как можно отсортировать коллекцию строк по длине:

```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

Вскоре мы познакомимся с функцией `sorted`, которая умеет порождать отсортированную копию любой последовательности.

Двоичный поиск и поддержание списка в отсортированном состоянии

В стандартном модуле `bisect` реализованы операции двоичного поиска и вставки в отсортированный список. Метод `bisect.bisect` находит позицию, в которую следует вставить новый элемент, чтобы список остался отсортированным, а метод `bisect.insort` производит вставку в эту позицию:

```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4

In [70]: bisect.bisect(c, 5)
Out[70]: 6

In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Функции из модуля bisect не проверяют, отсортирован ли исходный список, так как это обошлось бы дорого с вычислительной точки зрения. Их применение к неотсортированному списку завершается без ошибок, но результат может оказаться неверным.

Вырезание

Из большинства последовательностей можно вырезать участки с помощью нотации, которая в простейшей форме сводится к передаче пары start:stop оператору доступа по индексу []:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

Срезу также можно присваивать последовательность:

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Элемент с индексом start включается в срез, элемент с индексом stop не включается, поэтому количество элементов в результате равно stop - start.

Любой член пары, как start, так и stop, можно опустить, тогда по умолчанию подразумевается начало и конец последовательности соответственно:

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

Если индекс в срезе отрицателен, то он отсчитывается от конца последовательности:

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

К семантике вырезания надо привыкнуть, особенно если вы раньше работали с R или MATLAB. На рис. 3.1 показано, как происходит вырезание при положительном и отрицательном индексах. В левом верхнем углу каждой ячейки проставлены индексы, чтобы было проще понять, где начинается и заканчивается срез при положительных и отрицательных индексах.

Допускается и вторая запятая, после которой можно указать шаг, например взять каждый второй элемент:

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 6, 1]
```

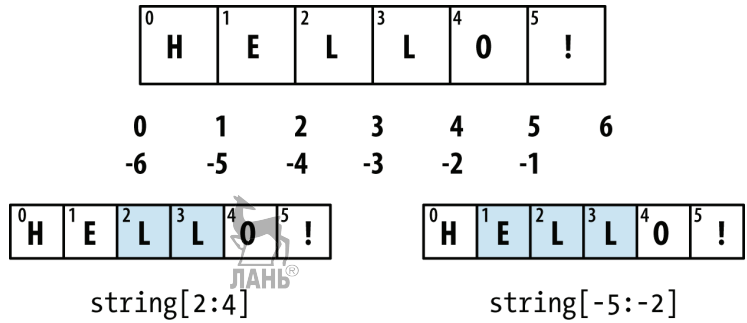


Рис. 3.1. Иллюстрация соглашений о вырезании в Python

Если задать шаг -1, то список или кортеж будут инвертированы:

```
In [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Встроенные функции последовательностей

У последовательностей в Python есть несколько полезных функций, которые следует знать и применять при любой возможности.

enumerate

При обходе последовательности часто бывает необходимо следить за индексом текущего элемента. «Ручками» это можно сделать так:

```
i = 0
for value in collection:
    # что-то сделать с value
    i += 1
```

Но, поскольку эта задача встречается очень часто, в Python имеется встроенная функция `enumerate`, которая возвращает последовательность кортежей `(i, value)`:

```
for i, value in enumerate(collection):
    # что-то сделать с value
```

Функция `enumerate` нередко используется для построения словаря, отображающего значения в последовательности (предполагаемые уникальными) на их позиции:

```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
   ....:     mapping[v] = i
```

```
In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

Функция sorted возвращает новый отсортированный список, построенный из элементов произвольной последовательности:

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

Функция sorted принимает те же аргументы, что метод списков sort.

zip

Функция zip «сшивает» элементы нескольких списков, кортежей или других последовательностей в пары, создавая список кортежей:

```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

Функция zip принимает любое число аргументов, а количество порождаемых ей кортежей определяется длиной *самой короткой* последовательности:

```
In [93]: seq3 = [False, True]

In [94]: zip(seq1, seq2, seq3)
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```

Очень распространенное применение zip – одновременный обход нескольких последовательностей, возможно, в сочетании с enumerate:

```
In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....:     print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```

Если имеется «сшитая» последовательность, то zip можно использовать, чтобы «распороть» ее. Это можно также представить себе как преобразование списка *строк* в список *столбцов*. Синтаксис, несколько причудливый, выглядит следующим образом:

```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   .....:            ('Schilling', 'Curt')]
```

```
In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

reversed

Функция reversed перебирает элементы последовательности в обратном порядке:

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Имейте в виду, что reversed – это генератор (данное понятие мы рассмотрим ниже), следовательно, он не создает инвертированную последовательность, если только не будет материализован (например, с помощью функции list или в цикле for).

Словарь

Словарь, пожалуй, является самой важной из встроенных в Python структур данных. Его также называют *хешем*, *отображением* или *ассоциативным массивом*. Он представляет собой коллекцию пар *ключ–значение* переменного размера, в которой и ключ, и значение – объекты Python. Создать словарь можно с помощью фигурных скобок {}, отделяя ключи от значений двоеточием:


```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Для доступа к элементам, вставки и присваивания применяется такой же синтаксис, как в случае списка или кортежа:

```
In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

Проверка наличия ключа в словаре тоже производится как для кортежа или списка:

```
In [107]: 'b' in d1
Out[107]: True
```

Для удаления ключа можно использовать либо ключевое слово del, либо метод pop (который не только удаляет ключ, но и возвращает ассоциированное с ним значение):

```
In [108]: d1[5] = 'some value'

In [109]: d1
Out[109]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}

In [110]: d1['dummy'] = 'another value'

In [111]: d1
Out[111]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}

In [112]: del d1[5]

In [113]: d1
Out[113]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}

In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

Методы keys и values возвращают соответственно список ключей и список значений. Хотя точный порядок пар *ключ–значение* не определен, эти методы возвращают ключи и значения в одном и том же порядке:

```
In [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

Два словаря можно объединить методом update:

```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

Метод update модифицирует словарь на месте, т. е. старые значения существующих ключей, переданных update, стираются.

Создание словаря из последовательностей

Нередко имеются две последовательности, которые естественно рассматривать как ключи и соответствующие им значения, а значит, требуется построить из них словарь. Первая попытка могла бы выглядеть так:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Поскольку словарь – это, по существу, коллекция 2-кортежей, функция dict принимает список 2-кортежей:

```
In [121]: mapping = dict(zip(range(5), reversed(range(5))))

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Ниже мы рассмотрим *словарное включение* – еще один элегантный способ построения словарей.

Значения по умолчанию

Очень часто можно встретить код, реализующий такую логику:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

Поэтому методы словаря get и pop могут принимать значение, возвращаемое по умолчанию, так что этот блок if-else можно упростить:

```
value = some_dict.get(key, default_value)
```

Метод get по умолчанию возвращает None, если ключ не найден, тогда как pop в этом случае возбуждает исключение. Часто бывает, что значениями в словаре являются другие коллекции, например списки. Так, можно классифицировать слова по первой букве и представить их набор в виде словаря списков:

```
In [123]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [124]: by_letter = {}

In [125]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:
```

```
In [126]: by_letter
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

Метод setdefault предназначен специально для этой цели. Цикл for выше можно переписать так:

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

В стандартном модуле collections есть полезный класс defaultdict, который еще больше упрощает решение этой задачи. Его конструктору передается тип или функция, генерирующие значение по умолчанию для каждой пары в словаре:

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

Допустимые типы ключей словаря

Значениями словаря могут быть произвольные объекты Python, но ключами должны быть неизменяемые объекты, например скалярные типы (int, float, строка) или кортежи (причем все объекты кортежа тоже должны быть неизменяемыми). Технически это свойство называется *хешируемостью*. Проверить, является ли объект хешируемым (и, стало быть, может быть ключом словаря), позволяет функция hash:

```
In [127]: hash('string')
Out[127]: -9167918882415130555

In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782

In [129]: hash((1, 2, [2, 3])) # ошибка, списки изменяемы
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-461-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3]))   # ошибка, списки изменяемы
TypeError: unhashable type: 'list'
```

Чтобы использовать список в качестве ключа, достаточно преобразовать его в кортеж, который допускает хеширование, если это верно для его элементов:

```
In [130]: d = {}

In [131]: d[tuple([1, 2, 3])] = 5

In [132]: d
Out[132]: {(1, 2, 3): 5}
```

Множество

Множество – это неупорядоченная коллекция уникальных элементов. Можно считать, что это словари, не содержащие значений. Создать множество можно двумя способами: с помощью функции `set` или задав *множество-литерал* в фигурных скобках:

```
In [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: set([1, 2, 3])

In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: set([1, 2, 3])
```

Множества поддерживают *теоретико-множественные операции*: объединение, пересечение, разность и симметрическую разность. Рассмотрим следующие два примера множеств:

```
In [135]: a = {1, 2, 3, 4, 5}
In [136]: b = {3, 4, 5, 6, 7, 8}
```

Их объединение – это множество, содержащее неповторяющиеся элементы, встречающиеся хотя бы в одном множестве. Вычислить его можно с помощью метода `union` или бинарного оператора `|`:

```
In [137]: a.union(b)
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}

In [138]: a | b
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}
```

Пересечение множеств содержит элементы, встречающиеся в обоих множествах. Вычислить его можно с помощью метода `intersection` или бинарного оператора `&`:

```
In [139]: a.intersection(b)
Out[139]: {3, 4, 5}

In [140]: a & b
Out[140]: {3, 4, 5}
```

Наиболее употребительные методы множеств перечислены в табл. 3.1.

Таблица 3.1. Операции над множествами в Python

Функция	Альтернативный синтаксис	Описание
`a.add(x)`	Нет	Добавить элемент `x` в множество `a`
`a.clear(x)`	Нет	Опустошить множество, удалив из него все элементы
`a.remove(x)`	Нет	Удалить элемент `x` из множества `a`
`a.pop(x)`	Нет	Удалить какой-то элемент `x` из множества `a` и возбудить исключение `KeyError`, если множество пусто

Таблица 3.1 (окончание)

Функция	Альтернативный синтаксис	Описание
`a.union(b)`	`a | b`	Найти все уникальные элементы, входящие либо в a, либо в b
`a.update(b)`	`a |= b`	Присвоить a объединение элементов a и b
`a.intersection(b)`	`a & b`	Найти все элементы, входящие и в a, и в b
`a.intersection_update(b)`	`a &= b`	Присвоить a пересечение элементов a и b
`a.difference(b)`	`a - b`	Найти элементы, входящие в a, но не входящие в b
`a.difference_update(b)`	`a -= b`	Записать в a элементы, которые входят в a, либо в b, но не входят в b
`a.symmetric_difference(b)`	`a ^ b`	Найти элементы, входящие либо в a, либо в b, но не в a и b одновременно
`a.symmetric_difference_update(b)`	`a ^= b`	Записать в a элементы, которые входят либо в a, либо в b, но не в a и b одновременно
`a.issubset(b)`	`a <= b`	`True`, если все элементы a входят также и в b
`a.issuperset(b)`	`a >= b`	`True`, если все элементы b входят также и в a
`a.isdisjoint(b)`	Нет	`True`, если у a и b нет ни одного общего элемента

У всех логических операций над множествами имеются варианты с обновлением на месте, которые позволяют заменить содержимое множества в левой части результатом операции. Для очень больших множеств это может оказаться эффективнее:

```
In [141]: c = a.copy()

In [142]: c |= b

In [143]: c
Out[143]: {1, 2, 3, 4, 5, 6, 7, 8}

In [144]: d = a.copy()

In [145]: d &= b

In [146]: d
Out[146]: {3, 4, 5}
```

Как и в случае со словарями, элементы множества, вообще говоря, должны быть неизменяемыми. Чтобы включить в множество элементы, подобные списку, необходимо сначала преобразовать их в кортеж.

Можно также проверить, является ли множество подмножеством (содержится в) или надмножеством (содержит) другого множества:

```
In [150]: a_set = {1, 2, 3, 4, 5}

In [151]: {1, 2, 3}.issubset(a_set)
Out[151]: True

In [152]: a_set.issuperset({1, 2, 3})
Out[152]: True
```

Множества называются равными, если состоят из одинаковых элементов:

```
In [153]: {1, 2, 3} == {3, 2, 1}
Out[153]: True
```

Списковое, словарное и множественное включения

Списковое включение[1] – одна из самых любимых особенностей Python. Этот механизм позволяет кратко записать создание нового списка, образованного фильтрацией элементов коллекции с одновременным преобразованием элементов, прошедших через фильтр. Основная синтаксическая форма такова:

```
[expr for val in collection if condition]
```

Это эквивалентно следующему циклу for:

```
result = []
for val in collection:
    if condition:
       result.append(expr)
```

Условие фильтрации можно опустить, оставив только выражение. Например, если задан список строк, то мы могли бы выделить из него строки длиной больше 2 и попутно преобразовать их в верхний регистр:

```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Словарное и множественное включения – естественные обобщения, которые предлагают аналогичную идиому для порождения словарей и множеств. Словарное включение выглядит так:

```
dict_comp = {key-expr : value-expr for value in collection if condition}
```

Множественное включение очень похоже на списковое, и квадратные скобки заменяются фигурными:

```
set_comp = {expr for value in collection if condition}
```

Все виды включений не более чем синтаксическая глазурь, упрощающая чтение и написание кода. Рассмотрим приведенный выше список строк. Допустим, требуется построить множество, содержащее длины входящих в коллекцию строк; это легко сделать с помощью множественного включения:

```
In [156]: unique_lengths = {len(x) for x in strings}
```

[1] Более-менее устоявшийся перевод термина list comprehension – «списковое включение» крайне неудачен и совершенно не отражает сути дела. Я предложил бы термин «трансфильтрация», объединяющий слова «трансформация» и «фильтрация», но не уверен в положительной реакции сообщества. – *Прим. перев.*

```
In [157]: unique_lengths
Out[157]: set([1, 2, 3, 4, 6])
```

То же самое можно записать в духе функционального программирования, воспользовавшись функцией `map`, с которой мы познакомимся ниже:

```
In [158]: set(map(len, strings))
Out[158]: {1, 2, 3, 4, 6}
```

В качестве простого примера словарного включения создадим словарь, сопоставляющий каждой строке ее позицию в списке:

```
In [159]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Вложенное списковое включение

Пусть имеется список списков, содержащий английские и испанские имена:

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
   .....:             ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

Возможно, вы взяли эти имена из двух разных файлов и решили рассортировать их по языкам. А теперь допустим, что требуется получить один список, содержащий все имена, в которых встречается не менее двух букв `e`. Конечно, это можно было бы сделать в таком простом цикле `for`:

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') > 2]
    names_of_interest.extend(enough_es)
```

Но можно обернуть всю операцию одним *вложенным списковым включением*:

```
In [162]: result = [name for names in all_data for name in names
   .....:           if name.count('e') >= 2]
In [163]: result
Out[163]: ['Steven']
```

Поначалу вложенное списковое включение с трудом укладывается в мозгу. Части `for` соответствуют порядку вложенности, а все фильтры располагаются в конце, как и раньше. Вот еще один пример, в котором мы линеаризуем список кортежей целых чисел, создавая один плоский список:

```
In [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [165]: flattened = [x for tup in some_tuples for x in tup]

In [166]: flattened
Out[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Помните, что порядок выражений for точно такой же, как если бы вы писали вложенные циклы for, а не списковое включение:

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

Глубина уровня вложенности не ограничена, хотя, если уровней больше трех, стоит задуматься о правильности выбора структуры данных. Важно отличать показанный выше синтаксис от спискового включения внутри спискового включения – тоже вполне допустимой конструкции:

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Эта операция порождает список списков, а не линеаризованный список всех внутренних элементов.

3.2. Функции

Функции – главный и самый важный способ организации и повторного использования кода в Python. Если вам кажется, что некоторый код может использоваться более одного раза, возможно, с небольшими вариациями, то имеет смысл оформить его в виде функции. Кроме того, функции могут сделать код более понятным, поскольку дают имя группе взаимосвязанных предложений.

Объявление функции начинается ключевым словом def, а результат возвращается в предложении return:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

Ничто не мешает иметь в функции несколько предложений return. Если при выполнении достигнут конец функции, а предложение return не встретилось, то возвращается None.

У функции могут быть *позиционные* и *именованные* аргументы. Именованные аргументы обычно используются для задания значений по умолчанию и необязательных аргументов. В примере выше x и y – позиционные аргументы, а z – именованный. Следующие вызовы функции эквивалентны:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

Основное ограничение состоит в том, что именованные аргументы должны находиться после всех позиционных (если таковые имеются). Сами же именованные аргументы можно задавать в любом порядке, это освобождает программиста от необходимости помнить, в каком порядке были указаны аргументы функции в объявлении. Важно лишь, как они называются.

Ключевые слова можно использовать и для передачи позиционных аргументов. Предыдущий пример можно было бы записать и так:

```
my_function(x=5, y=6, z=7)
my_function(y=6, x=5, z=7)
```

Иногда код при этом становится понятнее.

Пространства имен, области видимости и локальные функции

Функции могут обращаться к переменным, объявленным в двух областях видимости: *глобальной* и *локальной*. Область видимости переменной в Python называют также *пространством имен*. Любая переменная, которой присвоено значение внутри функции, по умолчанию попадает в локальное пространство имен. Локальное пространство имен создается при вызове функции, и в него сразу же заносятся аргументы функции. По завершении функции локальное пространство имен уничтожается (хотя бывают и исключения, см. ниже раздел о замыканиях). Рассмотрим следующую функцию:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

При вызове `func()` создается пустой список `a`, в него добавляется пять элементов, а затем, когда функция завершается, список `a` уничтожается. Но допустим, что мы объявили `a` следующим образом:

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Присваивать значение глобальной переменной внутри функции допустимо, но такие переменные должны быть объявлены глобальными с помощью ключевого слова `global`:

```
In [168]: a = None

In [169]: def bind_a_variable():
.....:        global a
.....:        a = []
```

```
.....:     bind_a_variable()
.....:

In [170]: print a
[]
```

Вообще, я не рекомендую злоупотреблять ключевым словом `global`. Обычно глобальные переменные служат для хранения состояния системы. Если вы понимаете, что пользуетесь ими слишком часто, то стоит подумать о переходе к объектно-ориентированному программированию (использовать классы).

Возврат нескольких значений

Когда я только начинал программировать на Python после многих лет работы на Java и C++, одной из моих любимых возможностей была возможность возвращать из функции несколько значений. Вот простой пример:

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

В анализе данных и других научных приложениях это встречается сплошь и рядом, потому что многие функции вычисляют несколько результатов. На самом деле функция здесь возвращает всего *один* объект, а именно кортеж, который затем распаковывается в результирующие переменные. В примере выше можно было поступить и так:

```
return_value = f()
```

В таком случае `return_value` было бы 3-кортежем, содержащим все три возвращенные переменные. Иногда разумнее возвращать несколько значений не в виде кортежа, а в виде словаря:

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

Полезен ли такой способ, зависит от решаемой задачи.

Функции являются объектами

Поскольку функции в Python – объекты, становятся возможны многие конструкции, которые в других языках выразить трудно. Пусть, например, мы производим очистку данных и должны применить ряд преобразований к следующему списку строк:

```
In [171]: states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda',
   .....:           'south carolina##', 'West virginia?']
```

Всякий, кому доводилось работать с присланными пользователями данными опроса, ожидает такого рода мусор. Чтобы сделать данный список строк пригодным для анализа, нужно произвести различные операции: удалить лишние пробелы и знаки препинания, оставить заглавные буквы только в нужных местах. Сделать это можно, например, с помощью встроенных методов строк и стандартного библиотечного модуля `re` для работы с регулярными выражениями:

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

Вот как выглядит результат:

```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Другой подход, который иногда бывает полезен, – составить список операций, которые необходимо применить к набору строк:

```
def remove_punctuation(value):
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

Далее поступаем следующим образом:

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
```

```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Подобный *функциональный* подход позволяет задать способ модификации строк на очень высоком уровне. Степень повторной используемости функции `clean_strings` определенно возросла!

Функции можно передавать в качестве аргументов другим функциям, например встроенной функции `map`, которая применяет переданную функцию к коллекции:

```
In [176]: for x in map(remove_punctuation, states):
   .....:     print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south carolina
West virginia
```

Анонимные (лямбда) функции

Python поддерживает так называемые *анонимные* функции, или *лямбда*-функции. По существу, это простые однострочные функции, возвращающие значение. Определяются они с помощью ключевого слова `lambda`, которое означает всего лишь «мы определяем анонимную функцию» и ничего более.

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

В этой книге я обычно употребляю термин «лямбда-функция». Они особенно удобны в ходе анализа данных, потому что, как вы увидите, во многих случаях функции преобразования данных принимают другие функции в качестве аргументов. Часто быстрее (и чище) передать лямбда-функцию, чем писать полноценное объявление функции или даже присваивать лямбда-функцию локальной переменной. Рассмотрим такой простенький пример:

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Можно было бы, конечно, написать [x * 2 for x in ints], но в данном случае нам удалось передать функции apply_to_list пользовательский оператор.

Еще пример: пусть требуется отсортировать коллекцию строк по количеству различных букв в строке.

```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Для этого можно передать лямбда-функцию методу списка sort:

```
In [178]: strings.sort(key=lambda x: len(set(list(x))))

In [179]: strings
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

Лямбда-функции называются анонимными, потому что, в отличие от функций, объявленных с помощью ключевого слова def, у такой функции нет явного атрибута __name__.

Каррирование: фиксирование части аргументов

Каррированием (в честь математика Хаскелла Карри) в информатике называется порождение новых функций из существующих путем *фиксирования некоторых аргументов*. Рассмотрим, к примеру, тривиальную функцию, складывающую два числа:

```
def add_numbers(x, y):
    return x + y
```

На ее основе можно создать новую функцию одной переменной add_five, которая прибавляет к своему аргументу 5:

```
add_five = lambda y: add_numbers(5, y)
```

Говорят, что второй аргумент функции add_numbers *каррирован*. В этом нет ничего особо примечательного – мы всего лишь определили новую функцию, которая вызывает существующую. Стандартный модуль functools упрощает эту процедуру за счет функции partial:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

Генераторы

Наличие единого способа обхода последовательностей, например объектов в списке или строк в файле, – важная особенность Python. Реализована она с помощью протокола *итератора*, общего механизма, наделяющего объекты свойством итерируемости. Например, при обходе (итерировании) словаря получаем хранящиеся в нем ключи:

```
In [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}
In [181]: for key in some_dict:
   .....:     print(key)
```

```
a
b
c
```

Встречая конструкцию for key in some_dict, интерпретатор Python сначала пытается создать итератор из some_dict:

```
In [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dictionary-keyiterator at 0x7fbbd5a9f908>
```

Итератор – это любой объект, который отдает интерпретатору Python объекты при использовании в контексте, аналогичном циклу for. Методы, ожидающие получить список или похожий на список объект, как правило, удовлетворяются любым итерируемым объектом. Это относится, в частности, к встроенным методам, например min, max и sum, и к конструкторам типов, например list и tuple:

```
In [184]: list(dict_iterator)
Out[184]: ['a', 'c', 'b']
```

Генератор – это простой способ конструирования итерируемого объекта. Если обычная функция выполняется и возвращает единственное значение, то генератор «лениво» возвращает последовательность значений, приостанавливаясь после возврата каждого в ожидании запроса следующего. Чтобы создать генератор, нужно вместо return использовать ключевое слово yield:

```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i ** 2
```

В момент вызова генератора никакой код не выполняется:

```
In [186]: gen = squares()

In [187]: gen
Out[187]: <generator object squares at 0x34c8280>
```

И лишь после запроса элементов генератор начинает выполнять свой код:

```
In [4]: for x in gen:
   ...:     print x,
   ...:
Генерируются квадраты чисел от 1 до  100
1 4 9 16 25 36 49 64 81 100
```

Генераторные выражения

Еще более лаконичный способ создать генератор – воспользоваться *генераторным выражением*. Такой генератор аналогичен списковому, словарному

и множественному включениям. Чтобы его создать, заключите выражение, которое выглядит как списковое включение, в круглые скобки вместо квадратных:

```
In [189]: gen = (x ** 2 for x in xrange(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x10a0a31e0>
```

Это в точности эквивалентно следующему более многословному определению генератора:

```
def _make_gen():
    for x in xrange(100):
        yield x ** 2
gen = _make_gen()
```

Генераторные выражения можно использовать внутри любой функции Python, принимающей генератор:

```
In [191]: sum(x ** 2 for x in xrange(100))
Out[191]: 328350

In [192]: dict((i, i **2) for i in xrange(5))
    Out[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Модуль itertools

Стандартный библиотечный модуль `itertools` содержит набор генераторов для многих общеупотребительных алгоритмов. Так, генератор `groupby` принимает произвольную последовательность и функцию, он группирует соседние элементы последовательности по значению, возвращенному функцией, например:

```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [196]: for letter, names in itertools.groupby(names, first_letter):
   .....:     print letter, list(names)  # names - это генератор
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

В табл. 3.2 описаны некоторые функции из модуля itertools, которыми я часто пользуюсь. Дополнительную информацию об этом полезном стандартном модуле можно почерпнуть из официальной документации на сайте https://docs.python.org/3/library/itertools.html.

Таблица 3.2. Некоторые полезные функции из модуля itertools

Функция	Описание
`combinations(iterable, k)`	Генерирует последовательность всех возможных k-кортежей, составленных из элементов `iterable`, без учета порядка
`permutations(iterable, k)`	Генерирует последовательность всех возможных k-кортежей, составленных из элементов `iterable`, с учетом порядка
`groupby(iterable[, keyfunc])`	Генерирует пары (ключ, субитератор) для каждого уникального ключа
`product(*iterables, repeat=1)`	Генерирует декартово произведение входных итерируемых величин в виде кортежей, как если бы использовался вложенный цикл `for`

Обработка исключений

Обработка ошибок, или *исключений*, в Python – важная часть создания надежных программ. В приложениях для анализа данных многие функции работают только для входных данных определенного вида. Например, функция `float` может привести строку к типу числа с плавающей точкой, но если формат строки заведомо некорректен, то завершается с ошибкой `ValueError`:

```
In [197]: float('1.2345')
Out[197]: 1.2345
In [198]: float('something')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-198-439904410854> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: 'something'
```

Пусть требуется написать версию `float`, которая не завершается с ошибкой, а возвращает поданный на вход аргумент. Это можно сделать, обернув вызов `float` блоком `try/except`:

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

Код в части `except` будет выполняться, только если `float(x)` возбуждает исключение:

```
In [200]: attempt_float('1.2345')
Out[200]: 1.2345

In [201]: attempt_float('something')
Out[201]: 'something'
```

Кстати, `float` может возбуждать и другие исключения, не только `ValueError`:

```
In [202]: float((1, 2))
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-202-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number, not 'tuple'
```

Возможно, вы хотите перехватить только исключение ValueError, поскольку TypeError (аргумент был не строкой и не числом) может свидетельствовать о логической ошибке в программе. Для этого нужно написать после except тип исключения:


```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

В таком случае получим:

```
In [204]: attempt_float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-204-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-203-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number, not 'tuple'
```

Можно перехватывать исключения нескольких типов, для чего достаточно написать кортеж типов (скобки обязательны):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

Иногда исключение не нужно перехватывать, но какой-то код должен быть выполнен вне зависимости от того, возникло исключение в блоке try или нет. Для этого служит предложение finally:


```
f = open(path, 'w')
try:
    write_to_file(f)
finally:
    f.close()
```

Здесь описатель файла f закрывается *в любом случае*. Можно также написать код, который исполняется, только если в блоке try не было исключения. Для этого используется ключевое слово else:

```
f = open(path, 'w')
try:
    write_to_file(f)
except:
    print 'Ошибка'
else:
    print 'Все хорошо'
finally:
    f.close()
```

Исключения в IPython

Если исключение возникает в процессе выполнения скрипта командой %run или при выполнении любого предложения, то IPython по умолчанию распечатывает весь стек (выполняет трассировку стека) и несколько строк вокруг каждого предложения в стеке, чтобы можно было понять контекст:

```
In [10]: %run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:
```

Наличие дополнительного контекста уже является весомым преимуществом по сравнению со стандартным интерпретатором Python (который никакого контекста не выводит). Объемом контекста можно управлять с помощью магической команды %xmode, он может варьироваться от Plain (так же как в стандартном интерпретаторе Python) до Verbose (печатаются значения

аргументов функций и многое другое). Ниже в этой главе мы увидим, что можно *пошагово выполнять стек* (с помощью команды `%debug` или `%pdb`) после возникновения ошибки, т. е. производить интерактивную постоперационную отладку.

3.3. Файлы и операционная система

В этой книге для чтения файла с диска и загрузки данных из него в структуры Python, как правило, используются такие высокоуровневые средства, как функция `pandas.read_csv`. Однако важно понимать основы работы с файлами в Python. По счастью, здесь все очень просто, и именно поэтому Python так часто выбирают, когда нужно работать с текстом или файлами.

Чтобы открыть файл для чтения или записи, пользуйтесь встроенной функцией `open`, которая принимает относительный или абсолютный путь:

```
In [207]: path = 'ch13/segismundo.txt'

In [208]: f = open(path)
```

По умолчанию файл открывается только для чтения – в режиме `'r'`. Далее описатель файла `f` можно рассматривать как список и перебирать строки:

```
for line in f:
    pass
```

У строк, прочитанных из файла, сохраняется признак конца строки (EOL), поэтому часто можно встретить код, который удаляет концы строк:

```
In [209]: lines = [x.rstrip() for x in open(path)]
In [210]: lines
Out[210]:
['Sueńa el rico en su riqueza,',
 'que más cuidados le ofrece;',
'',
'sueńa el pobre que padece',
'su miseria y su pobreza;',
'',
'sueńa el que a medrar empieza,',
'sueńa el que afana y pretende,',
'sueńa el que agravia y ofende,',
'',
'y en el mundo, en conclusión,',
'todos sueńan lo que son,',
'aunque ninguno lo entiende.',
'']
```

Если для создания объекта файла использовалась функция `open`, то следует явно закрывать файл по завершении работы с ним. Закрытие файла возвращает ресурсы операционной системе:

```
In [211]: f.close()
```

Упростить эту процедуру позволяет предложение `with`:

```
In [212]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```

В таком случае файл `f` автоматически закрывается по выходе из блока `with`.

Если бы мы написали `f = open(path, 'w')`, то был бы создан *новый* файл *examples/segismundo.txt* (будьте осторожны!), а старый был бы перезаписан. Существует также режим `'x'`, в котором создается допускающий запись файл, но лишь в том случае, если его еще не существует, в противном случае возбуждается исключение. Все допустимые режимы ввода-вывода перечислены в табл. 3.3.

Таблица 3.3. Режимы открытия файла в Python

Режим	Описание
`r`	Режим чтения
`w`	Режим записи. Создается новый файл (старый с тем же именем удаляется)
`a`	Дописывание в конец существующего файла (если файла нет, его создают)
`r+`	Чтение и запись
`b`	Уточнение режима для двоичных файлов: `'rb'` или `'wb'`
`t`	Текстовый режим (байты автоматически декодируются в Unicode). Этот режим подразумевается по умолчанию, если не указано противное. Букву `t` можно комбинировать с другими режимами (например, `'rt'` или `'xt'`)

При работе с файлами, допускающими чтение, чаще всего употребляются методы `read`, `seek` и `tell`. Метод `read` возвращает определенное число символов из файла. Что такое «символ», определяется кодировкой файла (например, UTF-8). Если же файл открыт в двоичном режиме, то под символами понимаются просто байты:

```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'Sueña el r'

In [215]: f2 = open(path, 'rb') # Двоичный режим

In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a el '
```

Метод `read` продвигает указатель файла вперед на количество прочитанных байтов. Метод `tell` сообщает текущую позицию:

```
In [217]: f.tell()
Out[217]: 11

In [218]: f2.tell()
Out[218]: 10
```

Хотя мы прочитали из файла 10 символов, позиция равна 11, потому что именно столько байтов пришлось прочитать, чтобы декодировать 19 символов в подразумеваемой по умолчанию кодировке файла. Чтобы узнать кодировку по умолчанию, воспользуемся модулем sys:

```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

Метод seek изменяет позицию в файле на указанную:

```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: 'ń'
```

Наконец, не забудем закрыть файлы:


```
In [223]: f.close()

In [224]: f2.close()
```

Для записи текста в файл служат методы write или writelines. Например, можно было бы создать вариант файла prof_mod.py без пустых строк:

```
In [225]: with open('tmp.txt', 'w') as handle:
   .....:     handle.writelines(x for x in open(path) if len(x) > 1)

In [226]: with open('tmp.txt') as f:
   .....:     lines = f.readlines()

In [227]: lines
Out[227]:
['Sueńa el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueńa el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueńa el que a medrar empieza,\n',
 'sueńa el que afana y pretende,\n',
 'sueńa el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
 'todos sueńan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

В табл. 3.4 приведены многие из наиболее употребительных методов работы с файлами.

Байты и Unicode в применении к файлам

По умолчанию Python открывает файлы (как для чтения, так и для записи) в *текстовом режиме*, предполагая, что вы намереваетесь работать со стро-

ками (которые хранятся в Unicode). Чтобы открыть файл в *двоичном режиме*, следует добавить к основному режиму букву b. Рассмотрим файл из предыдущего раздела (содержащий не-ASCII-символы в кодировке UTF-8):

```
In [230]: with open(path) as f:
   .....:     chars = f.read(10)
In [231]: chars
Out[231]: 'Sueńa el r'
```

Таблица 3.4. Наиболее употребительные методы и атрибуты для работы с файлами в Python

Метод	Описание
`read([size])`	Возвращает прочитанные из файла данные в виде строки. Необязательный аргумент `size` сообщает, сколько байтов читать
`readlines([size])`	Возвращает список прочитанных из файла строк. Необязательный аргумент `size` сообщает, сколько строк читать
`write(str)`	Записывает переданную строку в файл
`writelines(strings)`	Записывает переданную последовательность строк в файл
`close()`	Закрывает описатель файла
`flush()`	Сбрасывает внутренний буфер ввода-вывода на диск
`seek(pos)`	Перемещает указатель чтения-записи на байт файла с указанным номером
`tell()`	Возвращает текущую позицию в файле в виде целого числа
`closed`	`True`, если файл закрыт

UTF-8 – это кодировка Unicode переменной длины, поэтому, когда я запрашиваю чтение нескольких символов из файла, Python читает столько байтов, чтобы после декодирования получилось указанное количество символов (их может быть всего 10, а может быть и целых 40). Если вместо этого открыть файл в режиме `'rb'`, то `read` прочитает ровно столько байтов, сколько запрошено:

```
In [232]: with open(path, 'rb') as f:
   .....:     data = f.read(10)
In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

Зная кодировку текста, вы можете декодировать байты в объект `str` самостоятельно, но только в том случае, если последовательность байтов корректна и полна:

```
In [234]: data.decode('utf8')
Out[234]: 'Sueńa el '

In [235]: data[:4].decode('utf8')
---------------------------------------------------------------------------
UnicodeDecodeError Traceback (most recent call last)
<ipython-input-235-300e0af10bb7> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpected end of data
```

Текстовый режим в сочетании с параметром encoding функции open – удобный способ преобразовать данные из одной кодировки Unicode в другую:

```
In [236]: sink_path = 'sink.txt'
In [237]: with open(path) as source:
   .....:     with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
   .....:         sink.write(source.read())
In [238]: with open(sink_path, encoding='iso-8859-1') as f:
   .....:     print(f.read(10))
Sueña el r
```

Будьте осторожны, вызывая метод seek для файла, открытого не в двоичном режиме. Если указанная позиция окажется в середине последовательности байтов, образующих один символ Unicode, то последующие операции чтения завершатся ошибкой:

```
In [240]: f = open(path)

In [241]: f.read(5)
Out[241]: 'Sueña'

In [242]: f.seek(4)
Out[242]: 4

In [243]: f.read(1)
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-243-7841103e33f5> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input, final)
    319     # декодировать входные данные (с учетом буфера)
    320     data = self.buffer + input
--> 321     (result, consumed) = self._buffer_decode(data, self.errors, final
)
    322     # сохранить недекодированные входные данные до следующего вызова
    323     self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid start byte

In [244]: f.close()
```

Если вы регулярно анализируете текстовые данные в кодировке, отличной от ASCII, то уверенное владение средствами работы с Unicode, имеющимися в Python, придется весьма кстати. Дополнительные сведения смотрите в онлайновой документации на сайте https://docs.python.org/3/.

3.4. Заключение

Вооружившись базовыми знаниями о языке Python и его среде, мы можем перейти к изучению NumPy и вычислениям с массивами.

Глава 4. Основы NumPy: массивы и векторные вычисления

Numerical Python, или, сокращенно, NumPy, – один из важнейших пакетов для численных расчетов в Python. В большинстве пакетов для научных расчетов используются объекты массивов NumPy, которые являются *универсальным языком* обмена данными.

Вот лишь часть того, что предлагает NumPy:

- `ndarray`, эффективный многомерный массив, предоставляющий быстрые арифметические операции с массивами и гибкий механизм *укладывания*;
- математические функции для выполнения быстрых операций над целыми массивами без явного выписывания циклов;
- средства для чтения массива данных с диска и записи его на диск, а также для работы с проецируемыми на память файлами;
- алгоритмы линейной алгебры, генерация случайных чисел и преобразование Фурье;
- средства для интеграции с кодом, написанным на C, C++ или Fortran.

Благодаря наличию простого C API в NumPy очень легко передавать данные внешним библиотекам, написанным на языке низкого уровня, а также получать от внешних библиотек данные в виде массивов NumPy. Эта возможность сделала Python излюбленным языком для обертывания имеющегося кода на C/C++/Fortran с приданием ему динамического и простого в использовании интерфейса.

Хотя сам по себе пакет NumPy почти не содержит средств для моделирования и научных расчетов, понимание массивов NumPy и ориентированных на эти массивы вычислений поможет гораздо эффективнее использовать инструменты типа pandas. Поскольку NumPy – обширная тема, я вынес описание многих продвинутых средств NumPy, в частности укладывания, в приложение A.

В большинстве приложений для анализа данных основной интерес представляет следующая функциональность:

- быстрые векторные операции для переформатирования и очистки данных, выборки подмножеств и фильтрации, преобразований и других видов вычислений;
- стандартные алгоритмы работы с массивами, например фильтрация, удаление дубликатов и теоретико-множественные операции;
- эффективная описательная статистика, агрегирование и обобщение данных;
- выравнивание данных и реляционные операции объединения и соединения разнородных наборов данных;
- описание условной логики в виде выражений-массивов вместо циклов с ветвлением `if-elif-else`;
- групповые операции с данными (агрегирование, преобразование, применение функции).

Хотя в NumPy имеются вычислительные основы для этих операций, по большей части для анализа данных (особенно структурированных или табличных) лучше использовать библиотеку pandas, потому что она предлагает развитый высокоуровневый интерфейс для решения большинства типичных задач обработки данных – простой и лаконичный. Кроме того, в pandas есть кое-какая предметно-ориентированная функциональность, например операции с временными рядами, отсутствующая в NumPy.

Вычисления с массивами в Python уходят корнями в 1995 год, когда Джим Хьюганин (Jim Hugunin) создал библиотеку Numeric. В течение следующих десяти лет программирование с массивами распространилось во многих научных сообществах, но в начале 2000-х годов библиотечная экосистема оказалась фрагментированной. В 2005 году Трэвис Олифант (Travis Oliphant) сумел собрать проект NumPy из существовавших тогда проектов Numeric и Numarray, чтобы объединить сообщество вокруг общей вычислительной инфраструктуры.

Одна из причин, по которым NumPy играет такую важную роль в численных расчетах, – то, что она проектировалась с прицелом на эффективную работу с большими массивами данных. Отметим, в частности, следующие аспекты:

- NumPy хранит данные в непрерывном блоке памяти независимо от других встроенных объектов Python. Алгоритмы NumPy, написанные на языке C, могут работать с этим блоком, не обременяя себя проверкой типов и другими накладными расходами. Массивы NumPy потребляют гораздо меньше памяти, чем встроенные в Python последовательности;
- в NumPy сложные операции применяются к массивам целиком, так что циклы `for` не нужны.

Чтобы вы могли составить представление о выигрыше в производительности, рассмотрим массив NumPy, содержащий миллион чисел, и эквивалентный список Python:

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

Умножим каждую последовательность на 2:

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

Алгоритмы, основанные на NumPy, в общем случае оказываются в 10–100 раз (а то и больше) быстрее аналогов, написанных на чистом Python, и потребляют гораздо меньше памяти.

4.1. NumPy ndarray: объект многомерного массива

Одна из ключевых особенностей NumPy – объект `ndarray` для представления N-мерного массива. Это быстрый и гибкий контейнер для хранения больших наборов данных в Python. Массивы позволяют выполнять математические операции над целыми блоками данных, применяя такой же синтаксис, как для соответствующих операций над скалярами.

Чтобы показать, как NumPy позволяет производить пакетные вычисления, применяя такой же синтаксис, как для встроенных в Python скалярных объектов, я начну с импорта NumPy и генерации небольшого массива случайных данных:

```
In [12]: import numpy as np

# Сгенерировать случайные данные
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047, 0.4789, -0.5194],
       [-0.5557, 1.9658, 1.3934]])
```

Затем произведу математические операции над `data`:

```
In [15]: data * 10
Out[15]:
array([[ -2.0471, 4.7894, -5.1944],
       [ -5.5573, 19.6578, 13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094, 0.9579, -1.0389],
       [-1.1115, 3.9316, 2.7868]])
```

В первом примере все элементы умножены на 10. Во втором примере соответственные элементы в каждой «ячейке» складываются.

В этой главе и во всей книге я буду придерживаться стандартного соглашения NumPy и писать `import numpy as np`. Конечно, вы можете включить в свой код предложение `from numpy import *`, чтобы не писать каждый раз `np.`, но я рекомендую не брать это в привычку. Пространство имен `numpy` очень велико и содержит ряд функций, имена которых совпадают с именами встроенных в Python функций (например, `min` и `max`).

ndarray – это обобщенный многомерный контейнер для однородных данных, т. е. в нем могут храниться только элементы одного типа. У любого массива есть атрибут `shape` – кортеж, описывающий размер по каждому измерению, и атрибут `dtype` – объект, описывающий *тип данных* в массиве:

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

В этой главе вы познакомитесь с основами работы с массивами NumPy в объеме, достаточном для чтения книги. Для многих аналитических приложений глубокое понимание NumPy необязательно, но овладение стилем мышления и методами программирования, ориентированными на массивы, – ключевой этап на пути становления эксперта по применению Python в научных приложениях.

Слова «массив», «массив NumPy» и «ndarray» в этой книге почти всегда означают одно и то же – объект ndarray.

Создание ndarray

Проще всего создать массив с помощью функции `array`. Она принимает любой объект, похожий на последовательность (в том числе другой массив), и порождает новый массив NumPy, содержащий переданные данные. Например, такое преобразование можно проделать со списком:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. , 7.5, 8. , 0. , 1. ])
```

Вложенные последовательности, например список списков одинаковой длины, можно преобразовать в многомерный массив:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Поскольку data2 – список списков, массив NumPy arr2 имеет два измерения, а его форма выведена из данных. В этом легко убедиться, прочитав атрибуты ndim и shape:

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```


Если явно не задано противное (подробнее об этом ниже), то функция np.array пытается самостоятельно определить подходящий тип данных для создаваемого массива. Этот тип данных хранится в специальном объекте dtype. Например, в примерах выше имеем:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

Помимо np.array существует еще ряд функций для создания массивов. Например, zeros и ones создают массивы заданной длины, состоящие из нулей и единиц соответственно, а shape.empty создает массив, не инициализируя его элементы. Для создания многомерных массивов нужно передать кортеж, описывающий форму:

```
In [29]: np.zeros(10)
Out[29]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:

array([[[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]],
       [[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]]])
```

Предполагать, что np.empty возвращает массив из одних нулей, небезопасно. Часто возвращается массив, содержащий неинициализированный мусор, как в примере выше.

Функция arange – вариант встроенной в Python функции range, только возвращаемым значением является массив:

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

В табл. 4.1 приведен краткий список стандартных функций создания массива. Поскольку NumPy ориентирован прежде всего на численные расчеты, тип данных, если он не указан явно, во многих случаях предполагается float64 (числа с плавающей точкой).

Таблица 4.1. Функции создания массива

Функция	Описание
array	Преобразует входные данные (список, кортеж, массив или любую другую последовательность) в ndarray. Тип dtype задается явно или выводится неявно. Входные данные по умолчанию копируются
asarray	Преобразует входные данные в ndarray, но не копирует, если на вход уже подан ndarray
arange	Аналогична встроенной функции range, но возвращает массив, а не список
ones, ones_like	Порождает массив, состоящий из одних единиц, с заданными атрибутами shape и dtype. Функция ones_like принимает другой массив и порождает массив из одних единиц с такими же значениями shape и dtype
zeros, zeros_like	Аналогичны ones и ones_like, только порождаемый массив состоит из одних нулей
empty, empty_like	Создают новые массивы, выделяя под них память, но, в отличие от ones и zeros, не инициализируют ее
full, full_like	Создают массивы с заданными атрибутами shape и dtype, в которых все элементы равны заданному символу-заполнителю. full_like принимает массив и порождает заполненный массив с такими же значениями атрибутов shape и dtype
eye, identity	Создают единичную квадратную матрицу N×N (элементы на главной диагонали равны 1, все остальные – 0)

Тип данных для ndarray

Тип данных, или dtype, – это специальный объект, который содержит информацию (*метаданные*), необходимую ndarray для интерпретации содержимого блока памяти:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Объектам dtype NumPy в значительной мере обязан своей эффективностью и гибкостью. В большинстве случаев они точно соответствуют внутреннему машинному представлению, что позволяет без труда читать и записывать двоичные потоки данных на диск, а также обмениваться данными с кодом, написанным на языке низкого уровня типа C или Fortran. Числовые dtype именуются единообразно: имя типа, например float или int, затем число, указывающее разрядность одного элемента. Стандартное значение с плавающей точкой двойной точности (хранящееся во внутреннем представлении объекта Python типа float) занимает 8 байтов, или 64 бита. Поэтому соответствующий тип в NumPy называется float64. В табл. 4.2 приведен полный список поддерживаемых NumPy типов данных.

Не пытайтесь сразу запомнить все типы данных NumPy, особенно если вы только приступаете к его изучению. Часто нужно заботиться лишь об общем *виде* обрабатываемых данных, например о числах с плавающей точкой, о комплексных, целых, булевых значениях, о строках или общих объектах Python. Если необходим более точный контроль над тем, как данные хранятся в памяти или на диске, особенно когда речь идет о больших наборах данных, то знать о возможности такого контроля полезно.

Таблица 4.2. Типы данных NumPy

Функция	Код типа	Описание
int8, uint8	i1, u1	Знаковое и беззнаковое 8-разрядное (1 байт) целое
int16, uint16	i2, u2	Знаковое и беззнаковое 16-разрядное (2 байта) целое
int32, uint32	i4, u4	Знаковое и беззнаковое 32-разрядное (4 байта) целое
int64, uint64	i8, u8	Знаковое и беззнаковое 64-разрядное (8 байт) целое
float16	f2	С плавающей точкой половинной точности
float32	f4	Стандартный тип с плавающей точкой одинарной точности. Совместим с типом C float
float64	f8 или d	Стандартный тип с плавающей точкой двойной точности. Совместим с типом C double и с типом Python float
float128	f16	С плавающей точкой расширенной точности
complex64, complex128, complex256	c8, c16, c32	Комплексные числа, вещественная и мнимая части которых представлены соответственно типами float32, float64 и float128
bool	?	Булев тип, способный хранить значения True и False
object	O	Тип объекта Python
string_	S	Тип ASCII-строки фиксированной длины (1 байт на символ). Например, строка длиной 10 имеет тип 'S10'
unicode_	U	Тип Unicode-строки фиксированной длины (количество байтов на символ зависит от платформы). Семантика такая же, как у типа string_ (например, 'U10')

Можно явно преобразовать, или *привести*, массив одного типа к другому, воспользовавшись методом astype:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
```

```
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

Здесь целые были приведены к типу с плавающей точкой. Если бы я попытался привести числа с плавающей точкой к целому типу, то дробная часть была бы отброшена:

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
Out[42]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

Если имеется массив строк, представляющих целые числа, то `astype` позволит преобразовать их в числовую форму:

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [45]: numeric_strings.astype(float)
Out[45]: array([ 1.25, -9.6 , 42. ])
```

Будьте осторожнее при работе с типом `numpy.string_`, поскольку в NumPy размер строковых данных фиксирован и входные данные могут быть обрезаны без предупреждения. Поведение pandas для нечисловых данных лучше согласуется с интуицией.

Если по какой-то причине выполнить приведение не удастся (например, если строку нельзя преобразовать в тип `float64`), то будет возбуждено исключение `TypeError`. Обратите внимание, что в примере выше я поленился и написал `float` вместо `np.float64`, но NumPy оказался достаточно умным – он умеет подменять типы Python эквивалентными dtype.

Можно также использовать атрибут dtype другого массива:

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

На dtype можно сослаться с помощью коротких кодов типа:

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')
In [50]: empty_uint32
Out[50]:
array([         0, 1075314688,          0, 1075707904,          0,
       1075838976,          0, 1072693248], dtype=uint32)
```

При вызове astype *всегда* создается новый массив (данные копируются), даже если новый dtype не отличается от старого.

Арифметические операции с массивами NumPy

Массивы важны, потому что позволяют выразить операции над совокупностями данных без выписывания циклов for. Обычно это называется *векторизацией*. Любая арифметическая операция над массивами одинакового размера применяется к соответственным элементам:

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])

In [53]: arr * arr
Out[53]:
array([[  1.,  4.,  9.],
       [ 16., 25., 36.]])

In [54]: arr - arr
Out[54]:
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
```

Как легко догадаться, арифметические операции, в которых участвует скаляр, применяются к каждому элементу массива:

```
In [55]: 1 / arr
Out[55]:
array([[ 1. ,   0.5 , 0.3333],
       [ 0.25 , 0.2 , 0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1. , 1.4142, 1.7321],
       [ 2. , 2.2361, 2.4495]])
```

Сравнение массивов одинакового размера дает булев массив:

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[ 0., 4., 1.],
       [ 7., 2., 12.]])

In [59]: arr2 > arr
Out[59]:
```

```
array([[False, True, False],
       [ True, False, True]], dtype=bool)
```

Операции между массивами разного размера называются *укладыванием*, мы будем подробно рассматривать их в приложении А. Глубокое понимание укладывания необязательно для чтения большей части этой книги.

Индексирование и вырезание

Индексирование массивов NumPy – обширная тема, поскольку подмножество массива или его отдельные элементы можно выбрать различными способами. С одномерными массивами все просто. На поверхностный взгляд они ведут себя как списки Python:

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

Как видите, если присвоить скалярное значение срезу, как в `arr[5:8] = 12`, то оно распространяется (или *укладывается*) на весь срез. Важнейшее отличие от списков состоит в том, что срез массива является *представлением* исходного массива. Это означает, что данные на самом деле не копируются, а любые изменения, внесенные в представление, попадают и в исходный массив.

Для демонстрации я сначала создам срез массива `arr`:

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

Если теперь изменить значения в `arr_slice`, то изменения отразятся и на исходном массиве `arr`:

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([ 0,  1,  2,  3,  4,  12,  12345,  12,  8,  9])
```

Присваивание неуточненному срезу `[:]` приводит к записи значения во все элементы массива:

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

При первом знакомстве с NumPy это может стать неожиданностью, особенно если вы привыкли к программированию массивов в других языках, где копирование данных применяется чаще. Но NumPy проектировался для работы с большими массивами данных, поэтому при безудержном копировании данных неизбежно возникли бы проблемы с быстродействием и памятью.

Чтобы получить копию, а не представление среза массива, нужно выполнить операцию копирования явно, например `arr[5:8].copy()`.

Для массивов большей размерности и вариантов больше. В случае двумерного массива результатом индексирования с одним индексом является не скаляр, а одномерный массив:

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```


К отдельным элементам можно обращаться рекурсивно. Но это слишком громоздко, поэтому для выбора одного элемента можно указать список индексов через запятую. Таким образом, следующие две конструкции эквивалентны:

```
In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

Рисунок 4.1 иллюстрирует индексирование двумерного массива. Лично мне удобно представлять ось 0 как «строки» массива, а ось 1 – как «столбцы».

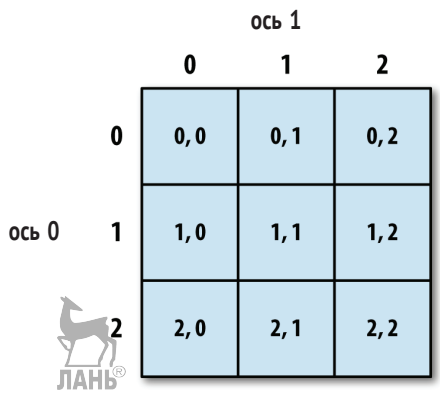


Рис. 4.1. Индексирование элементов в массиве NumPy

Если при работе с многомерным массивом опустить несколько последних индексов, то будет возвращен объект ndarray меньшей размерности, содержащий данные по указанным при индексировании осям. Так, пусть имеется массив arr3d размерности 2×2×3:

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1, 2, 3],
        [ 4, 5, 6]],
       [[ 7, 8, 9],
        [10, 11, 12]]])
```

Тогда arr3d[0] – массив размерности 2×3:

```
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Выражению arr3d[0] можно присвоить как скалярное значение, так и массив:

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7, 8, 9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
array([[[ 1, 2, 3],
        [ 4, 5, 6]],
       [[ 7, 8, 9],
        [10, 11, 12]]])
```

Аналогично arr3d[1, 0] дает все значения, список индексов которых начинается с (1, 0), т. е. одномерный массив:

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

Результат такой же, как если бы мы индексировали в два приема:

```
In [85]: x = arr3d[1]

In [86]: x
```

```
Out[86]:

array([[ 7, 8, 9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

Отметим, что во всех случаях, когда выбираются участки массива, результат является представлением.

Индексирование срезами

Как и для одномерных объектов, наподобие списков Python, для объектов ndarray можно формировать срезы:

```
In [88]: arr
Out[88]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

In [89]: arr[1:6]
Out[89]: array([ 1, 2, 3, 4, 64])
```

Рассмотрим приведенный выше двумерный массив `arr2d`. Применение к нему вырезания дает несколько иной результат:

```
In [90]: arr2d
Out[90]:

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
      [4, 5, 6]])
```

Как видите, вырезание производится вдоль оси 0, первой оси. Поэтому срез содержит диапазон элементов вдоль этой оси. Выражение `arr2d[:2]` полезно читать так: «выбрать первые две строки `arr2d`».

Можно указать несколько срезов – как несколько индексов:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],
       [5, 6]])
```

При таком вырезании мы всегда получаем представления массивов с таким же числом измерений, как у исходного. Сочетая срезы и целочисленные индексы, можно получить массивы меньшей размерности.

Например, я могу выбрать вторую строку, а в ней только первые два столбца:

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

Аналогично я могу выбрать третий столбец, а в нем только первые две строки:

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

Иллюстрация приведена на рис. 4.2. Отметим, что двоеточие без указания числа означает, что нужно взять всю ось целиком, поэтому для получения осей только высших размерностей можно поступить следующим образом:

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```


Разумеется, присваивание выражению-срезу означает присваивание всем элементам этого среза:

```
In [96]: arr2d[:2, 1:] = 0
```

```
In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

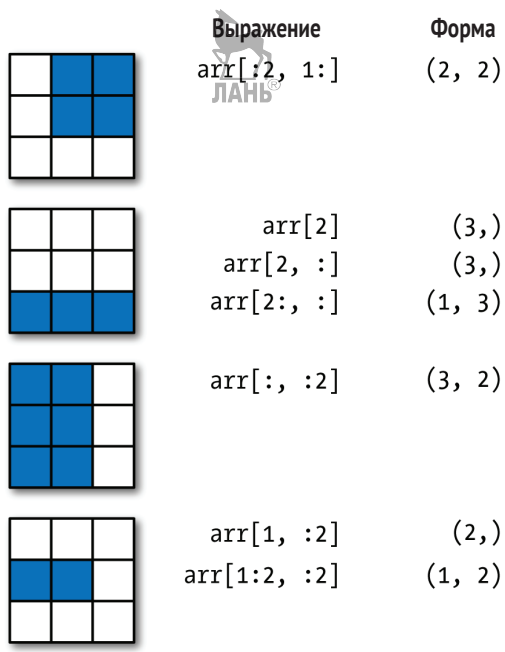


Рис. 4.2. Вырезание из двумерного массива

Булево индексирование

Пусть имеется некоторый массив с данными и массив имен, содержащий дубликаты. Я хочу воспользоваться функцией `randn` из модуля `numpy.random`, чтобы сгенерировать случайные данные с нормальным распределением:

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Допустим, что каждое имя соответствует строке в массиве `data`, и мы хотим выбрать все строки, которым соответствует имя `'Bob'`. Операции сравнения массивов (например, `==`), как и арифметические, также векторизованы. Поэтому сравнение `names` со строкой `'Bob'` дает массив булевых величин:

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

Этот булев массив можно использовать для индексирования другого массива:

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.669 , -0.4386, -0.5397, 0.477 ]])
```

Длина булева массива должна совпадать с длиной индексируемой им оси. Можно даже сочетать булевы массивы со срезами и целыми числами (или последовательностями целых чисел, о чем речь пойдет ниже):

В следующих примерах я выбираю строки, в которых `names == 'Bob'`, и одновременно задаю индекс столбцов:

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 , 1.2464],
       [-0.5397, 0.477 ]])
```

```
In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464, 0.477 ])
```

Чтобы выбрать все, кроме `'Bob'`, можно либо воспользоваться оператором сравнения `!=`, либо применить отрицание условия, обозначаемое знаком `-`:

```
In [106]: names != 'Bob'
Out[106]: array([False, True, True, False, True, True, True], dtype=bool)
```

```
In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Если требуется инвертировать условие общего вида, то пригодится оператор `~`:

```
In [108]: cond = names == 'Bob'
```

```
In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Чтобы сформировать составное булево условие, включающее два из трех имен, воспользуемся булевыми операторами `&` (И) и `|` (ИЛИ):

```
In [110]: mask = (names == 'Bob') | (names == 'Will')
```

```
In [111]: mask
Out[111]: array([True, False, True, True, True, False, False], dtype=bool)
```

```
In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

При выборке данных из массива путем булева индексирования *всегда* создается копия данных, даже если возвращенный массив совпадает с исходным.

Ключевые слова Python `and` и `or` с булевыми массивами не работают. Используйте вместо них `&` (и) и `|` (или).

Задание значений с помощью булевых массивов работает в соответствии с ожиданиями. Чтобы заменить все отрицательные значения в массиве data нулем, нужно всего лишь написать:

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.0072, 0.    , 0.275 , 0.2289],
       [ 1.3529, 0.8864, 0.    , 0.    ],
       [ 1.669 , 0.    , 0.    , 0.477 ],
       [ 3.2489, 0.    , 0.    , 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [ 0.    , 0.    , 0.    , 0.    ]])
```

Задать целые строки или столбцы с помощью одномерного булева массива тоже просто:


```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7.    , 7.    , 7.    , 7.    ],
       [ 1.0072, 0.    , 0.275 , 0.2289],
       [ 7.    , 7.    , 7.    , 7.    ],
       [ 7.    , 7.    , 7.    , 7.    ],
       [ 7.    , 7.    , 7.    , 7.    ],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [ 0.    , 0.    , 0.    , 0.    ]])
```

Как мы вскоре увидим, такого рода операции над двумерными данными удобно производить в pandas.

Прихотливое индексирование

Термином *прихотливое индексирование* (fancy indexing) в NumPy обозначается индексирование с помощью целочисленных массивов. Допустим, имеется массив 8×4:

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....: arr[i] = i

In [119]: arr
Out[119]:
array([[ 0., 0., 0., 0.],
       [ 1., 1., 1., 1.],
       [ 2., 2., 2., 2.],
       [ 3., 3., 3., 3.],
```

```
       [ 4., 4., 4., 4.],
       [ 5., 5., 5., 5.],
       [ 6., 6., 6., 6.],
       [ 7., 7., 7., 7.]])
```

Чтобы выбрать подмножество строк в определенном порядке, можно просто передать список или массив целых чисел, описывающих желаемый порядок:

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4., 4., 4., 4.],
       [ 3., 3., 3., 3.],
       [ 0., 0., 0., 0.],
       [ 6., 6., 6., 6.]])
```

Надеюсь, что этот код делает именно то, что вы ожидаете! Если указать отрицательный индекс, то номер соответствующей строки будет отсчитываться с конца:

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5., 5., 5., 5.],
       [ 3., 3., 3., 3.],
       [ 1., 1., 1., 1.]])
```

При передаче нескольких массивов индексов делается несколько иное: выбирается одномерный массив элементов, соответствующих каждому кортежу индексов:

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

О методе `reshape` мы подробнее поговорим в приложении А.

Здесь отбираются элементы в позициях `(1, 0)`, `(5, 3)`, `(7, 1)` и `(2, 2)`. Вне зависимости от количества измерений массива (в данном случае двух) результат прихотливого индексирования всегда одномерный.

В данном случае поведение прихотливого индексирования отличается от того, что ожидают многие пользователи (я в том числе): получить прямо-

угольный регион, образованный подмножеством строк и столбцов матрицы. Добиться этого можно, например, так:

```
In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Имейте в виду, что прихотливое индексирование, в отличие от вырезания, всегда порождает новый массив, в который копируются данные.

Транспонирование массивов и перестановка осей

Транспонирование – частный случай изменения формы, при этом также возвращается представление исходных данных без какого-либо копирования. У массивов имеется метод `transpose` и специальный атрибут `T`:

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0, 5, 10],
       [ 1, 6, 11],
       [ 2, 7, 12],
       [ 3, 8, 13],
       [ 4, 9, 14]])
```

При вычислениях с матрицами эта операция применяется очень часто. Вот, например, как вычисляется матрица X^TX с помощью метода `np.dot`:

```
In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
```

```
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

Для массивов большей размерности метод transpose принимает кортеж номеров осей, описывающий их перестановку (чтобы ум за разум совсем заехал):

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Здесь порядок осей изменен: вторая ось стала первой, первая – второй, а третья осталась неизменной.

Обычное транспонирование с помощью .T – частный случай перестановки осей. У объекта ndarray имеется метод swapaxes, который принимает пару номеров осей и меняет их местами, в результате чего данные реорганизуются:

```
In [135]: arr
Out[135]:
array([[[ 0, 1, 2, 3],
        [ 4, 5, 6, 7]],
       [[ 8, 9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0, 4],
        [ 1, 5],
        [ 2, 6],
        [ 3, 7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

Метод swapaxes также возвращает представление без копирования данных.

4.2. Универсальные функции: быстрые поэлементные операции над массивами

Универсальной функцией, или *u-функцией*, называется функция, которая выполняет поэлементные операции над данными, хранящимися в объектах ndarray. Можно считать, что это векторные обертки вокруг простых функций, которые принимают одно или несколько скалярных значений и порождают один или несколько скалярных результатов.

Многие u-функции – простые поэлементные преобразования, например `sqrt` или `exp`:

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[130]:
array([ 0.    , 1.    , 1.4142, 1.7321, 2. , 2.2361, 2.4495,
        2.6458, 2.8284, 3.    ])

In [140]: np.exp(arr)
Out[140]:
array([ 1.      ,   2.7183,    7.3891,   20.0855,   54.5982,
        148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Такие u-функции называются *унарными*. Другие, например `add` или `maximum`, принимают два массива (и потому называются *бинарными*) и возвращают один результирующий массив:

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119, 1.0048, 1.3272, -0.9193, -1.5491, 0.0222, 0.7584,
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01 , 0.05 , 0.6702, 0.853 , -0.9559, -0.0235,
       -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626, 1.0048, 1.3272, 0.6702, 0.853 , 0.0222, 0.7584,
       -0.6605])
```

Здесь `numpy.maximum` вычисляет поэлементные максимумы в массивах `x` и `y`.

Хотя и нечасто, но можно встретить u-функцию, возвращающую несколько массивов. Примером может служить `modf`, векторный вариант встроенной

в Python функции divmod: она возвращает дробные и целые части хранящихся в массиве чисел с плавающей точкой:

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 , 0.3731, 0.6182, 0.45 , 0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6., 5., 3., 3., 5.])
```

U-функции принимают необязательный аргумент out, который позволяет выполнять операции над массивами на месте:

```
In [151]: arr
Out[151]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5.0077])

In [152]: np.sqrt(arr)
Out[152]: array([    nan,     nan,     nan, 2.318 , 1.9022, 1.8574, 2.2378])

In [153]: np.sqrt(arr, arr)
Out[153]: array([    nan,     nan,     nan, 2.318 , 1.9022, 1.8574, 2.2378])

In [154]: arr
Out[154]: array([    nan,     nan,     nan, 2.318 , 1.9022, 1.8574, 2.2378])
```

В табл. 4.3 и 4.4 перечислены имеющиеся u-функции.

Таблица 4.3. Унарные u-функции

Функция	Описание
abs, fabs	Вычислить абсолютное значение целых, вещественных или комплексных элементов массива. Для вещественных данных fabs работает быстрее
sqrt	Вычислить квадратный корень из каждого элемента. Эквивалентно arr ** 0.5
square	Вычислить квадрат каждого элемента. Эквивалентно arr ** 2
exp	Вычислить экспоненту e^x каждого элемента
log, log10, log2, log1p	Натуральный (по основанию e), десятичный, двоичный логарифм и функция log(1 + x) соответственно
sign	Вычислить знак каждого элемента: 1 (для положительных чисел), 0 (для нуля) или –1 (для отрицательных чисел)
ceil	Вычислить для каждого элемента наименьшее целое число, не меньшее его
floor	Вычислить для каждого элемента наибольшее целое число, не большее его
rint	Округлить элементы до ближайшего целого с сохранением dtype
modf	Вернуть дробные и целые части массива в виде отдельных массивов
isnan	Вернуть булев массив, показывающий, какие значения являются NaN (не числами)

Таблица 4.3 (окончание)

Функция	Описание
`isfinite`, `isinf`	Вернуть булев массив, показывающий, какие элементы являются конечными (не inf и не NaN) или бесконечными соответственно
`cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`	Обычные и гиперболические тригонометрические функции
`arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctanh`	Обратные тригонометрические функции
`logical_not`	Вычислить значение истинности `not x` для каждого элемента. Эквивалентно `-arr`

Таблица 4.4. Бинарные u-функции

Функция	Описание
`add`	Сложить соответственные элементы массивов
`subtract`	Вычесть элементы второго массива из соответственных элементов первого
`multiply`	Перемножить соответственные элементы массивов
`divide`, `floor_divide`	Деление и деление с отбрасыванием остатка
`power`	Возвести элементы первого массива в степени, указанные во втором массиве
`maximum`, `fmax`	Поэлементный максимум. Функция `fmax` игнорирует значения NaN
`minimum`, `fmin`	Поэлементный минимум. Функция `fmin` игнорирует значения NaN
`mod`	Поэлементный модуль (остаток от деления)
`copysign`	Копировать знаки значений второго массива в соответственные элементы первого массива
`greater`, `greater_equal`, `less`, `less_equal`, `equal`, `not_equal`	Поэлементное сравнение, возвращается булев массив. Эквивалентны инфиксным операторам `>`, `>=`, `<`, `<=`, `==`, `!=`
`logical_and`, `logical_or`, `logical_xor`	Вычислить логическое значение истинности логических операций. Эквивалентны инфиксным операторам `&`, `|`, `^`

4.3. Программирование с применением массивов

С помощью массивов NumPy многие виды обработки данных можно записать очень кратко, не прибегая к циклам. Такой способ замены явных циклов выражениями-массивами обычно называется *векторизацией*. Вообще, векторные операции с массивами выполняются на один-два (а то и больше) порядка быстрее, чем эквивалентные операции на чистом Python. Позже, в приложении А, я расскажу об *укладывании*, действенном методе векторизации вычислений.

В качестве простого примера предположим, что нужно вычислить функцию `sqrt(x^2 + y^2)` на регулярной сетке. Функция `np.meshgrid` принимает два одномерных массива и порождает две двумерные матрицы, соответствующие всем парам `(x, y)` элементов, взятых из обоих массивов:

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 равноотстоящих точек

In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Теперь для вычисления функции достаточно написать такое же выражение, как для двух точек:

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
array([[ 7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       ...,
       [ 7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
       [ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       [ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

Предвосхищая главу 9, воспользуюсь библиотекой matplotlib для визуализации двумерного массива:

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[161]: <matplotlib.colorbar.Colorbar instance at 0x4e46d40>

In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```

На рис. 4.3 показан результат применения функции `imshow` из библиотеки matplotlib для создания изображения по двумерному массиву значений функции.

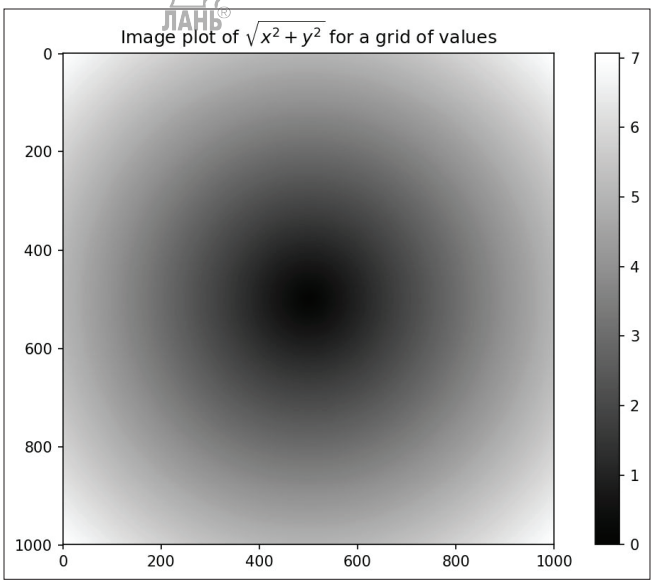


Рис. 4.3. График функции двух переменных на сетке

Запись логических условий в виде операций с массивами

Функция `numpy.where` – это векторный вариант тернарного выражения `x if condition else y`. Пусть есть булев массив и два массива значений:

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

Допустим, мы хотим брать значение из массива `xarr`, если соответственное значение в массиве `cond` равно `True`, а в противном случае – значение из `yarr`. Эту задачу решает такая операция спискового включения:

```
In [168]: result = [(x if c else y)
   .....: for x, y, c in zip(xarr, yarr, cond)]

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

Здесь сразу несколько проблем. Во-первых, для больших массивов это будет не быстро (потому что весь код написан на чистом Python). Во-вторых, к многомерным массивам такое решение вообще неприменимо. С помощью функции np.where можно написать очень лаконичный код:

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1, 2.2, 1.3, 1.4, 2.5])
```

Второй и третий аргументы np.where не обязаны быть массивами – один или оба могут быть скалярами. При анализе данные where обычно применяются, чтобы создать новый массив на основе существующего. Предположим, имеется матрица со случайными данными и мы хотим заменить все положительные значения на 2, а все отрицательные – на –2. С помощью np.where сделать это очень просто:

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True, True, False, True],
       [ True, True, True, False],
       [ True, False, True, True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2, 2, -2, 2],
       [ 2, 2, 2, -2],
       [ 2, -2, 2, 2]])
```

С помощью метода np.where можно комбинировать скаляры и массивы. Например, я могу заменить все положительные элементы arr константой 2:

```
In [176]: np.where(arr > 0, 2, arr) # заменить положительные элементы на 2
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

Передавать where можно не только массивы одинакового размера или скаляры.

Математические и статистические операции

Среди методов массива есть математические функции, которые вычисляют статистики массива в целом или данных вдоль одной оси. Выполнить агрегирование (часто его называют *редукцией*) типа `sum`, `mean` или стандартного отклонения `std` можно с помощью как метода экземпляра массива, так и функции на верхнем уровне NumPy:

Ниже я сгенерирую случайные данные с нормальным распределением и вычислю некоторые агрегаты:

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

Функции типа `mean` и `sum` принимают необязательный аргумент `axis`, при наличии которого вычисляется статистика по заданной оси. В результате порождается массив на единицу меньшей размерности:

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 , 0.1875, -0.502 , -0.0881, 0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345, 2.2381, 1.1486])
```

Здесь `arr.mean(1)` означает «вычислить среднее по столбцам», а `arr.sum(0)` – «вычислить сумму по строкам».

Другие методы, например `cumsum` и `cumprod`, ничего не агрегируют, а порождают массив промежуточных результатов:

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

Для многомерных массивов функция `cumsum` и другие функции с нарастающим итогом возвращают массив того же размера, элементами которого являются частичные агрегаты по указанной оси, вычисленные для каждого среза меньшей размерности:

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
In [187]: arr

Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[ 0,  0,   0],
       [ 3, 12,  60],
       [ 6, 42, 336]])
```

Полный список приведен в табл. 4.5. Как многие из этих методов применяются на практике, мы увидим в последующих главах.

Таблица 4.5. Статистические методы массива

Метод	Описание
`sum`	Сумма элементов всего массива или вдоль одной оси. Для массивов нулевой длины функция `sum` возвращает 0
`mean`	Среднее арифметическое. Для массивов нулевой длины равно NaN
`std, var`	Стандартное отклонение и дисперсия соответственно. Может быть задано число степеней свободы (по умолчанию знаменатель равен n)
`min, max`	Минимум и максимум
`argmin, argmax`	Индексы минимального и максимального элементов
`cumsum`	Нарастающая сумма с начальным значением 0
`cumprod`	Нарастающее произведение с начальным значением 1

Методы булевых массивов

В вышеупомянутых методах булевы значения приводятся к 1 (`True`) и 0 (`False`), поэтому функция `sum` часто используется для подсчета значений `True` в булевом массиве:

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # количество положительных значений
Out[191]: 42
```

Но существует еще два метода, `any` и `all`, особенно полезных в случае булевых массивов. Метод `any` проверяет, есть ли в массиве хотя бы одно значение, равное `True`, а `all` – все ли значения в массиве равны `True`:

```
In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

Эти методы работают и для небулевых массивов, и тогда все отличные от нуля элементы считаются равными True.

Сортировка

Как и встроенные в Python списки, массивы NumPy можно сортировать на месте методом sort:

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938, 1.24 , -0.1357, 1.43 , -0.8469])

In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357, 0.6095, 1.24 , 1.43 ])
```

Любой одномерный участок многомерного массива можно отсортировать на месте, передав методу sort номер оси:

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033, 1.2636],
       [-0.9616, -0.4457, 0.4684],
       [-1.8245,  0.6254, 1.0229],
       [-0.3501,  0.0909, 1.1074],
       [-1.7415, -0.8948, 0.218 ]])
```

Метод верхнего уровня np.sort возвращает отсортированную копию массива, а не сортирует массив на месте. Чтобы не мудрствуя лукаво вычислить квантили массива, нужно отсортировать его и выбрать значение с конкретным рангом:

```
In [203]: large_arr = randn(1000)

In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # 5%-ный квантиль
Out[205]: -1.5311513550102103
```

Дополнительные сведения о методах сортировки в NumPy и о более сложных приемах, например косвенной сортировке, см. в приложении А. В библиотеке pandas есть еще несколько операций, относящихся к сортировке (например, сортировка таблицы по одному или нескольким столбцам).

Устранение дубликатов и другие теоретико-множественные операции

В NumPy имеются основные теоретико-множественные операции для одномерных массивов. Пожалуй, самой употребительной является `np.unique`, она возвращает отсортированное множество уникальных значений в массиве:

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

Сравните `np.unique` с альтернативой на чистом Python:

```
In [210]: sorted(set(names))
Out[210]: ['Bob', 'Joe', 'Will']
```

Функция `np.in1d` проверяет, присутствуют ли значения из одного массива в другом, и возвращает булев массив:

```
In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [182]: np.in1d(values, [2, 3, 6])
Out[182]: array([ True, False, False, True, True, False, True], dtype=bool)
```

В табл. 4.6 перечислены все теоретико-множественные функции, имеющиеся в NumPy.

Таблица 4.6. Теоретико-множественные операции с массивами

Метод	Описание
`unique(x)`	Вычисляет отсортированное множество уникальных элементов
`intersect1d(x, y)`	Вычисляет отсортированное множество элементов, общих для x и y

Таблица 4.6 (окончание)

Метод	Описание
`union1d(x, y)`	Вычисляет отсортированное объединение элементов
`in1d(x, y)`	Вычисляет булев массив, показывающий, какие элементы x встречаются в y
`setdiff1d(x, y)`	Вычисляет разность множеств, т. е. элементы, принадлежащие x, но не принадлежащие y
`setxor1d(x, y)`	Симметрическая разность множеств; элементы, принадлежащие одному массиву, но не обоим сразу

4.4. Файловый ввод-вывод массивов

NumPy умеет сохранять на диске и загружать с диска данные в текстовом или двоичном формате. В этом разделе мы рассмотрим только встроенный в NumPy двоичный формат, поскольку для загрузки текстовых и табличных данных большинство пользователей предпочитает pandas и другие инструменты (дополнительные сведения см. в главе 6).

`np.save` и `np.load` – основные функции для эффективного сохранения и загрузки данных с диска. По умолчанию массивы хранятся в несжатом двоичном формате в файле с расширением *.npy*.

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
```

Если путь к файлу не заканчивается суффиксом `.npy`, то последний будет добавлен. Хранящийся на диске массив можно загрузить в память функцией `np.load`:

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Можно сохранить несколько массивов в zip-архиве с помощью функции `np.savez`, которой массивы передаются в виде именованных аргументов:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

При считывании *npz*-файла мы получаем похожий на словарь объект, который отложенно загружает отдельные массивы:

```
In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Если данные хорошо сжимаются, то можно вместо этого использовать метод `numpy.savez_compressed`:

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

4.5. Линейная алгебра

Линейно-алгебраические операции: умножение и разложение матриц, вычисление определителей и другие – важная часть любой библиотеки для работы с массивами. В отличие от некоторых языков, например MATLAB, в NumPy применение оператора * к двум двумерным массивам вычисляет поэлементное, а не матричное произведение. А для перемножения матриц имеется функция dot – в виде как метода массива, так и функции в пространстве имен numpy:

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])

In [226]: y
Out[226]:
array([[  6., 23.],
       [ -1., 7.],
       [  8., 9.]])

In [227]: x.dot(y)
Out[227]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

x.dot(y) эквивалентно np.dot(x, y):

```
In [228]: np.dot(x, y)
Out[228]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

Произведение двумерного массива и одномерного массива подходящего размера дает одномерный массив:

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([ 6., 15.])
```

Символ @ (появившийся в версии Python 3.5) также может использоваться в роли инфиксного оператора умножения матриц:

```
In [230]: x @ np.ones(3)
Out[230]: array([ 6., 15.])
```

В модуле numpy.linalg имеется стандартный набор алгоритмов, в частности разложение матриц, нахождение обратной матрицы и вычисление определителя. Все они реализованы на базе тех же отраслевых библиотек, написанных

на Fortran, которые используются и в других языках, например MATLAB и R: BLAS, LAPACK и, возможно (в зависимости от сборки NumPy), библиотеки MKL (Math Kernel Library), поставляемой компанией Intel:

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)

In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[ 933.1189,    871.8258, -1417.6902, -1460.4005,  1782.1391],
       [ 871.8258,    815.3929, -1325.9965, -1365.9242,  1666.9347],
       [-1417.6902, -1325.9965,  2158.4424,  2222.0191, -2711.6822],
       [-1460.4005, -1365.9242,  2222.0191,  2289.0575, -2793.422 ],
       [ 1782.1391,  1666.9347, -2711.6822, -2793.422 ,  3409.5128]])

In [235]: mat.dot(inv(mat))
Out[235]:
array([[ 1., 0., -0., -0., -0.],
       [-0., 1.,  0.,  0.,  0.],
       [ 0., 0.,  1.,  0.,  0.],
       [-0., 0.,  0.,  1., -0.],
       [-0., 0.,  0.,  0.,  1.]])

In [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[-1.6914,  4.38  ,  0.1757,  0.4075, -0.7838],
       [ 0.    , -2.6436,  0.1939, -3.072 , -1.0702],
       [ 0.    ,  0.    , -0.8138,  1.5414,  0.6155],
       [ 0.    ,  0.    ,  0.    , -2.6445, -2.1669],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.0002]])
```

Выражение `X.T.dot(X)` вычисляет произведение матрицы `X` на транспонированную к ней матрицу `X.T`.

В табл. 4.7 перечислены наиболее употребительные линейно-алгебраические функции.

Таблица 4.7. Наиболее употребительные функции из модуля `numpy.linalg`

Функция	Описание
`diag`	Возвращает диагональные элементы квадратной матрицы в виде одномерного массива или преобразует одномерный массив в квадратную матрицу, в которой все элементы, кроме находящихся на главной диагонали, равны нулю
`dot`	Вычисляет произведение матриц
`trace`	Вычисляет след матрицы – сумму диагональных элементов
`det`	Вычисляет определитель матрицы
`eig`	Вычисляет собственные значения и собственные векторы квадратной матрицы

Таблица 4.7 (окончание)

Функция	Описание
`inv`	Вычисляет обратную матрицу
`pinv`	Вычисляет псевдообратную матрицу Мура-Пенроуза для квадратной матрицы
`qr`	Вычисляет QR-разложение
`svd`	Вычисляет сингулярное разложение (SVD)
`solve`	Решает линейную систему `Ax = b`, где `A` – квадратная матрица
`lstsq`	Вычисляет решение уравнения `y = Xb` по методу наименьших квадратов

4.6. Генерация псевдослучайных чисел

Модуль `numpy.random` дополняет встроенный модуль `random` функциями, которые генерируют целые массивы случайных чисел с различными распределениями вероятности. Например, с помощью функции можно получить случайный массив 4×4 с нормальным распределением:

```
In [238]: samples = np.random.normal(size=(4, 4))
In [239]: samples
Out[239]:
array([[ 0.5732, 0.1933,  0.4429,  1.2796],
       [ 0.575 , 0.4339, -0.7658, -1.237 ],
       [-0.5367, 1.8545, -0.92  , -0.1082],
       [ 0.1525, 0.9435, -1.0953, -0.144 ]])
```

Встроенный в Python модуль `random` умеет выдавать только по одному случайному числу за одно обращение. Ниже видно, что `numpy.random` более чем на порядок быстрее стандартного модуля при генерации очень больших выборок:

```
In [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Мы называем эти числа *псевдослучайными*, потому что они генерируются детерминированным алгоритмом на основе *начального значения*, которое можно изменить с помощью метода `np.random.seed`:

```
In [244]: np.random.seed(1234)
```

В функциях генерации данных из модуля `numpy.random` используется глобальное начальное значение. Чтобы не вводить глобальное состояние, можно воспользоваться методом `numpy.random.RandomState`, который порождает случайное число, не зависящее от прочих:

```
In [245]: rng = np.random.RandomState(1234)
In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206, 0.8872, 0.8596,
       -0.6365,  0.0157, -2.2427])
```

В табл. 4.8 приведен неполный перечень функций, имеющихся в модуле `numpy.random`. В следующем разделе я приведу несколько примеров их использования для генерации больших случайных массивов.

Таблица 4.8. Наиболее употребительные функции из модуля `numpy.random`

Функция	Описание
`seed`	Задает начальное значение генератора случайных чисел
`permutation`	Возвращает случайную перестановку последовательности или диапазона
`shuffle`	Случайным образом переставляет последовательность на месте
`rand`	Случайная выборка с равномерным распределением
`randint`	Случайная выборка целого числа из заданного диапазона
`randn`	Случайная выборка с нормальным распределением со средним 0 и стандартным отклонением 1 (интерфейс похож на MATLAB)
`binomial`	Случайная выборка с биномиальным распределением
`normal`	Случайная выборка с нормальным (гауссовым) распределением
`beta`	Случайная выборка с бета-распределением
`chisquare`	Случайная выборка с распределением хи-квадрат
`gamma`	Случайная выборка с гамма-распределением
`uniform`	Случайная выборка с равномерным распределением на полуинтервале [0, 1)

4.7. Пример: случайное блуждание

Проиллюстрируем операции с массивами на примере случайного блуждания (https://en.wikipedia.org/wiki/Random_walk). Сначала рассмотрим простое случайное блуждание с начальной точкой 0 и шагами 1 и –1, выбираемыми с одинаковой вероятностью.

Вот реализация одного случайного блуждания с 1000 шагов на чистом Python с помощью встроенного модуля `random`:

```
In [247]: import random
   .....: position = 0
   .....: walk = [position]
   .....: steps = 1000
   .....: for i in range(steps):
   .....:     step = 1 if random.randint(0, 1) else -1
   .....:     position += step
   .....:     walk.append(position)
   .....:
```

На рис. 4.4 показаны первые 100 значений такого случайного блуждания.

```
In [249]: plt.plot(walk[:100])
```

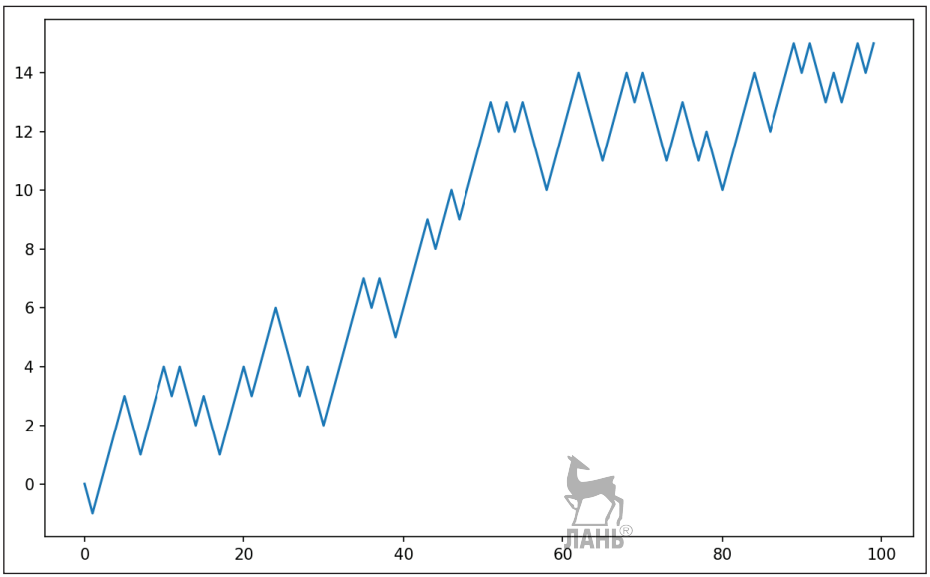


Рис. 4.4. Простое случайное блуждание

Наверное, вы обратили внимание, что `walk` – это просто нарастающая сумма случайных шагов, которую можно вычислить как выражение-массив. Поэтому я воспользуюсь модулем `np.random`, чтобы за один присест подбросить 1000 монет с исходами 1 и –1 и вычислить нарастающую сумму:

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps)

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()
```

Теперь можно приступить к вычислению статистики, например минимального и максимального значений на траектории блуждания:

```
In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

Более сложная статистика – *момент первого пересечения* – это шаг, на котором траектория случайного блуждания впервые достигает заданного значения. В данном случае мы хотим знать, сколько времени потребуется на то, чтобы удалиться от начала (нуля) на десять единиц в любом направлении.

Выражение np.abs(walk) >= 10 дает булев массив, показывающий, в какие моменты блуждание достигало или превышало 10, однако нас интересует индекс *первого* значения 10 или –10. Его можно вычислить с помощью функции argmax, которая возвращает индекс первого максимального значения в булевом массиве (True – максимальное значение):

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[258]: 37
```

Отметим, что использование здесь argmax не всегда эффективно, потому что она всегда просматривает весь массив. В данном частном случае мы знаем, что первое же встретившееся значение True является максимальным.

Моделирование сразу нескольких случайных блужданий

Если бы нам требовалось смоделировать много случайных блужданий, скажем 5000, то это можно было бы сделать путем совсем небольшой модификации приведенного выше кода. Если функциям из модуля numpy.random передать 2-кортеж, то они сгенерируют двумерный массив случайных чисел и мы сможем вычислить нарастающие суммы по строкам, т. е. все 5000 случайных блужданий за одну операцию:

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 или 1

In [261]: steps = np.where(draws > 0, 1, -1)

In [262]: walks = steps.cumsum(1)

In [263]: walks
Out[263]:
array([[  1,  0,  1, ...,   8,   7,   8],
       [  1,  0, -1, ...,  34,  33,  32],
       [  1,  0, -1, ...,   4,   5,   4],
       ...,
       [  1,  2,  1, ...,  24,  25,  26],
       [  1,  2,  3, ...,  14,  13,  14],
       [ -1, -2, -3, ..., -24, -23, -22]])
```

Теперь мы можем вычислить максимум и минимум по всем блужданиям:

```
In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```

Вычислим для этих блужданий минимальный момент первого пересечения с уровнем 30 или –30. Это не так просто, потому что не в каждом блуждании уровень 30 достигается. Проверить, так ли это, можно с помощью метода any:

```
In [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False, True, False, ..., False, True, False], dtype=bool)

In [268]: hits30.sum()         # Сколько раз достигалось 30 или -30
Out[268]: 3410
```

Имея этот булев массив, мы можем выбрать те строки `walks`, в которых достигается уровень 30 (по абсолютной величине), и вызвать `argmax` вдоль оси 1 для получения моментов пересечения:

```
In [269]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [270]: crossing_times.mean()
Out[270]: 498.88973607038122
```

Поэкспериментируйте с другими распределениями шагов, не ограничиваясь подбрасыванием симметричной монеты. Всего-то и нужно взять другую функцию генерации случайных чисел, например `normal`, для генерации шагов с нормальным распределением с заданными средним и стандартным отклонениями:

```
In [271]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```

4.8. Заключение

Хотя бóльшая часть книги посвящена выработке навыков манипулирования данными в pandas, мы и дальше будем работать с массивами в таком же стиле. В приложении A более глубоко рассмотрим возможности NumPy и расскажем о дополнительных приемах вычислений с массивами.

Глава 5. Первое знакомство с pandas

Библиотека pandas будет нашим основным инструментом в оставшейся части книги. Она содержит структуры данных и средства манипуляции данными, спроектированные с целью максимально упростить и ускорить очистку и анализ данных в Python. Pandas часто используется в сочетании с такими инструментами численных расчетов, как NumPy и SciPy, аналитическими библиотеками statsmodels и scikit-learn и библиотеками визуализации данных типа matplotlib. Pandas переняла от NumPy многое из идиоматического стиля работы с массивами, в особенности функции манипуляции массивами и стремление избегать циклов `for` при обработке данных.

Хотя pandas взяла на вооружение многие идиомы кодирования NumPy, между этими библиотеками есть важное различие. Pandas предназначена для работы с табличными или неоднородными данными. Напротив, NumPy больше подходит для работы с однородными числовыми данными, организованными в виде массивов.

Pandas стала проектом с открытым исходным кодом в 2010 году и с тех пор превратилась в довольно большую библиотеку, применяемую в самых разных практических приложениях. Количество соразработчиков перевалило за 800, все они так или иначе помогают развивать проект, который используют в повседневной работе.

В этой книге используется следующее соглашение об импорте pandas:

```
In [1]: import pandas as pd
```

Стало быть, встретив в коде `pd.`, знайте, что имеется в виду pandas. Возможно, вы захотите также импортировать в локальное пространство имен классы Series и DataFrame, потому что они используются часто:

```
In [2]: from pandas import Series, DataFrame
```

5.1. Введение в структуры данных pandas

Чтобы начать работу с pandas, вы должны освоить две основные структуры данных: *Series* и *DataFrame*. Они, конечно, не являются универсальным решением любой задачи, но все же образуют солидную и простую для использования основу большинства приложений.

Объект Series

Series – одномерный, похожий на массив объект, содержащий последовательность данных (типов, похожих на поддерживаемые NumPy) и ассоциированный с ним массив меток, который называется *индексом*. Простейший объект Series состоит только из массива данных:

```
In [11]: obj = Series([4, 7, -5, 3])

In [12]: obj
Out[13]:
0    4
1    7
2   -5
3    3
dtype: int64
```


В строковом представлении Series, отображаемом в интерактивном режиме, индекс находится слева, а значения – справа. Поскольку мы не задали индекс для данных, то по умолчанию создается индекс, состоящий из целых чисел от 0 до N – 1 (где N – длина массива данных). Имея объект Series, получить представление самого массива и его индекса можно с помощью атрибутов `values` и `index` соответственно:

```
In [13]: obj.values
Out[13]: array([ 4, 7, -5, 3])

In [14]: obj.index  # как range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```


Часто желательно создать объект Series с индексом, идентифицирующим каждый элемент данных:

```
In [15]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64

In [17]: obj2.index
Out[17]: Index([d, b, a, c], dtype=object)
```

В отличие от массивов NumPy, для выборки одного или нескольких значений можно использовать метки в индексе:

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

Здесь `['c', 'a', 'd']` интерпретируется как список индексов, хотя он содержит не целые числа, а строки.

Функции NumPy или похожие на них операции, например фильтрация с помощью булева массива, скалярное умножение или применение математических функций, сохраняют связь между индексом и значением:


```
In [21]: obj2[obj2 > 0]
Out[21]:
d   6
b   7
c   3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d   12
b   14
a  -10
c    6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d    403.428793
b   1096.633158
a      0.006738
c     20.085537
dtype: float64
```


Объект Series можно также представлять как упорядоченный словарь фиксированной длины, поскольку он отображает индекс на данные. Его можно передавать многим функциям, ожидающим получить словарь:

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

Если имеется словарь Python, содержащий данные, то из него можно создать объект Series:

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
dtype: int64
```

Если передается только словарь, то в получившемся объекте Series ключи будут храниться в индексе по порядку:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California      NaN
Ohio        35000.0
Oregon      16000.0
Texas       71000.0
dtype: float64
```

В данном случае три значения, найденных в sdata, помещены в соответствующие им позиции, а для метки 'California' никакого значения не нашлось, поэтому ей соответствует признак NaN (не число), которым в pandas обозначаются отсутствующие значения. Поскольку строки 'Utah' не было в списке states, то ее нет и в результирующем объекте.

Говоря об отсутствующих данных, я иногда буду употреблять термин «NA». Для распознавания отсутствующих данных в pandas следует использовать функции isnull и notnull:

```
In [32]: pd.isnull(obj4)
Out[32]:
California   True
Ohio        False
Oregon      False
Texas       False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California  False
Ohio         True
Oregon       True
```

```
Texas        True
dtype: bool
```

У объекта Series есть также методы экземпляра:

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
```

Более подробно работа с отсутствующими данными будет обсуждаться в главе 7.

Во многих приложениях сказано, что при выполнении арифметических операций объект Series автоматически выравнивает данные по индексной метке:

```
In [35]: obj3
Out[35]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
dtype: int64
```


```
In [36]: obj4
Out[36]:
California     NaN
Ohio       35000.0
Oregon     16000.0
Texas      71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
Out[37]:
California     NaN
Ohio       70000.0
Oregon     32000.0
Texas     142000.0
Utah           NaN
dtype: float64
```

Вопрос о выравнивании данных будет подробнее рассмотрен ниже. Если у вас имеется опыт работы с базами данных, то можете считать, что это аналог операции соединения.

И у самого объекта Series, и у его индекса имеется атрибут `name`, тесно связанный с другими частями функциональности pandas:

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'
```

```
In [40]: obj4
Out[40]:
state
California       NaN
Ohio         35000.0
Oregon       16000.0
Texas        71000.0
Name: population, dtype: float64
```

Индекс объекта Series можно изменить на месте с помощью присваивания:

```
In [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

Объект DataFrame

Объект DataFrame представляет табличную структуру данных, состоящую из упорядоченной коллекции столбцов, причем типы значений (числовой, строковый, булев и т. д.) в разных столбцах могут различаться. В объекте DataFrame хранятся два индекса: по строкам и по столбцам. Можно считать, что это словарь объектов Series, имеющих общий индекс. Внутри объекта данные хранятся в виде одного или нескольких двумерных блоков, а не в виде списка, словаря или еще какой-нибудь коллекции одномерных массивов. Технические детали внутреннего устройства DataFrame выходят за рамки этой книги.

Хотя в DataFrame данные хранятся в двумерном формате, в виде таблицы, нетрудно представить и данные более высокой размерности, если воспользоваться иерархическим индексированием. Эту тему мы обсудим в следующем разделе, она лежит в основе многих продвинутых механизмов обработки данных в pandas.

Есть много способов сконструировать объект DataFrame, один из самых распространенных – на основе словаря списков одинаковой длины или массивов NumPy:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

Получившемуся DataFrame автоматически будет назначен индекс, как и в случае Series, и столбцы расположатся по порядку:

```
In [45]: frame
Out[45]:
  pop  state year
0 1.5   Ohio 2000
1 1.7   Ohio 2001
2 3.6   Ohio 2002
3 2.4 Nevada 2001
4 2.9 Nevada 2002
5 3.2 Nevada 2003
```

В Jupyter-блокноте объекты DataFrame отображаются в виде HTML-таблицы, более удобной для браузера.

Для больших объектов DataFrame метод `head` отбирает только первые пять строк:

```
In [46]: frame.head()
Out[46]:
  pop  state year
0 1.5   Ohio 2000
1 1.7   Ohio 2001
2 3.6   Ohio 2002
3 2.4 Nevada 2001
4 2.9 Nevada 2002
```

Если задать последовательность столбцов, то столбцы DataFrame расположатся строго в указанном порядке:

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
  year  state pop
0 2000   Ohio 1.5
1 2001   Ohio 1.7
2 2002   Ohio 3.6
3 2001 Nevada 2.4
4 2002 Nevada 2.9
5 2003 Nevada 3.2
```

Если запросить столбец, которого нет в `data`, то он будет заполнен значениями `NaN`:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])
```

```
In [49]: frame2
Out[49]:
      year  state pop debt
one   2000   Ohio 1.5  NaN
two   2001   Ohio 1.7  NaN
three 2002   Ohio 3.6  NaN
four  2001 Nevada 2.4  NaN
five  2002 Nevada 2.9  NaN
six   2003 Nevada 3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Столбец DataFrame можно извлечь как объект Series, воспользовавшись нотацией словарей или помощью атрибута:

```
In [51]: frame2['state']
Out[51]:
one      Ohio
two      Ohio
three    Ohio
four   Nevada
five   Nevada
six    Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one     2000
two     2001
three   2002
four    2001
five    2002
six     2003
Name: year, dtype: int64
```

Возможность доступа к столбцам как к атрибутам (например, `frame2.year`) и автозавершение имен столбцов по нажатии **Tab** предоставляются в IPython для удобства.
Синтаксис `frame2[column]` работает для любого имени столбца, а `frame2.column` – только когда имя столбца – допустимое имя переменной Python.

Отметим, что возвращенный объект Series имеет тот же индекс, что и DataFrame, а его атрибут `name` установлен соответствующим образом.

Строки также можно извлечь по позиции или по имени с помощью специального атрибута `loc` (подробнее об этом ниже):

```
In [53]: frame2.loc['three']
Out[53]:
year    2002
```

```
state   Ohio
pop      3.6
debt     NaN
Name: three, dtype: object
```

Столбцы можно модифицировать путем присваивания. Например, пустому столбцу 'debt' можно было бы присвоить скалярное значение или массив значений:

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
      year  state pop debt
one   2000   Ohio 1.5 16.5
two   2001   Ohio 1.7 16.5
three 2002   Ohio 3.6 16.5
four  2001 Nevada 2.4 16.5
five  2002 Nevada 2.9 16.5
six   2003 Nevada 3.2 16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
      year   state pop debt
one   2000   Ohio  1.5 0.0
two   2001   Ohio  1.7 1.0
three 2002   Ohio  3.6 2.0
four  2001 Nevada  2.4 3.0
five  2002 Nevada  2.9 4.0
six   2003 Nevada  3.2 5.0
```

Когда столбцу присваивается список или массив, длина значения должна совпадать с длиной DataFrame. Если же присваивается объект Series, то его метки будут точно выровнены с индексом DataFrame, а в «дырки» будут вставлены значения NA:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
      year  state pop debt
one   2000   Ohio 1.5  NaN
two   2001   Ohio 1.7 -1.2
three 2002   Ohio 3.6  NaN
four  2001 Nevada 2.4 -1.5
five  2002 Nevada 2.9 -1.7
six   2003 Nevada 3.2  NaN
```

Присваивание несуществующему столбцу приводит к созданию нового столбца. Для удаления столбцов служит ключевое слово `del`, как и в обычном словаре.

Для демонстрации работы `del` я сначала добавлю новый столбец булевых признаков, показывающих, находится ли в столбце `state` значение `'Ohio'`:

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
      year  state pop debt eastern
one   2000   Ohio 1.5  NaN    True
two   2001   Ohio 1.7 -1.2    True
three 2002   Ohio 3.6  NaN    True
four  2001 Nevada 2.4 -1.5   False
five  2002 Nevada 2.9 -1.7   False
six   2003 Nevada 3.2  NaN   False
```

Новый столбец нельзя создать, пользуясь синтаксисом `frame2.eastern`.

Затем для удаления этого столбца я воспользуюсь методом `del`:

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Столбец, возвращенный в ответ на запрос к DataFrame по индексу, является *представлением*, а не копией данных. Следовательно, любые модификации этого объекта Series найдут отражение в DataFrame. Чтобы скопировать столбец, нужно вызвать метод `copy` объекта Series.

Еще одна распространенная форма данных – словарь словарей:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Если передать вложенный словарь объекту DataFrame, то pandas интерпретирует ключи внешнего словаря как столбцы, а ключи внутреннего словаря – как индексы строк:

```
In [66]: frame3 = DataFrame(pop)

In [67]: frame3
Out[67]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

Объект DataFrame можно транспонировать (переставить местами строки и столбцы), воспользовавшись таким же синтаксисом, как для словарей NumPy:

```
In [68]: frame3.T
Out[68]:
       2000  2001  2002
Nevada  NaN   2.4   2.9
Ohio    1.5   1.7   3.6
```

Ключи внутренних словарей объединяются и сортируются для образования индекса результата. Однако этого не происходит, если индекс задан явно:

```
In [69]: DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
     Nevada  Ohio
2001    2.4   1.7
2002    2.9   3.6
2003    NaN   NaN
```

Словари объектов Series интерпретируются очень похоже:

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: DataFrame(pdata)
Out[71]:
     Nevada  Ohio
2000    NaN   1.5
2001    2.4   1.7
```

Полный перечень возможных аргументов конструктора DataFrame приведен в табл. 5.1.

Если у объектов, возвращаемых при обращении к атрибутам `index` и `columns` объекта DataFrame, установлен атрибут `name`, то он также выводится:

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [73]: frame3
Out[73]:
state  Nevada   Ohio
year
2000      NaN    1.5
2001      2.4    1.7
2002      2.9    3.6
```

Как и в случае Series, атрибут `values` возвращает данные, хранящиеся в DataFrame, в виде двумерного массива ndarray:

```
In [74]: frame3.values
Out[74]:
array([[ nan, 1.5],
```

```
       [ 2.4, 1.7],
       [ 2.9, 3.6]])
```

Если у столбцов DataFrame разные типы данных, то dtype массива `values` будет выбран так, чтобы охватить все столбцы:

```
In [75]: frame2.values
Out[75]:
array([[2000, Ohio, 1.5, nan],
       [2001, Ohio, 1.7, -1.2],
       [2002, Ohio, 3.6, nan],
       [2001, Nevada, 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Таблица 5.1. Аргументы конструктора DataFrame

Тип	Примечания
Двумерный ndarray	Матрица данных, дополнительно можно передать метки строк и столбцов
Словарь массивов, списков или кортежей	Каждая последовательность становится столбцом объекта DataFrame. Все последовательности должны быть одинаковой длины
Структурированный массив NumPy	Интерпретируется так же, как «словарь массивов»
Словарь объектов Series	Каждое значение становится столбцом. Если индекс явно не задан, то индексы объектов Series объединяются и образуют индекс строк результата
Словарь словарей	Каждый внутренний словарь становится столбцом. Ключи объединяются и образуют индекс строк, как в случае «словаря объектов Series»
Список словарей или объектов Series	Каждый элемент списка становится строкой объекта DataFrame. Объединение ключей словаря или индексов объектов Series становится множеством меток столбцов DataFrame
Список списков или кортежей	Интерпретируется так же, как «двумерный ndarray»
Другой объект DataFrame	Используются индексы DataFrame, если явно не заданы другие индексы
Объект NumPy MaskedArray	Как «двумерный ndarray» с тем отличием, что замаскированные значения становятся отсутствующими в результирующем объекте DataFrame

Индексные объекты

В индексных объектах pandas хранятся метки вдоль осей и прочие метаданные (например, имена осей). Любой массив или иная последовательность меток, указанная при конструировании Series или DataFrame, преобразуется в объект Index:

```
In [76]: obj = Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index([a, b, c], dtype=object)
```

```
In [79]: index[1:]
Out[79]: Index([b, c], dtype=object)
```

Индексные объекты неизменяемы, т. е. пользователь не может их модифицировать:

```
index[1] = 'd' # TypeError
```

Неизменяемость важна, для того чтобы несколько структур данных могли совместно использовать один и тот же индексный объект, не опасаясь его повредить:


```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

Некоторые пользователи редко используют преимущества, предоставляемые индексами, но, поскольку существуют операции, которые возвращают результат, содержащий индексированные данные, важно понимать, как индексы работают.

Индексный объект не только похож на массив, но и ведет себя как множество фиксированного размера:


```
In [85]: frame3
Out[85]:
state Nevada Ohio
year
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

В отличие от множеств Python, индекс в pandas может содержать повторяющиеся метки:

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Если при выборке указана повторяющаяся метка, то будут выбраны все элементы с такой меткой.

У любого объекта Index есть ряд свойств и методов для ответа на типичные вопросы о хранящихся в нем данных. Они перечислены в табл. 5.2.

Таблица 5.2. Методы и свойства объекта Index

Метод	Описание
`append`	Конкатенирует с дополнительными индексными объектами, порождая новый объект Index
`diff`	Вычисляет теоретико-множественную разность, представляя ее в виде индексного объекта
`intersection`	Вычисляет теоретико-множественное пересечение
`union`	Вычисляет теоретико-множественное объединение
`isin`	Вычисляет булев массив, показывающий, содержится ли каждое значение индекса в переданной коллекции
`delete`	Вычисляет новый индексный объект, получающийся после удаления элемента с индексом `i`
`drop`	Вычисляет новый индексный объект, получающийся после удаления переданных значений
`insert`	Вычисляет новый индексный объект, получающийся после вставки элемента в позицию с индексом `i`
`is_monotonic`	Возвращает True, если каждый элемент больше предыдущего или равен ему
`is_unique`	Возвращает True, если в индексе нет повторяющихся значений
`unique`	Вычисляет массив уникальных значений в индексе

5.2. Базовая функциональность

В этом разделе мы рассмотрим фундаментальные основы взаимодействия с данными, хранящимися в объектах Series и DataFrame. В последующих главах более детально обсудим вопросы анализа и манипуляции данными с применением pandas. Эта книга не задумывалась как исчерпывающая документация по библиотеке pandas, я хотел лишь акцентировать внимание на наиболее важных чертах, оставив не столь употребительные (если не сказать эзотерические) вещи для самостоятельного изучения.

Переиндексация

Для объектов pandas важен метод `reindex`, т. е. возможность создания нового объекта, данные в котором *согласуются* с новым индексом. Рассмотрим пример:

```
In [91]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [92]: obj
Out[92]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Если вызвать reindex для этого объекта Series, то данные будут реорганизованы в соответствии с новым индексом, а если каких-то из имеющихся в этом индексе значений раньше не было, то вместо них будут подставлены отсутствующие значения:

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```


Для упорядоченных данных, например временных рядов, иногда желательно произвести интерполяцию, или восполнение, отсутствующих значений в процессе переиндексации. Это позволяет сделать параметр method; так, если задать для него значение ffill, то будет произведено прямое восполнение значений:

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
In [96]: obj3
Out[96]:
0    blue
2  purple
4  yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0    blue
1    blue
2  purple
3  purple
4  yellow
5  yellow
dtype: object
```

В случае объекта DataFrame функция reindex может изменять строки, столбцы или то и другое. Если ей передать просто последовательность, то в результирующем объекте переиндексируются строки:

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'])
In [99]: frame
Out[99]:
   Ohio Texas California
a    0      1          2
c    3      4          5
d    6      7          8

In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [101]: frame2
Out[101]:
   Ohio Texas California
a   0.0   1.0        2.0
b   NaN   NaN        NaN
c   3.0   4.0        5.0
d   6.0   7.0        8.0
```

Столбцы можно переиндексировать, задав ключевое слово columns:

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
  Texas Utah California
a     1  NaN          2
c     4  NaN          5
d     7  NaN          8
```

В табл. 5.3 приведены более полные сведения об аргументах reindex.

Ниже мы подробно рассмотрим более лаконичный способ переиндексации посредством индексирования метками с помощью loc. Многие только им и пользуются:

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
  Texas Utah California
a   1.0  NaN        2.0
b   NaN  NaN        NaN
c   4.0  NaN        5.0
d   7.0  NaN        8.0
```

Таблица 5.3. Аргументы функции `reindex`

Аргумент	Описание
`index`	Последовательность, которая должна стать новым индексом. Может быть экземпляром `Index` или любой другой структурой данных Python, похожей на последовательность. Экземпляр `Index` будет использован как есть, без копирования
`method`	Метод интерполяции (восполнения), возможные значения приведены в табл. 5.4
`fill_value`	Значение, которое должно подставляться вместо отсутствующих значений, появляющихся в результате переиндексации
`limit`	При прямом или обратном восполнении максимальная длина восполняемой лакуны (выраженная числом элементов)
`tolerance`	При прямом или обратном восполнении максимальная длина восполняемой лакуны (выраженная абсолютным расстоянием)
`level`	Сопоставить с простым объектом `Index` на указанном уровне `MultiIndex`, иначе выбрать подмножество
`copy`	Не копировать данные, если новый индекс эквивалентен старому. По умолчанию `True` (т. е. всегда копировать данные)

Удаление элементов из оси

Удалить один или несколько элементов из оси просто, если имеется индексный массив или список, не содержащий этих значений. Метод `drop` возвращает новый объект, в котором указанные значения удалены из оси:

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a   0.0
b   1.0
c   2.0
d   3.0
e   4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a   0.0
b   1.0
d   3.0
e   4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
a   0.0
b   1.0
e   4.0
dtype: float64
```

В случае DataFrame указанные в индексе значения можно удалить из любой оси. Для иллюстрации сначала создадим объект DataFrame:

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          one two three four
Ohio        0   1     2    3
Colorado    4   5     6    7
Utah        8   9    10   11
New York   12  13    14   15
```

Если вызвать drop, указав последовательность меток, то будут удалены строки с такими метками (ось 0):

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
          one two three four
Utah        8   9    10   11
New York   12  13    14   15
```

Чтобы удалить столбцы, следует передать axis=1 или axis='columns':

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one three four
Ohio        0     2    3
Colorado    4     6    7
Utah        8    10   11
New York   12    14   15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one three
Ohio        0     2
Colorado    4     6
Utah        8    10
New York   12    14
```

Многие функции, подобные drop, которые модифицируют размер или форму Series или DataFrame, могут изменять объект *на месте*, не создавая новый объект:

```
In [115]: obj.drop('c', inplace=True)
In [116]: obj
Out[116]:
a    0.0
b    1.0
d    3.0
```

```
e    4.0
dtype: float64
```

Будьте осторожны с режимом `inplace`, поскольку все удаленные в нем данные уничтожаются.

Доступ по индексу, выборка и фильтрация

Доступ по индексу к объекту Series (`obj[...]`) работает так же, как для массивов NumPy, с тем отличием, что индексами могут быть не только целые, но и любые значения из индекса объекта Series. Вот несколько примеров:

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a   0.0
b   1.0
c   2.0
d   3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
c   2.0
d   3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b   1.0
a   0.0
d   3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b   1.0
d   3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a   0.0
b   1.0
dtype: float64
```

Вырезание с помощью меток отличается от обычного вырезания в Python тем, что конечная точка включается:

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

Установка с помощью этих методов модифицирует соответствующий участок Series:

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Обращение по индексу к DataFrame предназначено для получения одного или нескольких столбцов путем задания одного значения или последовательности:

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....: index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....: columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          one two three four
Ohio        0   1    2     3
Colorado    4   5    6     7
Utah        8   9   10    11
New York   12  13   14    15

In [130]: data['two']
Out[130]:
Ohio      1
Colorado  5
Utah      9
New York 13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three one
Ohio          2   0
Colorado      6   4
```

```
Utah        10   8
New York    14  12
```

У доступа по индексу есть несколько частных случаев. Во-первых, выборка или вырезание данных с помощью булева массива:

```
In [132]: data[:2]
Out[132]:
          one two three four
Ohio        0   1     2    3
Colorado    4   5     6    7

In [133]: data[data['three'] > 5]
Out[133]:
          one two three four
Colorado    4   5     6    7
Utah        8   9    10   11
New York   12  13    14   15
```

Для удобства предоставляется специальный синтаксис выборки строк `data[:2]`. Если передать один элемент или список оператору `[]`, то выбираются столбцы.

Еще одна возможность – доступ по индексу с помощью булева DataFrame, например, порожденного в результате скалярного сравнения:

```
In [134]: data < 5
Out[134]:
             one    two  three   four
Ohio        True   True   True   True
Colorado    True  False  False  False
Utah       False  False  False  False
New York   False  False  False  False

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
             one    two  three   four
Ohio           0      0      0      0
Colorado       0      5      6      7
Utah           8      9     10     11
New York      12     13     14     15
```

Это делает DataFrame синтаксически более похожим на ndarray в данном частном случае.

Выборка с помощью loc и iloc

Для доступа к строкам DataFrame по индексу с помощью меток я ввел специальные индексные операторы `loc` и `iloc`. Они позволяют выбрать подмножество строк и столбцов DataFrame с применением нотации NumPy, используя либо метки строк (`loc`), либо целые числа (`iloc`).

В качестве вступительного примера выберем одну строку и несколько столбцов по меткам:

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two    5
three  6
Name: Colorado, dtype: int64
```

Затем произведем аналогичную выборку, но уже по целочисленным индексам:

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four  11
one    8
two    9
Name: Utah, dtype: int64
```



```
In [139]: data.iloc[2]
Out[139]:
one    8
two    9
three 10
four  11
Name: Utah, dtype: int64
```

```
In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
         four one two
Colorado    7   0   5
Utah       11   8   9
```


Обе функции индексирования работают не только с одиночными метками или списками меток, но и со срезами:

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio     0
Colorado 5
Utah     9
Name: two, dtype: int64
```

```
In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
         one two three
Colorado   0   5     6
Utah       8   9    10
New York  12  13    14
```

Таким образом, существует много способов выборки и реорганизации данных, содержащихся в объекте pandas. Для DataFrame краткая сводка многих

из них приведена в табл. 5.4. Позже мы увидим, что при работе с иерархическими индексами есть ряд дополнительных возможностей.

Проектируя pandas, я подспудно ощущал, что нотация `frame[:, col]` для выборки столбца слишком громоздкая (и провоцирующая ошибки), поскольку выборка столбца – одна из самых часто встречающихся операций. Поэтому я пошел на компромисс и перенес все интересное поведение, связанное с доступом по индексу (в виде числа или метки), в оператор `ix`. На практике это стало причиной множества особых случаев, когда в качестве меток использовались целые числа, поэтому команда разработчиков pandas решила ввести операторы `loc` и `iloc`, чтобы разделить индексный доступ по метке и по целому числу.

Оператор доступа по индексу `ix` все еще существует, но объявлен нежелательным. Я не рекомендую пользоваться им.

Таблица 5.4. Варианты доступа по индексу для объекта DataFrame

Вариант	Примечание
`df[val]`	Выбрать один столбец или последовательность столбцов из DataFrame. Частные случаи: булев массив (фильтрация строк), срез (вырезание строк) или булев DataFrame (установка значений в позициях, удовлетворяющих некоторому критерию)
`df.loc[val]`	Выбрать одну строку или подмножество строк из DataFrame по метке
`obj.loc[:, val]`	Выбрать один столбец или подмножество столбцов по метке
`df.iloc[val1, val2]`	Выбрать строки и столбцы по метке
`df.iloc[where]`	Выбрать одну строку или подмножество строк из DataFrame по целочисленной позиции
`obj.iloc[:, where]`	Выбрать один столбец или подмножество столбцов по целочисленной позиции
`df.loc[where_i, where_j]`	Выбрать строки и столбцы по целочисленной позиции
`df.at[label_i, label_j]`	Выбрать одно скалярное значение по меткам строки и столбца
`df.iat[label_i, label_j]`	Выбрать одно скалярное значение по целочисленным позициям строки и столбца
метод `reindex`	Выбрать строки или столбцы по меткам
методы `get_value, set_value`	Выбрать одно значение по меткам строки и столбца

Целочисленные индексы

Новички часто испытывают затруднения при работе с объектами pandas, индексированными целыми числами, из-за различий с семантикой индексирования встроенных в Python структур данных, таких как списки и кортежи. Например, вряд ли вы ожидаете столкнуться с ошибкой в следующем коде:

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

В данном примере pandas могла бы откатиться к целочисленному индексированию, но в общем случае попытка сделать это приводит к тонким

ошибкам. Здесь мы имеем индекс, содержащий 0, 1, 2, но понять, чего хочет пользователь (индексировать по метке или по позиции), трудно:

```
In [144]: ser
Out[144]:
0   0.0
1   1.0
2   2.0
dtype: float64
```

С другой стороны, если индекс не является целым числом, то никакой неоднозначности не возникает:

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 2.0
```

Чтобы не оставлять место двусмысленности, в случае когда индекс по некоторой оси содержит целые числа, выборка данных всегда производится по метке. А чтобы точно выразить свои намерения, используйте метод `loc` (для доступа по метке) или `iloc` (для доступа по позиции):

```
In [147]: ser[:1]
Out[147]:
0   0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0   0.0
1   1.0
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0   0.0
dtype: float64
```

Арифметические операции и выравнивание данных

Одна из самых важных черт pandas – поведение арифметических операций для объектов с разными индексами. Если при сложении двух объектов обнаруживаются различные пары индексов, то результирующий индекс будет объединением индексов. Для тех, кто имеет опыт работы с базами данных, отметим, что это аналог внешнего соединения по индексным меткам. Рассмотрим простой пример:

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])
```

```
In [152]: s1
Out[152]:
a   7.3
c  -2.5
d   3.4
e   1.5
dtype: float64

In [153]: s2
Out[153]:
a  -2.1
c   3.6
e  -1.5
f   4.0
g   3.1
dtype: float64
```

Сложение этих объектов дает:

```
In [154]: s1 + s2
Out[154]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

Вследствие внутреннего выравнивания данных образуются отсутствующие значения в позициях, для которых не нашлось соответственной пары. Отсутствующие значения распространяются на последующие операции.

В случае DataFrame выравнивание производится как для строк, так и для столбцов:

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                     index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [157]: df1
Out[157]:
          b   c   d
Ohio     0.0 1.0 2.0
Texas    3.0 4.0 5.0
Colorado 6.0 7.0 8.0

In [158]: df2
Out[158]:
         b    d    e
```

```
Utah   0.0  1.0  2.0
Ohio   3.0  4.0  5.0
Texas  6.0  7.0  8.0
Oregon 9.0 10.0 11.0
```

При сложении этих объектов получается DataFrame, индекс и столбцы которого являются объединениями индексов и столбцов слагаемых:

```
In [159]: df1 + df2
Out[159]:
            b   c    d   e
Colorado  NaN NaN  NaN NaN
Ohio      3.0 NaN  6.0 NaN
Oregon    NaN NaN  NaN NaN
Texas     9.0 NaN 12.0 NaN
Utah      NaN NaN  NaN NaN
```

Поскольку столбцы `'c'` и `'e'` не встречаются в обоих объектах DataFrame сразу, то в результирующем объекте они помечены как отсутствующие. То же самое относится к меткам строк, не встречающимся в обоих объектах.

Если сложить объекты DataFrame, не имеющие общих столбцов и строк, то результат будет содержать только значения NaN.

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0  1
1  2

In [163]: df2
Out[163]:
   B
0  3
1  4

In [164]: df1 - df2
Out[164]:
    A   B
0 NaN NaN
1 NaN NaN
```

Восполнение значений в арифметических методах

При выполнении арифметических операций с объектами, проиндексированными по-разному, иногда желательно поместить специальное значение, например 0, в позицию операнда, которым в другом операнде соответствует отсутствующая позиция:

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list('abcd'))
In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
    a   b    c    d
0 0.0 1.0  2.0  3.0
1 4.0 5.0  6.0  7.0
2 8.0 9.0 10.0 11.0

In [169]: df2
Out[169]:
     a    b    c    d    e
0  0.0  1.0  2.0  3.0  4.0
1  5.0  NaN  7.0  8.0  9.0
2 10.0 11.0 12.0 13.0 14.0
3 15.0 16.0 17.0 18.0 19.0
```


Сложение этих объектов порождает отсутствующие значения в позициях, которые имеются не в обоих операндах:

```
In [170]: df1 + df2
Out[170]:
     a    b    c    d   e
0  0.0  2.0  4.0  6.0 NaN
1  9.0  NaN 13.0 15.0 NaN
2 18.0 20.0 22.0 24.0 NaN
3  NaN  NaN  NaN  NaN NaN
```

Теперь я вызову метод `add` объекта `df1` и передам ему объект `df2` и значение параметра `fill_value`:

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
     a    b    c    d   e
0  0.0  2.0  4.0  6.0  4.0
1  9.0  5.0 13.0 15.0  9.0
2 18.0 20.0 22.0 24.0 14.0
3 15.0 16.0 17.0 18.0 19.0
```


В табл. 5.5 приведен перечень арифметических методов объектов Series и DataFrame. У каждого из них имеется вариант, начинающийся буквой `r`, в котором аргументы поменяли местами. Поэтому следующие два предложения эквивалентны:

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
```

```
1 0.250000 0.200000 0.166667 0.142857
2 0.125000 0.111111 0.100000 0.090909
```

```
In [173]: df1.rdiv(1)
Out[173]:
         a        b        c        d
0      inf 1.000000 0.500000 0.333333
1 0.250000 0.200000 0.166667 0.142857
2 0.125000 0.111111 0.100000 0.090909
```

Точно так же, выполняя переиндексацию объекта Series или DataFrame, можно указать восполняемое значение:

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
    a   b    c    d  e
0 0.0 1.0  2.0  3.0  0
1 4.0 5.0  6.0  7.0  0
2 8.0 9.0 10.0 11.0  0
```

Таблица 5.5. Гибкие арифметические методы

Метод	Описание
`add`, `radd`	Сложение (+)
`sub`, `rsub`	Вычитание (-)
`div`, `rdiv`	Деление (/)
`floordiv`, `rfloordiv`	Деление с отсечением (//)
`mul`, `rmul`	Умножение (*)
`pow`, `rpow`	Возведение в степень (**)

Операции между DataFrame и Series

Как и в случае массивов NumPy, арифметические операции между DataFrame и Series корректно определены. В качестве пояснительного примера рассмотрим вычисление разности между двумерным массивом и одной из его строк:

```
In [175]: arr = np.arange(12.).reshape((3, 4))
```

```
In [176]: arr
Out[176]:
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])
```

```
In [178]: arr - arr[0]
Out[178]:
array([[ 0.,  0.,  0.,  0.],
```

```
       [ 4., 4., 4., 4.],
       [ 8., 8., 8., 8.]])
```

Когда мы вычитаем `arr[0]` из `arr`, операция производится один раз для каждой строки. Это называется *укладыванием* и подробно объясняется в приложении А, поскольку имеет отношение к массивам NumPy вообще. Операции между DataFrame и Series аналогичны:

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [180]: series = frame.iloc[0]

In [181]: frame
Out[181]:
         b    d    e
Utah   0.0  1.0  2.0
Ohio   3.0  4.0  5.0
Texas  6.0  7.0  8.0
Oregon 9.0 10.0 11.0

In [182]: series
Out[182]:
b   0.0
d   1.0
e   2.0
Name: Utah, dtype: float64
```

По умолчанию при выполнении арифметических операций между DataFrame и Series индекс Series сопоставляется со столбцами DataFrame, а укладываются строки:

```
In [183]: frame - series
Out[183]:
         b   d   e
Utah   0.0 0.0 0.0
Ohio   3.0 3.0 3.0
Texas  6.0 6.0 6.0
Oregon 9.0 9.0 9.0
```

Если какой-нибудь индекс не найден либо в столбцах DataFrame, либо в индексе Series, то объекты переиндексируются для образования объединения:

```
In [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])
In [185]: frame + series2
Out[185]:
         b   d    e   f
Utah   0.0 NaN  3.0 NaN
Ohio   3.0 NaN  6.0 NaN
Texas  6.0 NaN  9.0 NaN
Oregon 9.0 NaN 12.0 NaN
```

Если вы хотите вместо этого сопоставлять строки, а укладывать столбцы, то должны будете воспользоваться каким-нибудь арифметическим методом. Например:

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
         b    d    e
Utah   0.0  1.0  2.0
Ohio   3.0  4.0  5.0
Texas  6.0  7.0  8.0
Oregon 9.0 10.0 11.0

In [188]: series3
Out[188]:
Utah    1.0
Ohio    4.0
Texas   7.0
Oregon 10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          b   d   e
Utah   -1.0 0.0 1.0
Ohio   -1.0 0.0 1.0
Texas  -1.0 0.0 1.0
Oregon -1.0 0.0 1.0
```

Передаваемый номер оси – это ось, *вдоль которой производится сопоставление*. В данном случае мы хотим сопоставлять с индексом строк DataFrame (`axis='index'` или `axis=0`) и укладывать поперек.

Применение функций и отображение

Универсальные функции NumPy (поэлементные методы массивов) отлично работают и с объектами pandas:

```
In [191]: frame
Out[191]:
               b        d         e
Utah  -0.204708 0.478943 -0.519439
Ohio  -0.555730 1.965781  1.393406
Texas  0.092908 0.281746  0.769023
Oregon 1.246435 1.007189 -1.296221

In [192]: np.abs(frame)
Out[192]:
             b        d        e
Utah   0.204708 0.478943 0.519439
```

```
Ohio   0.555730 1.965781 1.393406
Texas  0.092908 0.281746 0.769023
Oregon 1.246435 1.007189 1.296221
```

Еще одна часто встречающаяся операция – применение функции, определенной для одномерных массивов, к каждому столбцу или строке. Именно это и делает метод `apply` объекта DataFrame:

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b   1.802165
d   1.684034
e   2.689627
dtype: float64
```

Здесь функция `f`, вычисляющая разность между максимальным и минимальным значениями Series, вызывается один раз для каждого столбца `frame`. В результате получается объект Series, для которого индексом являются столбцы `frame`.

Если передать методу `apply` аргумент `axis='columns'`, то функция будет вызываться по одному разу для каждой строки:

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah   0.998382
Ohio   2.521511
Texas  0.676115
Oregon 2.542656
dtype: float64
```

Многие из наиболее распространенных статистик массивов (например, `sum` и `mean`) – методы DataFrame, поэтому применять `apply` в этом случае необязательно.

Функция, передаваемая методу `apply`, не обязана возвращать скалярное значение, она может вернуть и объект Series, содержащий несколько значений:

```
In [196]: def f(x):
    .....: return Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
Out[197]:
             b         d         e
min   -0.555730  0.281746 -1.296221
max    1.246435  1.965781  1.393406
```

Можно использовать и поэлементные функции Python. Допустим, требуется вычислить форматированную строку для каждого элемента `frame` с плавающей точкой. Это позволяет сделать метод `applymap`:

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
           b     d      e
Utah   -0.20  0.48  -0.52
Ohio   -0.56  1.97   1.39
Texas   0.09  0.28   0.77
Oregon  1.25  1.01  -1.30
```

Этот метод называется `applymap`, потому что в классе Series есть метод `map` для применения функции к каждому элементу:

```
In [200]: frame['e'].map(format)
Out[200]:
Utah     -0.52
Ohio      1.39
Texas     0.77
Oregon   -1.30
Name: e, dtype: object
```

Сортировка и ранжирование

Сортировка набора данных по некоторому критерию – еще одна важная встроенная операция. Для лексикографической сортировки по индексу служит метод `sort_index`, который возвращает новый отсортированный объект:

```
In [201]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a   1
b   2
c   3
d   0
dtype: int64
```

В случае DataFrame можно сортировать по индексу, ассоциированному с любой осью:

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
Out[204]:
      d a b c
one   4 5 6 7
three 0 1 2 3

In [205]: frame.sort_index(axis=1)
```

```
Out[205]:
      a b c d
three 1 2 3 0
one   5 6 7 4
```

По умолчанию данные сортируются в порядке возрастания, но можно отсортировать их и в порядке убывания:

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

Для сортировки Series по значениям служит метод `sort_values`:

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2  -3
3   2
0   4
1   7
dtype: int64
```

Отсутствующие значения по умолчанию оказываются в конце Series:

```
In [209]: obj = Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4   -3.0
5    2.0
0    4.0
2    7.0
1   NaN
3   NaN
dtype: float64
```

Объект DataFrame можно сортировать по значениям в одном или нескольких столбцах. Для этого имена столбцов следует передать в качестве значения параметра by метода `sort_values`:

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [212]: frame
Out[212]:
  a  b
0 0  4
1 1  7
2 0 -3
3 1  2
```

```
In [213]: frame.sort_values(by='b')
Out[213]:
  a  b
2 0 -3
3 1  2
0 0  4
1 1  7
```

Для сортировки по нескольким столбцам следует передать список имен:

```
In [214]: frame.sort_index(by=['a', 'b'])
Out[214]:
   a   b
2  0  -3
0  0   4
3  1   2
1  1   7
```

Ранжирование заключается в присваивании рангов – от единицы до числа присутствующих в массиве элементов. Для ранжирования применяется метод `rank` объектов Series и DataFrame; по умолчанию `rank` обрабатывает связанные ранги, присваивая каждой группе средний ранг:

```
In [215]: obj = Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ранги можно также присваивать в соответствии с порядком появления в данных:

```
In [217]: obj.rank(method='first')
Out[217]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Здесь элементам 0 и 2 присвоен не средний ранг 6.5, а ранг 6 и 7, потому что в данных метка 0 предшествует метке 2.

Можно ранжировать и в порядке убывания:

```
# В случае связанных рангов выбрать максимальный ранг в группе
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0   2.0
1   7.0
2   2.0
3   4.0
4   5.0
5   6.0
6   4.0
dtype: float64
```

В табл. 5.6 перечислены способы обработки связанных рангов.

Таблица 5.6. Способы обработки связанных рангов

Способ	Описание
`'average'`	По умолчанию: одинаковым значениям присвоить средний ранг
`'min'`	Всем элементам группы присвоить минимальный ранг
`'max'`	Всем элементам группы присвоить максимальный ранг
`'first'`	Присваивать ранги в порядке появления значений в наборе данных
`'dense'`	Как `method='min'`, но при переходе к следующей группе элементов с одинаковым рангом ранг всегда увеличивается на 1, а не на количество элементов в группе

DataFrame умеет вычислять ранги как по строкам, так и по столбцам:

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})

In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5

In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

Индексы по осям с повторяющимися значениями

Во всех рассмотренных до сих пор примерах метки на осях (значения индекса) были уникальны. Хотя для многих функций pandas (например, `reindex`)

требуется уникальность меток, в общем случае это необязательно. Рассмотрим небольшой объект Series с повторяющимися значениями в индексе:

```
In [222]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
a   0
a   1
b   2
b   3
c   4
```

О том, являются значения уникальными или нет, можно узнать, опросив свойство `is_unique`:

```
In [224]: obj.index.is_unique
Out[224]: False
```

Выборка данных – одна из основных операций, поведение которых меняется в зависимости от наличия или отсутствия дубликатов. При доступе по индексу, встречающемуся несколько раз, возвращается объект Series, если же индекс одиночный, то скалярное значение:

```
In [225]: obj['a']
Out[225]:
a   0
a   1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

Это иногда усложняет код, поскольку тип, получающийся в результате индексирования, может зависеть от того, повторяется метка или нет.

Такое же правило действует и для доступа к строкам в DataFrame:

```
In [227]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [228]: df
Out[228]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [229]: df.ix['b']
Out[229]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

5.3. Редукция и вычисление описательных статистик

Объекты pandas оснащены набором стандартных математических и статистических методов. Большая их часть попадает в категорию *редукций*, или *сводных статистик*, – методов, которые вычисляют единственное значение (например, сумму или среднее) для Series или объект Series – для строк либо столбцов DataFrame. По сравнению с эквивалентными методами массивов NumPy все они игнорируют отсутствующие значения. Рассмотрим небольшой объект DataFrame:

```
In [230]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                 [np.nan, np.nan], [0.75, -1.3]],
   .....:                index=['a', 'b', 'c', 'd'],
   .....:                columns=['one', 'two'])

In [231]: df
Out[231]:
     one    two
a   1.40    NaN
b   7.10   -4.5
c    NaN    NaN
d   0.75   -1.3
```

Метод `sum` объекта DataFrame возвращает Series, содержащий суммы по столбцам:

```
In [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

Если передать параметр `axis=1`, то суммирование будет производиться по строкам:

```
In [233]: df.sum(axis=1)
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
dtype: float64
```

Отсутствующие значения исключаются, если только не отсутствует весь срез (в данном случае строка или столбец). Это можно подавить, задав параметр `skipna`:

```
In [234]: df.mean(axis=1, skipna=False)
```

```
Out[234]:
a      NaN
b    1.300
c      NaN
d   -0.275
```

Перечень часто употребляемых параметров методов редукции приведен в табл. 5.9.

Таблица 5.9. Параметры методов редукции

Метод	Описание
`axis`	Ось, по которой производится редуцирование. В случае DataFrame 0 означает строки, 1 – столбцы
`skipna`	Исключать отсутствующие значения. По умолчанию `True`
`level`	Редуцировать с группировкой по уровням, если индекс по оси иерархический (MultiIndex)

Некоторые методы, например `idxmin` и `idxmax`, возвращают косвенные статистики, скажем, индекс, при котором достигается минимум или максимум:

```
In [235]: df.idxmax()
Out[235]:
one   b
two   d
dtype: object
```

Есть также *аккумулирующие* методы:

```
In [236]: df.cumsum()
Out[236]:
     one   two
a   1.40   NaN
b   8.50  -4.5
c    NaN   NaN
d   9.25  -5.8
```

Наконец, существуют методы, не относящиеся ни к редуцирующим, ни к аккумулирующим. Примером может служить метод `describe`, который возвращает несколько сводных статистик за одно обращение:

```
In [237]: df.describe()
Out[237]:
             one         two
count   3.000000    2.000000
mean    3.083333   -2.900000
std     3.493685    2.262742
min     0.750000   -4.500000
25%     1.075000   -3.700000
50%     1.400000   -2.900000
75%     4.250000   -2.100000
max     7.100000   -1.300000
```

В случае нечисловых данных describe возвращает другие сводные статистики:

```
In [238]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count     16
unique     3
top        a
freq       8
dtype: object
```

Полный список сводных статистик и родственных методов приведен в табл. 5.8.

Таблица 5.8. Описательные и сводные статистики

Метод	Описание
count	Количество значений, исключая отсутствующие
describe	Вычисляет набор сводных статистик для Series или для каждого столбца DataFrame
min, max	Вычисляет минимальное или максимальное значение
argmin, argmax	Вычисляет позицию в индексе (целые числа), при котором достигается минимальное или максимальное значение соответственно
idxmin, idxmax	Вычисляет значение индекса, при котором достигается минимальное или максимальное значение соответственно
quantile	Вычисляет выборочный квантиль в диапазоне от 0 до 1
sum	Сумма значений
mean	Среднее значение
median	Медиана (50%-ный квантиль)
mad	Среднее абсолютное отклонение от среднего
var	Выборочная дисперсия
std	Выборочное стандартное отклонение
skew	Асимметрия (третий момент)
kurt	Куртозис (четвертый момент)
cumsum	Нарастающая сумма
cummin, cummax	Нарастающий минимум или максимум соответственно
cumprod	Нарастающее произведение
diff	Первая арифметическая разность (полезно для временных рядов)
pct_change	Вычисляет процентное изменение

Корреляция и ковариация

Некоторые сводные статистики, например корреляция и ковариация, вычисляются по парам аргументов. Рассмотрим объекты DataFrame, содержащие цены акций и объемы биржевых сделок, взятые с сайта Yahoo! Finance. Для анализа применим дополнительный пакет pandas-datareader. Если он еще не установлен, установите его с помощью программ conda или pip:

```
conda install pandas-datareader
```

Я воспользуюсь модулем pandas_datareader, чтобы скачать данные о нескольких ценных бумагах:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
            for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
            for ticker, data in all_data.items()})
```

Возможно, когда вы будете читать этот текст, служба Yahoo! Finance уже прекратит существование, поскольку компания Yahoo! была приобретена Verizon в 2017 году. Обратитесь к онлайновой документации по пакету pandas-datareader для получения сведений об актуальной функциональности.

Теперь вычислим процентные изменения цен:

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                AAPL        GOOG         IBM        MSFT
Date
2016-10-17  -0.000680    0.001837    0.002072   -0.003483
2016-10-18  -0.000681    0.019616   -0.026168    0.007690
2016-10-19  -0.002979    0.007846    0.003583   -0.002255
2016-10-20  -0.000512   -0.005652    0.001719   -0.004867
2016-10-21  -0.003930    0.003011   -0.012474    0.042096
```

Метод corr объекта Series вычисляет корреляцию перекрывающихся, отличных от NA, выровненных по индексу значений в двух объектах Series. Соответственно, метод cov вычисляет ковариацию:

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

Поскольку MSFT – допустимое имя атрибута Python, те же столбцы можно выбрать более лаконично:

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

С другой стороны, методы corr и cov объекта DataFrame возвращают соответственно полную корреляционную или ковариационную матрицу в виде DataFrame:

```
In [247]: returns.corr()
```

```
Out[247]:
          AAPL     GOOG      IBM     MSFT
AAPL 1.000000 0.407919 0.386817 0.389695
GOOG 0.407919 1.000000 0.405099 0.465919
IBM  0.386817 0.405099 1.000000 0.499764
MSFT 0.389695 0.465919 0.499764 1.000000
```

```
In [248]: returns.cov()
Out[248]:
          AAPL     GOOG     IBM      MSFT
AAPL 0.000277 0.000107 0.000078 0.000095
GOOG 0.000107 0.000251 0.000078 0.000108
IBM  0.000078 0.000078 0.000146 0.000089
MSFT 0.000095 0.000108 0.000089 0.000215
```

С помощью метода `corrwith` объекта DataFrame можно вычислить попарные корреляции между столбцами или строками DataFrame и другим объектом Series или DataFrame. Если передать ему объект Series, то будет возвращен Series, содержащий значения корреляции, вычисленные для каждого столбца:

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL  0.386817
GOOG  0.405099
IBM   1.000000
MSFT  0.499764
dtype: float64
```

Если передать объект DataFrame, то будут вычислены корреляции столбцов с соответственными именами. Ниже я вычисляю корреляции процентных изменений с объемом сделок:

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL  -0.075565
GOOG  -0.007067
IBM   -0.204849
MSFT  -0.092950
dtype: float64
```

Если передать `axis='columns'`, то будут вычислены корреляции строк. Во всех случаях перед началом вычислений данные выравниваются по меткам.

Уникальные значения, счетчики значений и членство

Еще один класс методов служит для извлечения информации о значениях, хранящихся в одномерном объекте Series. Для иллюстрации рассмотрим пример:

```
In [251]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

Метод unique возвращает массив уникальных значений в Series:

```
In [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: array([c, a, d, b], dtype=object)
```

Уникальные значения необязательно возвращаются в отсортированном порядке, но могут быть отсортированы впоследствии, если это необходимо (uniques.sort()). Метод value_counts вычисляет объект Series, содержащий частоты встречаемости значений:

```
In [254]: obj.value_counts()
Out[254]:
c   3
a   3
b   2
d   1
dtype: int64
```

Для удобства этот объект отсортирован по значениям в порядке убывания. Функция value_counts может быть также вызвана как метод pandas верхнего уровня. В таком случае она применима к любому массиву или последовательности:

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a   3
b   2
c   3
d   1
dtype: int64
```

Метод isin вычисляет булев вектор членства в множестве и может быть полезен для фильтрации набора данных относительно подмножества значений в объекте Series или столбце DataFrame:

```
In [256]: obj
Out[256]:
0  c
1  a
2  d
3  a
4  a
5  b
6  b
7  c
8  c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])
```

```
In [258]: mask
Out[258]:
0   True
1  False
2  False
3  False
4  False
5   True
6   True
7   True
8   True
dtype: bool
```

```
In [259]: obj[mask]
Out[259]:
0  c
5  b
6  b
7  c
8  c
dtype: object
```

С isin тесно связан метод Index.get_indexer, который возвращает массив индексов, описывающий сопоставление между массивом потенциально повторяющихся значений и массивом, содержащим только различные значения:

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

Справочная информация по этим методам приведена в табл. 5.9.

Таблица 5.9. Уникальные значения, счетчики значений и методы «раскладывания»

Метод	Описание
isin	Вычисляет булев массив, показывающий, содержится ли каждое принадлежащее Series значение в переданной последовательности
get_indexer	Вычисляет для каждого значения массива целочисленный индекс в другом массиве неповторяющихся значений; полезен, когда нужно выполнить выравнивание данных и операции, подобные соединению
unique	Вычисляет массив уникальных значений в Series и возвращает их в порядке появления
value_counts	Возвращает объект Series, который содержит уникальное значение в качестве индекса и его частоту в качестве соответствующего значения. Результат отсортирован в порядке убывания частот

Иногда требуется вычислить гистограмму нескольких взаимосвязанных столбцов в DataFrame. Приведем пример:

```
In [263]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
    .....:                  'Qu2': [2, 3, 1, 2, 3],
    .....:                  'Qu3': [1, 5, 2, 4, 4]})
```

```
In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Передача pandas.value_counts методу apply этого объекта DataFrame дает:

```
In [265]: result = data.apply(pd.value_counts).fillna(0)
```

```
In [266]: result
Out[266]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

Здесь метками строк результирующего объекта являются неповторяющиеся значения, встречающиеся в столбцах. Значениями же являются счетчики значений в столбце.

5.4. Заключение

В следующей главе мы обсудим имеющиеся в pandas средства чтения (или *загрузки*) и записи наборов данных. Затем мы займемся вопросами очистки, трансформации, анализа данных и визуализации данных.

Глава 6. Чтение и запись данных, форматы файлов

Доступ к данным – обязательный первый шаг применения большинства описанных в книге инструментов. Я буду рассматривать в основном ввод и вывод с помощью объектов pandas, хотя, разумеется, и в других библиотеках нет недостатка в соответствующих средствах.

Обычно средства ввода-вывода относят к нескольким категориям: чтение файлов в текстовом или каком-то более эффективном двоичном формате, загрузка из баз данных и взаимодействие с сетевыми источниками, например API доступа к вебу.

6.1. Чтение и запись данных в текстовом формате

В библиотеке pandas имеется ряд функций для чтения табличных данных, представленных в виде объекта DataFrame. Все они перечислены в табл. 6.1, хотя чаще всего вы будете иметь дело с функциями `read_csv` и `read_table`.

Я дам краткий обзор функций, которые служат для преобразования текстовых данных в объект DataFrame. Их необязательные параметры можно отнести к нескольким категориям.

- *Индексирование*: какие столбцы рассматривать как индекс возвращаемого DataFrame и откуда брать имена столбцов: из файла, от пользователя или вообще ниоткуда.
- *Выведение типа и преобразование данных*: включает определенные пользователем преобразования значений и список маркеров отсутствующих данных.
- *Разбор даты и времени*: включает средства комбинирования, в том числе сбор данных о дате и времени из нескольких исходных столбцов в один результирующий.

- *Итерирование*: поддержка обхода очень больших файлов.
- *Проблемы грязных данных*: пропуск заголовка или концевика, комментариев и другие мелочи, например обработка числовых данных, в которых тройки разрядов разделены запятыми.

Таблица 6.1. Функции чтения в pandas

Функция	Описание
`read_csv`	Загружает данные с разделителями из файла, URL-адреса или похожего на файл объекта. По умолчанию разделителем является запятая
`read_table`	Загружает данные с разделителями из файла, URL-адреса или похожего на файл объекта. По умолчанию разделителем является символ табуляции (`'\t'`)
`read_fwf`	Читает данные в формате с фиксированной шириной столбцов (без разделителей)
`read_clipboard`	Вариант `read_table`, который читает данные из буфера обмена. Полезно для преобразования в таблицу данных на веб-странице
`read_excel`	Читает табличные данные из файлов Excel в формате XLS или XLSX
`read_hdf`	Читает HDF5-файлы, записанные pandas
`read_html`	Читает все таблицы, обнаруженные в HTML-документе
`read_json`	Читает данные из строки в формате JSON (JavaScript Object Notation)
`read_msgpack`	Читает данные pandas, представленные в двоичном формате MessagePack
`read_pickle`	Читает произвольный объект, хранящийся в формате Python pickle
`read_sas`	Читает набор данных в одном из пользовательских форматов хранения системы SAS
`read_sql`	Читает результаты SQL-запроса (применяя пакет SQLAlchemy) в объект pandas DataFrame
`read_stata`	Читает набор данных из файла в формате Stata
`read_feather`	Читает данные из двоичного файла в формате Feather

В связи с тем что в реальности данные могут быть очень грязными, некоторые функции загрузки (в особенности `read_csv`) со временем обросли гигантским количеством опций (так, у `read_csv` их уже больше 50). В онлайновой документации есть много примеров их применения. Не исключено, что вы найдете подходящий, сражаясь с конкретным файлом.

Некоторые функции, в частности `pandas.read_csv`, производят *выведение типа*, поскольку информация о типах данных в столбцах не входит в формат данных. Это означает, что необязательно явно указывать, какие столбцы являются числовыми, целыми, булевыми или строками. Есть и такие форматы, например HDF5, Feather и msgpack, в которых сведения о типах данных хранятся явно.

Для обработки дат и нестандартных типов требуется больше усилий. Начнем с текстового файла, содержащего короткий список данных через запятую (формат CSV):

```
In [8]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Здесь я воспользовался командой Unix cat, которая печатает содержимое файла на экране без какого-либо форматирования. Если вы работаете в Windows, можете с тем же успехом использовать команду type.

Поскольку данные разделены запятыми, мы можем прочитать их в DataFrame с помощью функции read_csv:

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Можно было бы также воспользоваться функцией read_table, указав разделитель:

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

В файле не всегда есть строка-заголовок. Рассмотрим такой файл:

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Прочитать его можно двумя способами. Можно поручить pandas выбрать имена столбцов по умолчанию, а можно задать их самостоятельно:

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
     0    1    2    3      4
0    1    2    3    4  hello
1    5    6    7    8  world
2    9   10   11   12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Допустим, мы хотим, чтобы столбец message стал индексом возвращаемого объекта DataFrame. Этого можно добиться, задав аргумент index_col, в котором указать, что индексом будет столбец с номером 4 или с именем 'message':

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
        a   b   c   d
message
hello   1   2   3   4
world   5   6   7   8
foo     9  10  11  12
```

Если вы хотите сформировать иерархический индекс из нескольких столбцов, то просто передайте список их номеров или имен:

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....:                      index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
          value1  value2
key1 key2
one  a         1       2
     b         3       4
     c         5       6
     d         7       8
two  a         9      10
     b        11      12
     c        13      14
     d        15      16
```

Иногда в таблице нет фиксированного разделителя, а для разделения полей используются пробелы или еще какой-то символ. В таком случае можно передать функции `read_table` регулярное выражение вместо разделителя. Рассмотрим такой текстовый файл:

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n'
```

В данном случае поля разделены переменным числом пробелов и, хотя можно было бы переформатировать данные вручную, проще передать функции read_table регулярное выражение \s+ в качестве разделителя:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Поскольку имен столбцов на одно меньше, чем строк, read_table делает вывод, что в данном частном случае первый столбец должен быть индексом DataFrame.

У функций разбора много дополнительных аргументов, которые помогают справиться с широким разнообразием файловых форматов (см. табл. 6.2). Например, параметр skiprows позволяет пропустить первую, третью и четвертую строки файла:

```
In [23]: !cat ch06/ex4.csv
# привет!
a,b,c,d,message
# хотелось немного усложнить тебе жизнь,
# а нечего читать CSV-файлы на компьютере
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Обработка отсутствующих значений – важная и зачастую сопровождаемая тонкими нюансами часть разбора файла. Отсутствующие значения обычно либо вообще опущены (пустые строки), либо представлены специальными *маркерами*. По умолчанию в pandas используется набор общеупотребительных маркеров: NA, -1.#IND и NULL:

```
In [25]: !cat ch06/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo

In [26]: result = pd.read_csv('examples/ex5.csv')
```

```
In [27]: result
Out[27]:
  something   a   b    c   d   message
0       one   1   2  3.0   4       NaN
1       two   5   6  NaN   8     world
2     three   9  10  1.0  12       foo
```

```
In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

Параметр na_values может принимать список или множество строк, рассматриваемых как маркеры отсутствующих значений:

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])
```

```
In [30]: result
Out[30]:
   something   a   b    c   d   message
0        one   1   2  3.0   4       NaN
1        two   5   6  NaN   8     world
2      three   9  10 11.0  12       foo
```

Если в разных столбцах применяются разные маркеры, то их можно задать с помощью словаря:

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}
```

```
In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
   something   a   b    c   d   message
0        one   1   2  3.0   4       NaN
1        NaN   5   6  NaN   8     world
2      three   9  10 11.0  12       NaN
```

В табл. 6.2 перечислены некоторые часто используемые аргументы функций pandas.read_csv и pandas.read_table.

Таблица 6.2. Аргументы функций read_csv и read_table

Аргумент	Описание
path	Строка, обозначающая путь в файловой системе, URL-адрес или похожий на файл объект
sep или delimiter	Последовательность символов или регулярное выражение, служащее для разделения полей в строке
header	Номер строки, содержащей имена столбцов. По умолчанию равен 0 (первая строка). Если строки-заголовка нет, должен быть равен None

Таблица 6.2 (окончание)

Аргумент	Описание
index_col	Номера или имена столбцов, трактуемых как индекс строк в результирующем объекте. Может быть задан один номер (имя) или список номеров (имен), определяющий иерархический индекс
names	Список имен столбцов результирующего объекта; задается, если header=None
skiprows	Количество игнорируемых начальных строк или список номеров игнорируемых строк (нумерация начинается с 0)
na_values	Последовательность значений, интерпретируемых как маркеры отсутствующих данных
comment	Один или несколько символов, начинающих комментарий, который продолжается до конца строки
parse_dates	Пытаться разобрать данные как дату и время; по умолчанию False. Если равен True, то производится попытка разобрать все столбцы. Можно также задать список столбцов, которые следует объединить перед разбором (если, например, время и даты заданы в разных столбцах)
keep_date_col	В случае когда для разбора данных столбцы объединяются, следует ли отбрасывать объединенные столбцы. По умолчанию True
converters	Словарь, содержащий отображение номеров или имен столбцов на функции. Например, {'foo': f} означает, что нужно применить функцию f ко всем значениям в столбце foo
dayfirst	При разборе потенциально неоднозначных дат предполагать международный формат (т. е. 7/6/2012 означает «7 июня 2012 года»). По умолчанию False
date_parser	Функция, применяемая для разбора дат
nrows	Количество читаемых строк от начала файла
iterator	Возвращает объект TextParser для чтения файла порциями
chunksize	Размер порции при итерировании
skip_footer	Сколько строк в конце файла игнорировать
verbose	Печатать разного рода информацию о ходе разбора, например количество отсутствующих значений, помещенных в нечисловые столбцы
encoding	Кодировка текста в случае Unicode. Например, 'utf-8' означает, что текст представлен в кодировке UTF-8
squeeze	Если в результате разбора данных оказалось, что имеется только один столбец, вернуть объект Series
thousands	Разделитель тысяч, например ',' или '.'

Чтение текстовых файлов порциями

Для обработки очень больших файлов или для того, чтобы определить правильный набор аргументов, необходимых для обработки большого файла, иногда требуется прочитать небольшой фрагмент файла или последовательно читать файл небольшими порциями.

Прежде чем приступать к обработке большого файла, попросим pandas отображать меньше данных:

```
In [33]: pd.options.display.max_rows = 10
```

Теперь имеем:

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
       one           two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
... ... ... ... ... ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

Чтобы прочитать только небольшое число строк (а не весь файл), нужно задать это число в параметре nrows:

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

Для чтения файла порциями задайте с помощью параметра chunksize размер порции в строках:

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>>
```

Объект TextParser, возвращаемый функцией read_csv, позволяет читать файл порциями размера chunksize. Например, можно таким образом итеративно читать файл ex6.csv, агрегируя счетчики значений в столбце 'key':

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.order(ascending=False)
```

Имеем:

```
In [40]: tot[:10]
Out[40]:
E   368.0
X   364.0
L   346.0
O   343.0
Q   340.0
M   338.0
J   337.0
F   335.0
K   334.0
H   330.0
dtype: float64
```

У объекта TextParser имеется также метод get_chunk, который позволяет читать куски произвольного размера.

Вывод данных в текстовом формате

Данные можно экспортировать в формате с разделителями. Рассмотрим одну из приведенных выше операций чтения CSV-файла:

```
In [41]: data = pd.read_csv('ch06/ex5.csv')
```

```
In [42]: data
Out[42]:
   something   a   b    c   d   message
0        one   1   2  3.0   4       NaN
1        two   5   6  NaN   8     world
2      three   9  10 11.0  12       foo
```

С помощью метода to_csv объекта DataFrame мы можем вывести данные в файл через запятую:

```
In [43]: data.to_csv('examples/out.csv')
```

```
In [44]: !cat ch06/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Конечно, допустимы и другие разделители (при выводе в sys.stdout результат отправляется на стандартный вывод, обычно на экран):

```
In [45]: import sys
```

```
In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
```

```
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Отсутствующие значения представлены пустыми строками, но можно вместо этого указать какой-нибудь маркер:

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

Если не указано противное, выводятся метки строк и столбцов. Но и те, и другие можно подавить:

```
In [48]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

Можно также вывести лишь подмножество столбцов, задав их порядок:

```
In [49]: data.to_csv(sys.stdout, index=False, cols=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

У объекта Series также имеется метод to_csv:

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01 00:00:00,0
2000-01-02 00:00:00,1
2000-01-03 00:00:00,2
2000-01-04 00:00:00,3
2000-01-05 00:00:00,4
2000-01-06 00:00:00,5
2000-01-07 00:00:00,6
```

Обработка данных в формате с разделителями

Как правило, табличные данные можно загрузить с диска с помощью функции pandas.read_table и родственных ей. Но иногда требуется ручная обработка. Не так уж необычно встретить файл, в котором одна или несколько строк сформированы неправильно, что сбивает read_table. Для иллюстрации базовых средств рассмотрим небольшой CSV-файл:

```
In [891]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3","4"
```

Для любого файла с односимвольным разделителем можно воспользоваться стандартным модулем Python csv. Для этого передайте открытый файл или объект, похожий на файл, методу csv.reader:

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

Итерирование файла с помощью объекта reader дает кортежи значений в каждой строке после удаления кавычек:

```
In [56]: for line in reader:
   .....:     print line
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3', '4']
```

Далее можно произвести любые манипуляции, необходимые для преобразования данных к нужному виду. Пойдем по шагам. Сначала прочитаем файл в список строк:

```
In [57]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

Затем отделим строку-заголовок от остальных:

```
In [58]: header, values = lines[0], lines[1:]
```

Далее мы можем создать словарь столбцов, применив словарное включение и выражение zip(*values), которое транспонирует строки и столбцы:

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [60]: data_dict
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

Встречаются различные вариации CSV-файлов. Для установления нового формата со своим разделителем, соглашением об употреблении кавычек и способе завершения строк необходимо определить простой подкласс класса csv.Dialect:

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

Параметры диалекта CSV можно задать также в виде именованных параметров `csv.reader`, не определяя подкласса:

```
reader = csv.reader(f, delimiter='|')
```

Возможные атрибуты `csv.Dialect` вместе с назначением каждого описаны в табл. 6.3.

Таблица 6.3. Параметры диалекта CSV

Аргумент	Описание
`delimiter`	Односимвольная строка, определяющая разделитель полей. По умолчанию `','`
`lineterminator`	Завершитель строк при выводе, по умолчанию `'\r\n'`. Объект reader игнорирует этот параметр, используя вместо него платформенное соглашение о концах строк
`quotechar`	Символ закавычивания для полей, содержащих специальные символы (например, разделитель). По умолчанию `'"'`
`quoting`	Соглашение об употреблении кавычек. Допустимые значения: `csv.QUOTE_ALL` (заключать в кавычки все поля), `csv.QUOTE_MINIMAL` (только поля, содержащие специальные символы, например разделитель), `csv.QUOTE_NONNUMERIC` и `csv.QUOTE_NON` (не заключать в кавычки). Полное описание см. в документации. По умолчанию `QUOTE_MINIMAL`
`skipinitialspace`	Игнорировать пробелы после каждого разделителя. По умолчанию `False`
`doublequote`	Как обрабатывать символ кавычки внутри поля. Если `True`, добавляется второй символ кавычки. Полное описание поведения см. в документации
`escapechar`	Строка для экранирования разделителя в случае, когда `quoting` равно `csv.QUOTE_NONE`. По умолчанию экранирование выключено

Если в файле употребляются более сложные или фиксированные многосимвольные разделители, то воспользоваться модулем `csv` не удастся. В таких случаях придется разбивать строку на части и производить другие действия по очистке данных, применяя метод строки `split` или метод регулярного выражения `re.split`.

Для *записи* файлов с разделителями вручную можно использовать метод `csv.writer`. Он принимает объект, который представляет открытый, допускающий запись файл и те же параметры диалекта и форматирования, что `csv.reader`:

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

Данные в формате JSON

Формат JSON (JavaScript Object Notation) стал очень популярен для обмена данными по протоколу HTTP между веб-сервером и браузером или другим клиентским приложением. Этот формат обладает куда большей гибкостью, чем табличный текстовый формат типа CSV. Например:

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

Данные в формате JSON очень напоминают код на Python с тем отличием, что отсутствующее значение обозначается null. Есть еще некоторые нюансы (например, запрещается ставить запятую после последнего элемента списка). Базовыми типами являются объекты (словари), массивы (списки), строки, числа, булевы значения и null. Ключ любого объекта должен быть строкой. На Python существует несколько библиотек для чтения и записи JSON-данных. Здесь я воспользуюсь модулем json, потому что он входит в стандартную библиотеку Python. Для преобразования JSON-строки в объект Python служит метод json.loads:

```
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

Напротив, метод json.dumps преобразует объект Python в формат JSON:

```
In [65]: asjson = json.dumps(result)
```

Как именно преобразовывать объект JSON или список таких объектов в DataFrame или еще какую-то структуру данных для анализа, решать вам. Для удобства предлагается возможность передать список словарей (которые раньше были объектами JSON) конструктору DataFrame и выбрать подмножество полей данных:

```
In [66]: siblings = DataFrame(result['siblings'], columns=['name', 'age'])

In [67]: siblings
Out[67]:
   name   age
0  Scott   25
1  Katie   33
```

Функция pandas.read_json умеет автоматически преобразовывать наборы данных определенного вида в формате JSON в объекты Series или DataFrame. Например:

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

Подразумеваемые по умолчанию параметры `pandas.read_json` предполагают, что каждый объект в JSON-массиве – это строка таблицы:

```
In [69]: data = pd.read_json('examples/example.json')
```

```
In [70]: data
Out[70]:
  a b c
0 1 2 3
1 4 5 6
2 7 8 9
```


Более полный пример чтения и манипулирования данными в формате JSON (включая вложенные записи) приведен при рассмотрении базы данных о продуктах питания USDA в главе 7.

XML и HTML: разбор веб-страниц

На Python написано много библиотек для чтения и записи данных в вездесущих форматах HTML и XML, в частности библиотеки lxml (http://lxml.de), Beautiful Soup и html5lib. Хотя lxml в общем случае работает гораздо быстрее остальных, другие библиотеки лучше справляются с неправильно сформированными HTML- и XML-файлами.

В pandas имеется функция `read_html`, которая использует внешние библиотеки типа lxml или Beautiful Soup для автоматического выделения таблиц из HTML-файлов и представления их в виде объектов DataFrame. Чтобы продемонстрировать, как это работает, я скачал из Федеральной корпорации страхования вкладов США HTML-файл (он упоминается в документации по pandas) со сведениями о банкротствах банков[1]. Сначала необходимо установить дополнительные библиотеки, необходимые функции `read_html`:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

Если вы не пользуетесь conda, то можно с тем же успехом выполнить команду `pip install lxml`.

У функции `pandas.read_html` имеется много параметров, но по умолчанию она ищет и пытается разобрать все табличные данные внутри тегов `<table>`. Результатом является список объектов DataFrame:

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')
```

```
In [74]: len(tables)
```


[1] Полный список см. на сайте https://www.fdic.gov/bank/individual/failed/banklist.html.

```
Out[74]: 1

In [75]: failures = tables[0]
In [76]: failures.head()
Out[76]:
                     Bank Name          City    ST  CERT \
0                  Allied Bank         Mulberry AR    91
1 The Woodbury Banking Company         Woodbury GA 11297
2       First CornerStone Bank  King of Prussia PA 35312
3           Trust Company Bank          Memphis TN  9956
4   North Milwaukee State Bank        Milwaukee WI 20364
                Acquiring Institution  Closing Date            Updated Date
0                         Today's Bank  September 23, 2016  November 17, 2016
1                          United Bank     August 19, 2016  November 17, 2016
2 First-Citizens Bank & Trust Company         May 6, 2016  September 6, 2016
3          The Bank of Fayette County      April 29, 2016  September 6, 2016
4 First-Citizens Bank & Trust Company      March 11, 2016      June 16, 2016
```

Поскольку в объекте `failures` много столбцов, pandas вставляет знак разрыва строки \.

Как будет показано в следующих главах, дальше мы можем произвести очистку и анализ данных, например посчитать количество банкротств по годам:

```
In [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
2010  157
2009  140
2011   92
2012   51
2008   25
...
2004    4
2001    4
2007    3
2003    3
2000    2
Name: Closing Date, Length: 15, dtype: int64
```

Разбор XML с помощью lxml.objectify

XML (расширяемый язык разметки) – еще один популярный формат представления структурированных данных, поддерживающий иерархически вложенные данные, снабженные метаданными. Текст этой книги на самом деле представляет собой набор больших XML-документов.

Выше я продемонстрировал работу функции `pandas.read_html`, которая пользуется библиотекой lxml или Beautiful Soup для разбора HTML-файлов. Форматы XML и HTML структурно похожи, но XML более общий. Ниже я покажу, как с помощью lxml разбирать данные в формате XML.

Управление городского транспорта Нью-Йорка (MTA) публикует временные ряды с данными о работе автобусов и электричек (http://www.mta.info/developers/download.html). Мы сейчас рассмотрим данные о качестве обслуживания, хранящиеся в виде XML-файлов. Для каждой автобусной и железнодорожной компании существует свой файл (например, `Performance_MNR.xml` для компании MetroNorth Railroad), содержащий данные за месяц в виде последовательности таких XML-записей:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
  systemwide. The availability rate is based on physical observations performed
  the morning of regular business days only. This is a new indicator the agency
  began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Используя `lxml.objectify`, мы разбираем файл и получаем ссылку на корневой узел XML-документа от метода `getroot`:

```
from lxml import objectify

path = 'datasets/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

Свойство `root.INDICATOR` возвращает генератор, последовательно отдающий все элементы `<INDICATOR>`. Для каждой записи мы заполняем словарь имен тегов (например, `YTD_ACTUAL`) значениями данных (некоторые теги пропускаются):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
```

```
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Наконец, преобразуем этот список словарей в объект DataFrame:

```
In [81]: perf = pd.DataFrame(data)

In [82]: perf.head()
Out[82]:
Empty DataFrame
Columns: []
Index: []
```

XML-документы могут быть гораздо сложнее, чем в этом примере. В частности, в каждом элементе могут быть метаданные. Рассмотрим тег гиперссылки в формате HTML, который является частным случаем XML:

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'

root = objectify.parse(StringIO(tag)).getroot()
```

Теперь мы можем обратиться к любому атрибуту тега (например, `href`) или к тексту ссылки:


```
In [84]: root
Out[84]: <Element a at 0x88bd4b0>

In [85]: root.get('href')
Out[85]: 'http://www.google.com'

In [86]: root.text
Out[86]: 'Google'
```

6.2. Двоичные форматы данных

Один из самых простых способов эффективно хранить данные в двоичном формате – воспользоваться встроенным в Python методом сериализации `pickle`. Поэтому у всех объектов pandas есть метод `save`, которых сохраняет данные на диске в виде pickle-файла:

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a   b   c   d  message
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo

In [89]: frame.save('examples/frame_pickle')
```

Прочитать данные с диска позволяет метод `pandas.load`, также упрощающий интерфейс с pickle:

```
In [90]: pd.load('examples/frame_pickle')
Out[90]:
   a   b   c   d  message
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo
```

`pickle` рекомендуется использовать только для краткосрочного хранения. Проблема в том, что невозможно гарантировать неизменность формата: сегодня вы сериализовали объект в формате pickle, а следующая версия библиотеки не сможет его десериализовать. Я приложил все усилия к тому, чтобы в pandas такое не случалось, но, возможно, наступит момент, когда придется «поломать» формат pickle.

В pandas встроена поддержка еще двух двоичных форматов данных: HDF5 и MessagePack. В следующем разделе я приведу несколько примеров работы с HDF5, но призываю вас самостоятельно исследовать другие форматы, чтобы оценить их эффективность и пригодность для анализа в вашей задаче. Из форматов хранения данных, поддерживаемых pandas или NumPy, упомяну следующие:

- bcolz – допускающий сжатие двоичный формат хранения по столбцам, основанный на библиотеке сжатия Blosc;
- Feather – кросс-языковой формат хранения по столбцам, который мы спроектировали вместе с Хэдли Уикхэмом (Hadley Wickham) из сообщества языка R. В Feather используется формат столбцовой организации памяти Apache Arrow.

Формат HDF5

HDF5 – хорошо зарекомендовавший себя файловый формат для хранения больших объемов научных данных в виде массивов. Для работы с ним используется библиотека, написанная на C и имеющая интерфейсы ко многим языкам, в том числе Java, Julia, Python и MATLAB. Акроним «HDF» в ее названии означает *hierarchical data format* (иерархический формат данных). Каждый HDF5-файл содержит внутри себя структуру узлов, напоминающую файловую систему, которая позволяет хранить несколько наборов данных вместе с относящимися к ним метаданными. В отличие от более простых форматов, HDF5 поддерживает сжатие на лету с помощью различных алгоритмов сжатия, что позволяет более эффективно хранить повторяющиеся комбинации данных. Для очень больших наборов данных, которые не помещаются в память, HDF5 – отличный выбор, потому что дает возможность эффективно читать и записывать небольшие участки гораздо больших массивов.

К библиотеке HDF5 существует целых два интерфейса из Python: PyTables и h5py. Но pandas предоставляет высокоуровневый интерфейс, который упро-

щает хранение объектов Series и DataFrame. Класс HDFStore работает как словарь и отвечает за низкоуровневые детали:

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1                frame       (shape->[100,1])

/obj1_col            series      (shape->[100])

/obj2                frame_table (typ->appendable,nrows->100,ncols->1,indexers->
[index])

/obj3                frame_table (typ->appendable,nrows->100,ncols->1,indexers->
[index])
```

Объекты из HDF5-файла можно извлекать как из словаря:

```
In [97]: store['obj1']
Out[97]:
          a
0  -0.204708
1   0.478943
2  -0.519439
3  -0.555730
4   1.965781
.. ...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

HDFStore поддерживает две схемы хранения: 'fixed' и 'table'. Последняя, вообще говоря, медленнее, но поддерживает запросы в специальном синтаксисе:

```
In [98]: store.put('obj2', frame, format='table')

In [99]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[99]:
          a
10  1.007189
11 -1.296221
```

```
12  0.274992
13  0.228913
14  1.352917
15  0.886429
In [100]: store.close()
```

Метод `put` – явный вариант синтаксиса `store['obj2'] = frame`, он позволяет дополнительно задавать другие параметры, например формат хранения.

Функция `pandas.read_hdf` дает лаконичный способ доступа к этой функциональности:

```
In [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')

In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[102]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

Если вы работаете с данными, которые хранятся на удаленных серверах, например Amazon S3 или HDFS, то может оказаться более подходящим какой-нибудь другой двоичный формат, разработанный специально для распределенных хранилищ, например Apache Parquet (http://parquet.apache.org/). Средства доступа к Parquet и другим подобным форматам хранения из Python пока еще разрабатываются, поэтому я не стану писать о них в этой книге.

Если вы собираетесь работать с очень большими объемами данных, то я рекомендую изучить PyTables и h5py и посмотреть, в какой мере они отвечают вашим потребностям. Поскольку многие задачи анализа данных ограничены прежде всего скоростью ввода-вывода (а не быстродействием процессора), использование средства типа HDF5 способно существенно ускорить работу приложения.

HDF5 *не является* базой данных. Лучше всего она приспособлена для работы с наборами данных, которые записываются один раз, а читаются многократно. Данные можно добавлять в файл в любой момент, но если это делает одновременно несколько клиентов, то файл можно повредить.

Чтение файлов Microsoft Excel

В pandas имеется также поддержка для чтения табличных данных в формате Excel 2003 (и более поздних версий) с помощью класса `ExcelFile`. На внутреннем уровне `ExcelFile` пользуется пакетами `xlrd` и `openpyxl` для чтения файлов в формате XLS и XLSX соответственно.

Для работы с классом `ExcelFile` создайте его экземпляр, передав конструктору путь к файлу с расширением `xls` или `xlsx`:

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

Прочитать данные из рабочего листа в объект DataFrame позволяет метод parse:

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   a  b  c  d  message
0  1  2  3  4    hello
1  5  6  7  8    world
2  9 10 11 12      foo
```


Если вы собираетесь читать несколько рабочих листов из файла, то быстрее создать объект ExcelFile, но можно просто передать имя файла функции pandas.read_excel:

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
   a  b  c  d  message
0  1  2  3  4    hello
1  5  6  7  8    world
2  9 10 11 12      foo
```

Для записи данных pandas в файл формата Excel следует сначала создать объект ExcelWriter, а затем записать в него данные, пользуясь методом to_excel объектов pandas:

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [109]: frame.to_excel(writer, 'Sheet1')

In [110]: writer.save()
```

Можно вместо этого передать путь к файлу методу to_excel, избежав тем самым создания ExcelWriter:


```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

6.3. Взаимодействие с HTML и Web API

Многие сайты предоставляют открытый API для получения данных в формате JSON или каком-то другом. Получить доступ к таким API из Python можно разными способами; я рекомендую простой пакет requests (http://docs.python-requests.org).

Чтобы найти последние 30 заявок, касающихся pandas на GitHub, мы можем отправить с помощью библиотеки requests такой HTTP-запрос GET:

```
In [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'
```

```
In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

Метод `json` объекта Response возвращает словарь, содержащий JSON-данные, представленные в виде объектов Python:

```
In [117]: data = resp.json()

In [118]: data[0]['title']
Out[118]: 'Period does not round down for frequencies less that 1 hour'
```

Каждый элемент в списке `data` – словарь, содержащий все данные на странице заявки в GitHub (кроме комментариев). Список `data` можно передать конструктору DataFrame и выделить интересующие нас поля:

```
In [119]: issues = pd.DataFrame(data, columns=['number', 'title',
   .....:                                      'labels', 'state'])

In [120]: issues
Out[120]:
   number                                              title \
0   17666  Period does not round down for frequencies les...
1   17665           DOC: improve docstring of function where
2   17664               COMPAT: skip 32-bit test on int repr
3   17662                          implement Delegator class
4   17654  BUG: Fix series rename called with str alterin...
..    ...                                                ...
25  17603  BUG: Correctly localize naive datetime strings...
26  17599                     core.dtypes.generic --> cython
27  17596   Merge cdate_range functionality into bdate_range
28  17587  Time Grouper bug fix when applied for list gro...
29  17583  BUG: fix tz-aware DatetimeIndex + TimedeltaInd...

                                               labels state
0                                                  []  open
1   [{'id': 134699, 'url': 'https://api.github.com...  open
2   [{'id': 563047854, 'url': 'https://api.github....  open
3                                                  []  open
4   [{'id': 76811, 'url': 'https://api.github.com/...  open
..                                                ...   ...
25  [{'id': 76811, 'url': 'https://api.github.com/...  open
26  [{'id': 49094459, 'url': 'https://api.github.c...  open
27  [{'id': 35818298, 'url': 'https://api.github.c...  open
28  [{'id': 233160, 'url': 'https://api.github.com...  open
29  [{'id': 76811, 'url': 'https://api.github.com/...  open
[30 rows x 4 columns]
```

Немного попотев, вы сможете создать высокоуровневые интерфейсы к распространенным API работы с вебом, которые возвращают объекты DataFrame для дальнейшего анализа.

6.4. Взаимодействие с базами данных

В корпоративных системах большая часть данных хранится не в текстовых или Excel-файлах. Широко используются реляционные базы данных на основе SQL (например, SQL Server, PostgreSQL и MySQL), а равно альтернативные базы данных, быстро набирающие популярность. Выбор базы данных обычно диктуется производительностью, необходимостью поддержания целостности данных и потребностями приложения в масштабируемости.

Загрузка данных из реляционной базы в DataFrame производится довольно прямолинейно, и в pandas есть несколько функций для упрощения этой процедуры. В качестве примера возьму базу данных SQLite, целиком размещающуюся в памяти, и драйвер `sqlite3`, включенный в стандартную библиотеку Python:

```
In [121]: import sqlite3

In [122]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL, d INTEGER
   .....: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>

In [125]: con.commit()
```

Затем вставлю несколько строк в таблицу:

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....:         ('Tallahassee', 'Florida', 2.6, 3),
   .....:         ('Sacramento', 'California', 1.7, 5)]

In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [128]: con.executemany(stmt, data)
Out[128]: <sqlite3.Cursor at 0x7f6b15c66ce0>

In [129]: con.commit()
```

Большинство драйверов SQL, имеющихся в Python (PyODBC, psycopg2, MySQLdb, pymssql и т. д.), при выборе данных из таблицы возвращают список кортежей:

```
In [130]: cursor = con.execute('select * from test')

In [131]: rows = cursor.fetchall()

In [132]: rows
Out[132]:
[(u'Atlanta', u'Georgia', 1.25, 6),
```

```
 (u'Tallahassee', u'Florida', 2.6, 3),
 (u'Sacramento', u'California', 1.7, 5)]
```

Этот список кортежей можно передать конструктору DataFrame, но необходимы еще имена столбцов, содержащиеся в атрибуте курсора `description`:

```
In [133]: cursor.description
Out[133]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [134]: DataFrame(rows, columns=zip(*cursor.description)[0])
Out[134]:
             a           b      c   d
0      Atlanta     Georgia   1.25   6
1  Tallahassee     Florida   2.60   3
2   Sacramento  California   1.70   5
```

Такое переформатирование не хочется выполнять при каждом запросе к базе данных. Проект SQLAlchemy (www.sqlalchemy.org) – популярная библиотека на Python, абстрагирующая многие различия между базами данных SQL. В pandas имеется функция `read_sql`, которая позволяет без труда читать данные из соединения, открытого SQLAlchemy. В примере ниже мы подключаемся к той же самой базе SQLite с помощью SQLAlchemy и читаем данные из ранее созданной таблицы:

```
In [135]: import sqlalchemy as sqla

In [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [137]: pd.read_sql('select * from test', db)
Out[137]:
            a           b     c  d
0     Atlanta     Georgia  1.25  6
1 Tallahassee     Florida  2.60  3
2  Sacramento  California  1.70  5
```

6.5. Заключение

Получение доступа к данным часто является первым шагом процесса анализа данных. В этой главе мы рассмотрели ряд полезных средств, позволяющих приступить к делу. В следующих главах более детально опишем переформатирование данных, визуализацию, анализ временных рядов и другие вопросы.

Глава 7. Очистка и подготовка данных

Значительная часть времени программиста, занимающегося анализом и моделированием данных, уходит на подготовку данных: загрузку, очистку, преобразование и реорганизацию. Часто говорят, что это составляет 80 % и даже более времени работы аналитика. Иногда способ хранения данных в файлах или в базе не согласуется с алгоритмом обработки. Многие предпочитают писать преобразования данных из одной формы в другую на каком-нибудь универсальном языке программирования типа Python, Perl, R или Java либо с помощью имеющихся в UNIX средств обработки текста типа sed или awk. По счастью, pandas дополняет стандартную библиотеку Python высокоуровневыми, гибкими и производительными базовыми преобразованиями и алгоритмами, которые позволяют переформатировать данные без особых проблем.

Если вы наткнетесь на манипуляцию, которой нет ни в этой книге, ни вообще в библиотеке pandas, не стесняйтесь внести предложение в списке рассылки или на сайте GitHub. Вообще, многое в pandas – в части как проектирования, так и реализации – обусловлено потребностями реальных приложений.

В этой главе мы обсудим средства работы с отсутствующими и повторяющимися данными, средства обработки строк и некоторые другие преобразования данных, применяемые в процессе анализа. А следующая глава будет посвящена различным способам комбинирования и реорганизации наборов данных.

7.1. Обработка отсутствующих данных

Отсутствующие данные – типичное явление в большинстве аналитических приложений. При проектировании pandas в качестве одной из целей ставилась задача сделать работу с отсутствующими данными как можно менее

болезненной. Например, при вычислении всех описательных статистик для объектов pandas отсутствующие данные не учитываются.

Способ представления отсутствующих данных в объектах pandas не идеален, но большинство пользователей он устраивает. В pandas для представления отсутствующих данных с плавающей точкой используется значение NaN (не число). Это просто *признак*, который легко распознать:

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0   aardvark
1  artichoke
2        NaN
3    avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

В pandas мы приняли соглашение, заимствованное из языка программирования R, – обозначать отсутствующие данные NA – *not available* (недоступны). В статистических приложениях NA может означать, что данные не существуют или существуют, но не наблюдаемы (например, из-за сложностей сбора данных). В процессе очистки данных зачастую важно анализировать сами отсутствующие данные, чтобы выявить проблемы, относящиеся к их сбору, или потенциальное смещение, вызванное отсутствием данных.

Встроенное в Python значение None также рассматривается как отсутствующее в массивах объектов:

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0    True
1   False
2    True
3   False
dtype: bool
```

В проекте pandas ведутся работы по совершенствованию внутренних деталей обработки отсутствующих данных, но функции пользовательского API, в частности pandas.isnull, абстрагируют многие досаждающие детали. В табл. 7.1 приведен перечень некоторых функций, относящихся к обработке отсутствующих данных.

Таблица 7.1. Методы обработки отсутствующих данных

Метод	Описание
`dropna`	Фильтрует метки оси в зависимости от того, существуют ли для метки отсутствующие данные, причем есть возможность указать различные пороги, определяющие, какое количество отсутствующих данных считать допустимым
`fillna`	Восполняет отсутствующие данные указанным значением или использует какой-нибудь метод интерполяции, например `'ffill'` или `'bfill'`
`isnull`	Возвращает объект, содержащий булевы значения, которые показывают, какие значения отсутствуют
`notnull`	Логическое отрицание `isnull`

Фильтрация отсутствующих данных

Существует несколько способов фильтрации отсутствующих данных. Конечно, можно сделать это и вручную с помощью функции `pandas.isnull` и булева индексирования, но часто бывает полезен метод `dropna`. Для Series он возвращает другой объект Series, содержащий только данные и значения индекса, отличные от NA:

```
In [15]: from numpy import nan as NA

In [16]: data = Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0   1.0
2   3.5
4   7.0
dtype: float64
```

Это эквивалентно такому коду:

```
In [18]: data[data.notnull()]
Out[18]:
0   1.0
2   3.5
4   7.0
dtype: float64
```


В случае объектов DataFrame все немного сложнее. Можно отбрасывать строки или столбцы, если они содержат только NA-значения или хотя бы одно NA-значение. По умолчанию метод `dropna` отбрасывает все строки, содержащие хотя бы одно отсутствующее значение:

```
In [19]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
   .....:                 [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
```

```
Out[21]:
     0    1    2
0    1  6.5    3
1    1  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5    3

In [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

Если передать параметр `how='all'`, то будут отброшены строки, которые целиком состоят из отсутствующих значений:

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
0    1  6.5    3
1    1  NaN  NaN
3  NaN  6.5    3
```

Для отбрасывания столбцов достаточно передать параметр `axis=1`:

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     0    1    2    4
0  1.0  6.5  3.0  NaN
1  1.0  NaN  NaN  NaN
2  NaN  NaN  NaN  NaN
3  NaN  6.5  3.0  NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

Родственный способ фильтрации строк DataFrame в основном применяется к временным рядам. Допустим, требуется оставить только строки, содержащие определенное количество наблюдений. Этот порог можно задать с помощью аргумента `thresh`:

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA
```

```
In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Восполнение отсутствующих данных

Иногда отсутствующие данные желательно не отфильтровывать (и потенциально вместе с ними отбрасывать полезные данные), а каким-то способом заполнить дыры. В большинстве случаев для этой цели можно использовать метод `fillna`. Ему передается константа, подставляемая вместо отсутствующих значений:

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Если передать методу `fillna` словарь, то можно будет подставлять вместо отсутствующих данных значение, зависящее от столбца:

```
In [34]: df.fillna({1: 0.5, 2: 0})
```

```
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Метод `fillna` возвращает новый объект, но можно также модифицировать существующий объект на месте:

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Те же методы интерполяции, что применяются для переиндексации, годятся и для `fillna`:

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0        1         2
0  0.476985 3.248944 -1.021228
1 -0.577087 0.124121  0.302614
2  0.523772      NaN  1.343810
3 -0.713544      NaN -2.370232
4 -1.860761      NaN       NaN
5 -1.265934      NaN       NaN

In [41]: df.fillna(method='ffill')
Out[41]:
          0        1         2
0  0.476985 3.248944 -1.021228
1 -0.577087 0.124121  0.302614
2  0.523772 0.124121  1.343810
3 -0.713544 0.124121 -2.370232
4 -1.860761 0.124121 -2.370232
```

```
5 -1.265934 0.124121 -2.370232
```

```
In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0        1         2
0  0.476985 3.248944 -1.021228
1 -0.577087 0.124121  0.302614
2  0.523772 0.124121  1.343810
3 -0.713544 0.124121 -2.370232
4 -1.860761      NaN -2.370232
5 -1.265934      NaN -2.370232
```

При некоторой изобретательности можно использовать `fillna` и другими способами, например передать среднее или медиану объекта Series:

```
In [43]: data = Series([1., NA, 3.5, NA, 7])
```

```
In [44]: data.fillna(data.mean())
Out[44]:
0   1.000000
1   3.833333
2   3.500000
3   3.833333
4   7.000000
dtype: float64
```

Справочная информация о методе `fillna` приведена в табл. 7.2.

Таблица 7.2. Аргументы метода `fillna`

Аргумент	Описание
`value`	Скалярное значение или похожий на словарь объект для восполнения отсутствующих значений
`method`	Метод интерполяции. По умолчанию, если не задано других аргументов, предполагается метод `'ffill'`
`axis`	Ось, по которой производится восполнение. По умолчанию `axis=0`
`inplace`	Модифицировать исходный объект, не создавая копию
`limit`	Для прямого и обратного восполнения максимальное количество непрерывных заполняемых промежутков

7.2. Преобразование данных

До сих пор мы в этой главе занимались реорганизацией данных. Фильтрация, очистка и прочие преобразования составляют еще один, не менее важный класс операций.

Устранение дубликатов

Строки-дубликаты могут появиться в объекте DataFrame по разным причинам. Приведем пример:

```
In [45]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
   .....:                  'k2': [1, 1, 2, 3, 3, 4, 4]})
In [46]: data
Out[46]:
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4
```

Метод `duplicated` объекта DataFrame возвращает булев объект Series, который для каждой строки показывает, есть в ней дубликаты или нет:

```
In [47]: data.duplicated()
Out[47]:
0   False
1    True
2   False
3   False
4    True
5   False
6    True
```

А метод `drop_duplicates` возвращает DataFrame, для которого массив, возвращенный методом `duplicated`, будет содержать только значения `False`:

```
In [48]: data.drop_duplicates()
Out[48]:
   k1 k2
0 one  1
1 two  1
2 one  2
3 two  3
4 one  3
5 two  4
```

По умолчанию оба метода принимают во внимание все столбцы, но можно указать произвольное подмножество столбцов, которые необходимо исследовать на наличие дубликатов. Допустим, есть еще один столбец значений, и мы хотим отфильтровать строки, которые содержат повторяющиеся значения в столбце `'k1'`:

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
   k1 k2 v1
0 one  1  0
1 two  1  1
```

По умолчанию методы duplicated и drop_duplicates оставляют первую встретившуюся строку с данной комбинацией значений. Но если задать параметр keep='last', то будет оставлена последняя строка:

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
   k1 k2 v1
0 one  1  0
1 two  1  1
2 one  2  2
3 two  3  3
4 one  3  4
6 two  4  6
```

Преобразование данных с помощью функции или отображения

Часто бывает необходимо произвести преобразование набора данных исходя из значений в некотором массиве, объекте Series или столбце объекта DataFrame. Рассмотрим гипотетические данные о сортах мяса:

```
In [52]: data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',
   .....:                   'corned beef', 'Bacon', 'pastrami', 'honey ham',
   .....:                   'nova lox'],
   .....:                   'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

Допустим, требуется добавить столбец, в котором указано соответствующее сорту мяса животное. Создадим отображение сортов мяса на виды животных:

```
meat_to_animal = {
  'bacon'      : 'pig',
  'pulled pork': 'pig',
  'pastrami':    'cow',
  'corned beef': 'cow',
  'honey ham':   'pig',
  'nova lox':    'salmon'
}
```

Метод map объекта Series принимает функцию или похожий на словарь объект, содержащий отображение, но в данном случае возникает мелкая проблема: у нас названия некоторых сортов мяса начинаются с заглавной буквы, наименования других – со строчной. Поэтому нужно привести все строки к нижнему регистру методом str.lower объекта Series:

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0         bacon
1   pulled pork
2         bacon
3      pastrami
4   corned beef
5         bacon
6      pastrami
7     honey ham
8      nova lox

Name: food, dtype: object

In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
          food ounces animal
0        bacon    4.0    pig
1  pulled pork    3.0    pig
2        bacon   12.0    pig
3     Pastrami    6.0    cow
4  corned beef    7.5    cow
5        Bacon    8.0    pig
6     pastrami    3.0    cow
7    honey ham    5.0    pig
8     nova lox    6.0 salmon
```

Можно было бы также передать функцию, выполняющую всю эту работу:

```
In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
0      pig
1      pig
2      pig
3      cow
4      cow
5      pig
6      cow
7      pig
8   salmon
Name: food, dtype: object
```

Метод map – удобное средство выполнения поэлементных преобразований и других операций очистки.

Замена значений

Восполнение отсутствующих данных методом fillna можно рассматривать как частный случай более общей замены значений. Если метод map, как мы только что видели, позволяет модифицировать подмножество значений, хранящихся в объекте, то метод replace предлагает для этого более простой и гибкий интерфейс. Рассмотрим такой объект Series:

```
In [60]: data = Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
0        1.0
1     -999.0
2        2.0
3     -999.0
4    -1000.0
5        3.0
dtype: float64
```

Значение -999 могло бы быть маркером отсутствующих данных. Чтобы заменить все такие значения теми, которые понимает pandas, воспользуемся методом replace, порождающим новый объект Series (если только не передан аргумент inplace=True):


```
In [62]: data.replace(-999, np.nan)
Out[62]:
0      1
1    NaN
2      2
3    NaN
4  -1000
5      3
dtype: float64
```

Чтобы заменить сразу несколько значений, нужно передать их список и заменяющее значение:

```
In [63]: data.replace([-999, -1000], np.nan)
Out[63]:
0      1
1    NaN
2      2
3    NaN
4    NaN
5      3
dtype: float64
```

Если для каждого заменяемого значения нужно свое заменяющее, передаем список замен:

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
dtype: float64
```

В аргументе можно передавать также словарь:

```
In [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
dtype: float64
```

Метод `data.replace` не то же самое, что метод `data.str.replace`, который выполняет поэлементную замену строки. Методы работы со строками будут рассмотрены при обсуждении объекта Series ниже в этой главе.

Переименование индексов осей

Как и значения в объекте Series, метки осей можно преобразовывать с помощью функции или отображения, порождающего новые объекты с другими метками. Оси можно также модифицировать на месте, не создавая новую структуру данных. Вот простой пример:

```
In [66]: data = DataFrame(np.arange(12).reshape((3, 4)),
  .....:                  index=['Ohio', 'Colorado', 'New York'],
  .....:                  columns=['one', 'two', 'three', 'four'])
```

Как и у объекта Series, у индексов осей имеется метод `map`:

```
In [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(str.upper)
Out[68]: array([OHIO, COLO, NEW ], dtype=object)
```

Индексу можно присваивать значение, т. е. модифицировать DataFrame на месте:

```
In [69]: data.index = data.index.map(transform)
```

```
In [70]: data
Out[70]:
     one two three four
OHIO   0   1     2    3
COLO   4   5     6    7
NEW    8   9    10   11
```

Если требуется создать преобразованный вариант набора данных, не меняя оригинал, то будет полезен метод `rename`:

```
In [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
     ONE TWO THREE FOUR
Ohio   0   1     2    3
Colo   4   5     6    7
New    8   9    10   11
```

Интересно, что `rename` можно использовать в сочетании с похожим на словарь объектом, который возвращает новые значения для подмножества меток оси:

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
  .....: columns={'three': 'peekaboo'})
Out[72]:
          one  two  peekaboo  four
INDIANA     0    1         2     3
COLO        4    5         6     7
NEW         8    9        10    11
```

Метод `rename` избавляет от необходимости копировать объект DataFrame вручную и присваивать значения его атрибутам `index` и `columns`. Чтобы модифицировать набор данных на месте, задайте параметр `inplace=True`:

```
In [73]: _ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [74]: data
Out[74]:
          one  two  three  four
INDIANA     0    1      2     3
COLO        4    5      6     7
NEW         8    9     10    11
```

Дискретизация и раскладывание

Непрерывные данные часто дискретизируются или как-то иначе раскладываются по интервалам – ящикам – для анализа. Предположим, имеются данные о группе лиц в каком-то исследовании, и требуется разложить их по ящикам, соответствующим возрасту – дискретной величине:

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Разобьем эти ящики на группы: от 18 до 25, от 26 до 35, от 35 до 60 и наконец от 61. Для этой цели в pandas есть функция cut:

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)

In [78]: cats
Out[78]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

pandas возвращает специальный объект Categorical. Показанный выше результат – ящики, вычисленные методом pandas.cut. Его можно рассматривать как массив строк с именами ящиков; на самом деле он содержит массив categories, в котором хранятся неповторяющиеся имена категорий, а также метки данных ages в атрибуте codes:

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8))

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')

In [81]: pd.value_counts(cats)
Out[81]:
(18, 25]    5
(35, 60]    3
(25, 35]    3
(60, 100]   1
dtype: int64
```

Заметим, что pd.value_counts(cats) – счетчики ящиков, вычисленных pandas.cut.

Согласно принятой в математике нотации интервалов, круглая скобка означает, что соответствующий конец не включается (*открыт*), а квадратная – что включается (*замкнут*). Чтобы сделать открытым правый конец, следует задать параметр right=False:

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36,
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

Можно также самостоятельно задать имена ящиков, передав список или массив в параметре labels:

```
In [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Mid
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

Если передать методу cut целое число ящиков, а не явно заданные границы, то он разобьет данные на группы равной длины исходя из минимального и максимального значений. Рассмотрим раскладывание равномерно распределенных данных по четырем ящикам:

```
In [85]: data = np.random.rand(20)

In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

Параметр precision=2 означает, что числа следует выводить с двумя десятичными знаками после точки.

Родственная функция qcut раскладывает данные исходя из выборочных квантилей. Метод cut обычно создает ящики, содержащие разное число точек, – это всецело устанавливается распределением данных. Но поскольку qcut пользуется выборочными квантилями, то по определению получаются ящики равного размера:

```
In [87]: data = np.random.randn(1000) # нормальное распределение

In [88]: cats = pd.qcut(data, 4) # разложить по квартилям

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68,
-0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265,
0.62] < (0.62, 3.928]]

In [90]: pd.value_counts(cats)
Out[90]:
(0.62, 3.928]    250
```

```
(-0.0265, 0.62]  250
(-0.68, -0.0265] 250
(-2.95, -0.68]   250
dtype: int64
```

Как и в случае cut, можно задать величины квантилей (числа от 0 до 1 включительно) самостоятельно:

```
In [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.0265,
1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.95, -1.187], (-0.0265, 1.286],
(-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, -0.0265] < (-0.0265,
1.286] < (1.286, 3.928]]
```

Мы еще вернемся к методам cut и qcut позже в этой главе, когда будем обсуждать агрегирование и групповые операции, поскольку эти функции дискретизации особенно полезны для анализа квантилей и групп.

Обнаружение и фильтрация выбросов

Фильтрация или преобразование выбросов – это в основном вопрос применения операций с массивами. Рассмотрим объект DataFrame с нормально распределенными данными:

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
                0           1            2            3
count 1000.000000 1000.000000  1000.000000  1000.000000
mean     0.049091    0.026112    -0.002544    -0.051827
std      0.996947    1.007458     0.995232     0.998311
min     -3.645860   -3.184377    -3.745356    -3.428254
25%     -0.599807   -0.612162    -0.687373    -0.747478
50%      0.047101   -0.013609    -0.022158    -0.088274
75%      0.756646    0.695298     0.699046     0.623331
max      2.653656    3.525865     2.735527     3.366626
```

Допустим, мы хотим найти в одном из столбцов значения, превышающие 3 по абсолютной величине:

```
In [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
41  -3.399312
```

```
136 -3.745356
Name: 2, dtype: float64
```

Чтобы выбрать все строки, в которых встречаются значения, по абсолютной величине превышающие 3, мы можем воспользоваться методом any для булева объекта DataFrame:

```
In [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
            0         1         2         3
41   0.457246 -0.025907 -3.399312 -0.974657
60   1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

Можно также присваивать значения данным, удовлетворяющим этому критерию. Следующий код срезает значения, выходящие за границы интервала от –3 до 3:

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3
```

```
In [98]: data.describe()
Out[98]:
0 1 2 3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.050286     0.025567    -0.001399    -0.051765
std       0.992920     1.004214     0.991414     0.995761
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
Max       2.653656     3.000000     2.735527     3.000000
```

Выражение np.sign(data) равно 1 или –1 в зависимости от того, является значение data положительным или отрицательным:

```
In [99]: np.sign(data).head()
Out[99]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

Перестановки и случайная выборка

Переставить (случайным образом переупорядочить) объект Series или строки объекта DataFrame легко с помощью функции `numpy.random.permutation`. Если передать функции `permutation` длину оси, для которой производится перестановка, то будет возвращен массив целых чисел, описывающий новый порядок:

```
In [100]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

Этот массив затем можно использовать для индексирования на основе `iloc` или, что эквивалентно, передать функции `take`:

```
In [103]: df
Out[103]:
0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [104]: df.take(sampler)
Out[104]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

Чтобы выбрать случайное подмножество без возвращения, можно использовать метод `sample` объектов Series и DataFrame:

```
In [105]: df.sample(n=3)
Out[105]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

Чтобы сгенерировать выборку *с возвращением* (когда разрешается выбирать один и тот же элемент несколько раз), передайте методу `sample` аргумент `replace=True`:

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])

In [107]: draws = choices.sample(n=10, replace=True)
```

```
In [108]: draws
Out[108]:
4   4
1   7
4   4
2  -1
0   5
3   6
1   7
4   4
0   5
4   4
dtype: int64
```

Вычисление индикаторных переменных

Еще одно преобразование, часто встречающееся в статистическом моделировании и машинном обучении, – преобразование категориальной переменной в фиктивную, или индикаторную, матрицу. Если в столбце объекта DataFrame встречается k различных значений, то можно построить матрицу или объект DataFrame с k столбцами, содержащими только нули и единицы. В библиотеке pandas для этого имеется функция `get_dummies`, хотя нетрудно написать и свою собственную. Вернемся к приведенному выше примеру DataFrame:

```
In [109]: df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                 'data1': range(6)})

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

Иногда желательно добавить префикс к столбцам индикаторного объекта DataFrame, который затем можно будет слить с другими данными. У функции `get_dummies` для этой цели предусмотрен аргумент `prefix`:

```
In [111]: dummies = pd.get_dummies(df['key'], prefix='key')

In [112]: df_with_dummy = df[['data1']].join(dummies)

In [113]: df_with_dummy
Out[113]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
```

```
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

Если некоторая строка DataFrame принадлежит нескольким категориям, то ситуация немного усложняется. Рассмотрим набор данных MovieLens 1M, который будет более подробно исследован в главе 14:

```
In [114]: mnames = ['movie_id', 'title', 'genres']

In [115]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
   .....:                        header=None, names=mnames)

In [116]: movies[:10]
Out[116]:
   movie_id                               title                        genres
0         1                   Toy Story (1995)   Animation|Children's|Comedy
1         2                     Jumanji (1995)  Adventure|Children's|Fantasy
2         3            Grumpier Old Men (1995)                Comedy|Romance
3         4           Waiting to Exhale (1995)                  Comedy|Drama
4         5 Father of the Bride Part II (1995)                        Comedy
5         6                        Heat (1995)         Action|Crime|Thriller
6         7                      Sabrina (1995)                Comedy|Romance
7         8                Tom and Huck (1995)          Adventure|Children's
8         9                Sudden Death (1995)                        Action
9        10                   GoldenEye (1995)     Action|Adventure|Thriller
```

Чтобы добавить индикаторные переменные для каждого жанра, данные придется немного переформатировать. Сначала построим список уникальных жанров, встречающихся в наборе данных:

```
In [117]: all_genres = []

In [118]: for x in movies.genres:
   .....:     all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)
Теперь имеем:
In [120]: genres
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

Для построения индикаторного DataFrame можно, например, начать с объекта DataFrame, содержащего только нули:

```
In [121]: zero_matrix = np.zeros((len(movies), len(genres)))

In [122]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

Затем перебираем все фильмы и присваиваем элементам в каждой строке объекта dummies значение 1. Для этого воспользуемся атрибутом dummies.columns, чтобы вычислить индексы столбцов для каждого жанра:

```
In [123]: gen = movies.genres[0]

In [124]: gen.split('|')
Out[124]: ['Animation', "Children's", 'Comedy']

In [125]: dummies.columns.get_indexer(gen.split('|'))
Out[125]: array([0, 1, 2])
```

Далее можно использовать .iloc, чтобы установить значения для этих индексов:

```
In [126]: for i, gen in enumerate(movies.genres):
   .....:     indices = dummies.columns.get_indexer(gen.split('|'))
   .....:     dummies.iloc[i, indices] = 1
   .....:
```

После этого можно, как и раньше, соединить с movies:

```
In [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [128]: movies_windic.iloc[0]
Out[128]:
movie_id                                    1
title                           Toy Story (1995)
genres                 Animation|Children's|Comedy
Genre_Animation                             1
Genre_Children's                            1
Genre_Comedy                                1
Genre_Adventure                             0
Genre_Fantasy                               0
Genre_Romance                               0
Genre_Drama                                 0
                              ...
Genre_Crime                                 0
Genre_Thriller                              0
Genre_Horror                                0
Genre_Sci-Fi                                0
Genre_Documentary                           0
Genre_War                                   0
Genre_Musical                               0
Genre_Mystery                               0
Genre_Film-Noir                             0
Genre_Western                               0
Name: 0, Length: 21, dtype: object
```

Для очень больших наборов данных такой способ построения индикаторных переменных для нескольких категорий быстрым не назовешь. Было бы лучше реализовать низкоуровневую функцию, которая пишет напрямую в массив NumPy, а затем обернуть результат объектом DataFrame.

В статистических приложениях бывает полезно сочетать функцию get_dummies с той или иной функцией дискретизации, например cut:

```
In [129]: np.random.seed(12345)

In [204]: values = np.random.rand(10)

In [205]: values
Out[205]:
array([ 0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645,
        0.6532, 0.7489, 0.6536])

In [206]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [207]: pd.get_dummies(pd.cut(values, bins))
Out[207]:
   (0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1]
0         0           0           0           0         1
1         0           1           0           0         0
2         1           0           0           0         0
3         0           1           0           0         0
4         0           0           1           0         0
5         0           0           1           0         0
6         0           0           0           0         1
7         0           0           0           1         0
8         0           0           0           1         0
9         0           0           0           1         0
```

7.3. Манипуляции со строками

Python уже давно является популярным языком манипулирования данными отчасти потому, что располагает простыми средствами обработки строк и текста. В большинстве случаев оперировать текстом легко – благодаря наличию встроенных методов у строковых объектов. В более сложных ситуациях, когда нужно сопоставлять текст с образцами, на помощь приходят регулярные выражения. Библиотека pandas расширяет этот инструментарий, позволяя применять методы строк и регулярных выражений к целым массивам и беря на себя возню с отсутствующими значениями.

Методы строковых объектов

Для многих приложений вполне достаточно встроенных методов работы со строками. Например, строку, в которой данные записаны через запятую, можно разбить на поля с помощью метода split:

```
In [134]: val = 'a,b, guido'

In [135]: val.split(',')
Out[135]: ['a', 'b', ' guido']
```

Метод split часто употребляется вместе с методом strip, чтобы убрать пробельные символы (в том числе переход на новую строку):

```
In [136]: pieces = [x.strip() for x in val.split(',')]

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

Чтобы конкатенировать строки, применяя в качестве разделителя двойное двоеточие, можно использовать оператор сложения:

```
In [138]: first, second, third = pieces

In [139]: first + '::' + second + '::' + third
Out[139]: 'a::b::guido'
```

Но это недостаточно общий метод. Быстрее и лучше соответствует духу Python другой способ: передать список или кортеж методу join строки '::':

```
In [140]: '::'.join(pieces)
Out[140]: 'a::b::guido'
```

Существуют также методы для поиска подстрок. Лучше всего искать подстроку с помощью ключевого слова in, но методы index и find тоже годятся:

```
In [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1

In [143]: val.find(':')
Out[143]: -1
```

Разница между find и index состоит в том, что index возбуждает исключение, если строка не найдена (вместо того чтобы возвращать –1):

```
In [144]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-218-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
Метод count возвращает количество вхождений подстроки:
In [219]: val.count(',')
Out[219]: 2
```

Метод replace заменяет вхождения образца указанной строкой. Он же применяется для удаления подстрок – достаточно в качестве заменяющей передать пустую строку:

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b:: guido'
```

```
In [147]: val.replace(',', '')
Out[147]: 'ab guido'
```

В табл. 7.3 перечислены некоторые методы работы со строками в Python.

Как мы вскоре увидим, во многих таких операциях можно использовать также регулярные выражения.

Таблица 7.3. Встроенные в Python методы строковых объектов

Метод	Описание
`count`	Возвращает количество неперекрывающихся вхождений подстроки в строку
`endswith, startswith`	Возвращает `True`, если строка оканчивается (начинается) указанной подстрокой
`join`	Использовать данную строку как разделитель при конкатенации последовательности других строк
`index`	Возвращает позицию первого символа подстроки в строке. Если подстрока не найдена, возбуждает исключение `ValueError`
`find`	Возвращает позицию первого символа *первого* вхождения подстроки в строку, как и `index`. Но если строка не найдена, то возвращает –1
`rfind`	Возвращает позицию первого символа *последнего* вхождения подстроки в строку. Если строка не найдена, то возвращает –1
`replace`	Заменяет вхождения одной строки другой строкой
`strip, rstrip, lstrip`	Удаляет пробельные символы, в том числе символы новой строки в начале и (или) конце строки
`split`	Разбивает строку на список подстрок по указанному разделителю
`lower`	Преобразует буквы в нижний регистр
`upper`	Преобразует буквы в верхний регистр
`ljust, rjust`	Выравнивает строку по левой или правой границе соответственно. Противоположный конец строки заполняется пробелами (или каким-либо другим символом), так чтобы получилась строка как минимум заданной длины

Регулярные выражения

Регулярные выражения представляют собой простое средство сопоставления строки с образцом. Синтаксически это строка, записанная с соблюдением правил языка регулярных выражений. Стандартный модуль `re` содержит методы для применения регулярных выражений к строкам. Ниже приводятся примеры.

Искусству написания регулярных выражений можно было бы посвятить отдельную главу, но это выходит за рамки данной книги. В интернете и в других книгах имеется немало отличных пособий и справочных руководств.

Функции из модуля `re` можно отнести к трем категориям: сопоставление с образцом, замена и разбиение. Естественно, все они взаимосвязаны; регулярное выражение описывает образец, который нужно найти в тексте, а затем его уже можно применять для разных целей. Рассмотрим простой пример: требуется разбить строку в тех местах, где имеется сколько-то пробельных символов (пробелов, знаков табуляции и знаков новой строки). Для

сопоставления с одним или несколькими пробельными символами служит регулярное выражение \s+:

```
In [148]: import re

In [149]: text = "foo    bar\t baz  \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

При обращении re.split('\s+', text) сначала *компилируется* регулярное выражение, а затем его методу split передается заданный текст. Можно просто откомпилировать регулярное выражение методом re.compile, создав тем самым объект, допускающий повторное использование:

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

Чтобы получить список всех подстрок, отвечающих данному регулярному выражению, следует воспользоваться методом findall:

```
In [153]: regex.findall(text)
Out[153]: ['    ', '\t ', '  \t']
```

Чтобы не прибегать к громоздкому экранированию знаков \ в регулярном выражении, пользуйтесь *примитивными* (raw) строковыми литералами, например r'C:\x' вместо 'C:\\x'.

Создавать объект регулярного выражения с помощью метода re.compile рекомендуется, если вы планируете применять одно и то же выражение к нескольким строкам, при этом экономится процессорное время.

С findall тесно связаны методы match и search. Если findall возвращает все найденные в строке соответствия, то search – лишь первое. А метод match находит *только* соответствие, начинающееся в начале строки. В качестве не столь тривиального примера рассмотрим блок текста и регулярное выражение, распознающее большинство адресов электронной почты:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""

pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# Флаг re.IGNORECASE делает регулярное выражение нечувствительным к регистру
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Применение метода findall к этому тексту порождает список почтовых адресов:

```
In [155]: regex.findall(text)
```

```
Out[155]:
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com']
```

Метод search возвращает специальный объект соответствия для первого встретившегося в тексте адреса. В нашем случае этот объект может сказать только о начальной и конечной позициях найденного в строке образца:

```
In [156]: m = regex.search(text)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [158]: text[m.start():m.end()]
Out[159]: 'dave@google.com'
```

Метод regex.match возвращает None, потому что он находит соответствие образцу только в начале строки:

```
In [159]: print regex.match(text)
None
```

Метод sub возвращает новую строку, в которой вхождения образца заменены указанной строкой:

```
In [160]: print regex.sub('REDACTED', text)
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Предположим, что мы хотим найти почтовые адреса и в то же время разбить каждый адрес на три компонента: имя пользователя, имя домена и суффикс домена. Для этого заключим соответствующие части образца в скобки:

```
In [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [162]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Метод groups объекта соответствия, порожденного таким модифицированным регулярным выражением, возвращает кортеж компонентов образца:

```
In [163]: m = regex.match('wesm@bright.net')

In [164]: m.groups()
Out[164]: ('wesm', 'bright', 'net')
```

Если в образце есть группы, то метод findall возвращает список кортежей:

```
In [165]: regex.findall(text)
Out[165]:
[('dave', 'google', 'com'),
```

```
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

Метод sub тоже имеет доступ к группам в каждом найденном соответствии с помощью специальных конструкций \1, \2 и т. д.:

```
In [166]: print regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text)
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

О регулярных выражениях в Python можно рассказывать еще долго, но бóльшая часть этого материала выходит за рамки данной книги. В табл. 7.4 приведена краткая сводка методов.

Таблица 7.4. Методы регулярных выражений

Метод	Описание
findall	Возвращает список всех непересекающихся образцов, найденных в строке
finditer	Аналогичен findall, но возвращает итератор
match	Ищет соответствие образцу в начале строки и факультативно выделяет в образце группы. Если образец найден, возвращает объект соответствия, иначе None
search	Ищет в строке образец; если найден, возвращает объект соответствия. В отличие от match, образец может находиться в любом месте строки, а не только в начале
split	Разбивает строку на части в местах вхождения образца
sub, subn	Заменяет все (sub) или только первые n (subn) вхождений образца указанной строкой. Чтобы в указанной строке сослаться на группы, выделенные в образце, используйте конструкции \1, \2, ...

Векторные строковые функции в pandas

Очистка замусоренного набора данных для последующего анализа подразумевает значительный объем манипуляций со строками и использование регулярных выражений. А чтобы жизнь не казалась медом, в столбцах, содержащих строки, иногда встречаются отсутствующие значения:

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [168]: data = pd.Series(data)

In [169]: data
Out[169]:
Dave     dave@google.com
Rob        rob@gmail.com
Steve    steve@gmail.com
Wes                  NaN
dtype: object
```

```
In [170]: data.isnull()
Out[170]:
Dave     False
Rob      False
Steve    False
Wes       True
dtype: bool
```

Методы строк и регулярных выражений можно применить к каждому значению с помощью метода `data.map` (которому передается лямбда или другая функция), но для отсутствующих значений они «грохнутся». Чтобы справиться с этой проблемой, в классе Series есть методы для операций со строками, которые пропускают отсутствующие значения. Доступ к ним производится через атрибут `str`; например, вот как можно было бы с помощью метода `str.contains` проверить, содержит ли каждый почтовый адрес подстроку `'gmail'`:

```
In [171]: data.str.contains('gmail')
Out[171]:
Dave     False
Rob       True
Steve     True
Wes        NaN
dtype: object
```

Регулярные выражения тоже можно так использовать, равно как и их флаги типа `IGNORECASE`:

```
In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
Dave      [('dave', 'google', 'com')]
Rob         [('rob', 'gmail', 'com')]
Steve     [('steve', 'gmail', 'com')]
Wes                               NaN
dtype: object
```

Существует два способа векторной выборки элементов: `str.get` или доступ к атрибуту `str` по индексу:

```
In [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave    True
Rob     True
Steve   True
Wes      NaN
dtype: object
```

Для доступа к элементам списков мы можем передать индекс любой из этих функций:

```
In [176]: matches.str.get(1)
Out[176]:
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64

In [177]: matches.str[0]
Out[177]:
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64
```


Аналогичный синтаксис позволяет вырезать строки:

```
In [178]: data.str[:5]
Out[178]:
Dave       dave@
Rob        rob@g
Steve      steve
Wes          NaN
```


В табл. 7.5 перечислены дополнительные методы строк в pandas.

Таблица 7.5. Неполный перечень векторных методов строковых объектов

Метод	Описание
`cat`	Поэлементно конкатенирует строки с необязательным разделителем
`contains`	Возвращает булев массив, показывающий, содержит ли каждая строка указанный образец
`count`	Подсчитывает количество вхождений образца
`extract`	Использует регулярное выражение с группами, чтобы выделить одну или несколько строк из объекта Series, содержащего строки; результатом является DataFrame, содержащий по одному столбцу на каждую группу
`endswith`	Эквивалентно `x.endswith(pattern)` для каждого элемента
`startswith`	Эквивалентно `x.startswith(pattern)` для каждого элемента
`findall`	Возвращает список всех вхождений образца для каждой строки
`get`	Доступ по индексу ко всем элементам (выбрать `i`-й элемент)
`isalnum`	Эквивалентно встроенному методу `str.isalnum`
`isalpha`	Эквивалентно встроенному методу `str.isalpha`
`isdecimal`	Эквивалентно встроенному методу `str.isdecimal`
`isdigit`	Эквивалентно встроенному методу `str.isdigit`
`islower`	Эквивалентно встроенному методу `str.islower`
`isnumeric`	Эквивалентно встроенному методу `str.isnumeric`

Таблица 7.5 (окончание)

Метод	Описание
`isupper`	Эквивалентно встроенному методу `str.isupper`
`join`	Объединяет строки в каждом элементе Series, вставляя между ними указанный разделитель
`len`	Вычисляет длину каждой строки
`lower, upper`	Преобразование регистра; эквивалентно `x.lower()` или `x.upper()` для каждого элемента
`match`	Вызывает `re.match` с указанным регулярным выражением для каждого элемента, возвращает список выделенных групп
`pad`	Дополняет строки пробелами слева, справа или с обеих сторон
`center`	Эквивалентно `pad(side='both')`
`repeat`	Дублирует значения; например, `s.str.repeat(3)` эквивалентно `x * 3` для каждой строки
`replace`	Заменяет вхождения образца указанной строкой
`slice`	Вырезает каждую строку в объекте Series
`split`	Разбивает строки по разделителю или по регулярному выражению
`strip`	Убирает пробельные символы, в том числе знак новой строки, с обеих сторон строки
`rstrip`	Убирает пробельные символы справа
`lstrip`	Убирает пробельные символы слева

7.4. Заключение

Эффективные средства подготовки данных способны значительно повысить продуктивность, поскольку оставляют больше времени для анализа данных. В этой главе мы рассмотрели целый ряд инструментов, но наше изложение было далеко не полным. В следующей главе изучим имеющиеся в pandas средства соединения и группировки.

Глава 8. Переформатирование данных: соединение, комбинирование и изменение формы

Во многих приложениях бывает, что данные разбросаны по многим файлам или базам данных либо организованы так, что их трудно проанализировать. Эта глава посвящена средствам комбинирования, соединения и реорганизации данных.

Сначала познакомимся с концепцией *иерархического индексирования* в pandas, которая широко применяется в некоторых из описываемых далее операций, а затем перейдем к деталям конкретных манипуляций данными. В главе 14 будут продемонстрированы различные применения этих средств.

8.1. Иерархическое индексирование

Иерархическое индексирование – важная особенность pandas, позволяющая организовать несколько (два и более) *уровней* индексирования по одной оси. Говоря абстрактно, это способ работать с многомерными данными, представив их в форме с меньшей размерностью. Начнем с простого примера – создадим объект Series с индексом в виде списка списков или массивов:

```
In [9]: data = pd.Series(np.random.randn(9),
   ...: index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...: [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
```

```
Out[10]:
a  1  -0.204708
   2   0.478943
   3  -0.519439
b  1  -0.555730
   3   1.965781
c  1   1.393406
   2   0.092908
d  2   0.281746
   3   0.769023
dtype: float64
```

Здесь мы видим отформатированное представление Series с мультииндексом (`MultiIndex`). Разрывы в представлении индекса означают «взять значение вышестоящей метки».

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

Для иерархически индексированного объекта возможен доступ по так называемому *частичному* индексу, что позволяет лаконично записывать выборку подмножества данных:

```
In [12]: data['b']
Out[12]:
1  -0.555730
3   1.965781
dtype: float64
```

```
In [13]: data['b':'c']
Out[13]:
b  1  -0.555730
       3 1.965781
c  1   1.393406
   2   0.092908
dtype: float64
```

```
In [14]: data.loc[['b', 'd']]
Out[14]:
b  1  -0.555730
   3   1.965781
d  2   0.281746
   3   0.769023
dtype: float64
```

В некоторых случаях возможна даже выборка с «внутреннего» уровня:

```
In [15]: data.loc[:, 2]
```

```
Out[15]:
a   0.478943
c   0.092908
d   0.281746
dtype: float64
```

Иерархическое индексирование играет важнейшую роль в изменении формы данных и групповых операциях, в том числе в построении сводных таблиц. Например, эти данные можно было бы преобразовать в DataFrame с помощью метода unstack:

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

Обратной к unstack операцией является stack:

```
In [17]: data.unstack().stack()
Out[17]:
a  1 -0.204708
   2  0.478943
   3 -0.519439
b  1 -0.555730
   3  1.965781
c  1  1.393406
   2  0.092908
d  2  0.281746
   3  0.769023
dtype: float64
```

Методы stack и unstack будут подробно рассмотрены в главе 7.

В случае DataFrame иерархический индекс может существовать для любой оси:

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   .....:                     index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
   .....:                     columns=[['Ohio', 'Ohio', 'Colorado'],
   .....:                              ['Green', 'Red', 'Green']])
```

```
In [19]: frame
Out[19]:
       Ohio       Colorado
      Green  Red    Green
a 1       0    1        2
  2       3    4        5
b 1       6    7        8
  2       9   10       11
```

Уровни иерархии могут иметь имена (как строки или любые объекты Python). В таком случае они будут показаны при выводе на консоль (не путайте имена индексов с метками на осях!):

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state    Ohio        Colorado
color   Green   Red     Green
key1 key2
a  1        0     1         2
   2        3     4         5
b  1        6     7         8
   2        9    10        11
```

Не путайте имена индексов `'state'` и `'color'` с метками строк.

Доступ по частичному индексу, как и раньше, позволяет выбирать группы столбцов:

```
In [23]: frame['Ohio']
Out[23]:
color   Green  Red
key1 key2
a  1        0    1
   2        3    4
b  1        6    7
   2        9   10
```

Мультииндекс можно создать отдельно, а затем использовать повторно; в показанном выше объекте DataFrame столбцы с именами уровней можно было бы создать так:

```
pd.MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                          names=['state', 'color'])
```

Переупорядочение и уровни сортировки

Иногда требуется изменить порядок уровней на оси или отсортировать данные по значениям на одном уровне. Метод `swaplevel` принимает номера или имена двух уровней и возвращает новый объект, в котором эти уровни переставлены (но во всех остальных отношениях данные не изменяются):

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state    Ohio         Colorado
color   Green   Red      Green
```

```
key2 key1
1    a      0     1          2
2    a      3     4          5
1    b      6     7          8
2    b      9    10         11
```

С другой стороны, метод sort_index выполняет сортировку данных, используя только значения на одном уровне. После перестановки уровней обычно вызывают также sort_index, чтобы лексикографически отсортировать результат:

```
In [25]: frame.sort_index(level=1)
Out[25]:
state      Ohio      Colorado
color     Green Red     Green
key1 key2
a    1        0   1         2
b    1        6   7         8
a    2        3   4         5
b    2        9  10        11
```

```
In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state      Ohio      Colorado
color     Green Red     Green
key2 key1
1    a        0   1         2
     b        6   7         8
2    a        3   4         5
     b        9  10        11
```

Производительность выборки данных из иерархически индексированных объектов будет гораздо выше, если индекс отсортирован лексикографически начиная с самого внешнего уровня, т. е. в результате вызова sort_index(level=0) или sort_index().

Сводная статистика по уровню

У многих методов объектов DataFrame и Series, вычисляющих сводные и описательные статистики, имеется параметр level для задания уровня, на котором требуется производить агрегирование по конкретной оси. Рассмотрим тот же объект DataFrame, что и выше; мы можем суммировать по уровню для строк или для столбцов:

```
In [27]: frame.sum(level='key2')
Out[27]:
state    Ohio         Colorado
color   Green   Red      Green
key2
1           6     8         10
2          12    14         16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color    Green    Red
key1 key2
a    1       2      1
     2       8      4
b    1      14      7
     2      20     10
```

Реализовано это с помощью имеющегося в pandas механизма `groupby`, который мы подробно рассмотрим позже.

Индексирование с помощью столбцов DataFrame

Не так уж редко возникает необходимость использовать один или несколько столбцов DataFrame в качестве индекса строк; альтернативно можно переместить индекс строк в столбцы DataFrame. Рассмотрим пример:

```
In [29]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
  .....:                    'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
  .....:                    'd': [0, 1, 2, 0, 1, 2, 3]})
```

```
In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

Метод `set_index` объекта DataFrame создает новый DataFrame, используя в качестве индекса один или несколько столбцов:

```
In [31]: frame2 = frame.set_index(['c', 'd'])
```

```
In [32]: frame2
Out[32]:
      a b
c   d
one 0 0 7
    1 1 6
    2 2 5
two 0 3 4
    1 4 3
    2 5 2
    3 6 1
```

По умолчанию столбцы удаляются из DataFrame, хотя их можно и оставить:

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
      a b   c d
c   d
one 0 0 7 one 0
    1 1 6 one 1
    2 2 5 one 2
two 0 3 4 two 0
    1 4 3 two 1
    2 5 2 two 2
    3 6 1 two 3
```

Есть также метод `reset_index`, который делает прямо противоположное `set_index`; уровни иерархического индекса перемещаются в столбцы:

```
In [34]: frame2.reset_index()
Out[34]:
    c d a b
0 one 0 0 7
1 one 1 1 6
2 one 2 2 5
3 two 0 3 4
4 two 1 4 3
5 two 2 5 2
6 two 3 6 1
```

8.2. Комбинирование и слияние наборов данных

Данные, хранящиеся в объектах pandas, можно комбинировать различными способами:

- метод `pandas.merge` соединяет строки объектов DataFrame по одному или нескольким ключам. Эта операция хорошо знакома пользователям реляционных баз данных;
- метод `pandas.concat` склеивает объекты, располагая их в стопке вдоль оси;
- метод экземпляра `combine_first` позволяет сращивать перекрывающиеся данные, чтобы заполнить отсутствующие в одном объекте данные значениями из другого объекта.

Я рассмотрю эти способы на многочисленных примерах. Мы будем неоднократно пользоваться ими в последующих главах.

Слияние объектов DataFrame как в базах данных

Операция *слияния* или *соединения* комбинирует наборы данных, соединяя строки по одному или нескольким *ключам*. Эта операция является одной из

основных в базах данных. Функция merge в pandas – портал ко всем алгоритмам такого рода.

Начнем с простого примера:

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b

In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

Это пример соединения типа *многие-к-одному*; в объекте df1 есть несколько строк с метками a и b, а в df2 – только одна строка для каждого значения в столбце key. Вызов merge для таких объектов дает:

```
In [39]: pd.merge(df1, df2)
Out[39]:
  data1  key  data2
0     0    b      1
1     1    b      1
2     6    b      1
3     2    a      0
4     4    a      0
5     5    a      0
```

Обратите внимание, что я не указал, по какому столбцу производить соединение. В таком случае merge использует в качестве ключей столбцы с одинаковыми именами. Однако рекомендуется все же указывать столбцы явно:

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
  data1  key  data2
0     0    b      1
1     1    b      1
2     6    b      1
```

```
3     2    a      0
4     4    a      0
5     5    a      0
```

Если имена столбцов в объектах различаются, то можно задать их порознь:

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})
In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})
In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
  data1 lkey  data2 rkey
0     0    b      1    b
1     1    b      1    b
2     6    b      1    b
3     2    a      0    a
4     4    a      0    a
5     5    a      0    a
```

Вероятно, вы обратили внимание, что значения 'c' и 'd' и ассоциированные с ними данные отсутствуют в результирующем объекте. По умолчанию функция merge производит внутреннее соединение ('inner'); в результирующий объект попадают только ключи, присутствующие в обоих объектах-аргументах. Альтернативы – 'left', 'right' и 'outer'. В случае внешнего соединения ('outer') берется объединение ключей, т. е. получается то же самое, что при совместном применении левого и правого соединений:

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
  data1 key  data2
0   0.0   b    1.0
1   1.0   b    1.0
2   6.0   b    1.0
3   2.0   a    0.0
4   4.0   a    0.0
5   5.0   a    0.0
6   3.0   c    NaN
7   NaN   d    2.0
```

В табл. 8.1 перечислены возможные значения аргумента how.

Таблица 8.1. Различные типы соединения, задаваемые аргументом how

Значение	Поведение
'inner'	Брать только комбинации ключей, встречающиеся в обеих таблицах
'left'	Брать все ключи, встречающиеся в левой таблице
'right'	Брать все ключи, встречающиеся в правой таблице
'outer'	Брать все комбинации ключей

Для слияния типа *многие-ко-многим* поведение корректно определено, хотя на первый взгляд неочевидно. Вот пример:

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                     'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                     'data2': range(5)})

In [47]: df1
Out[47]:
  data1 key
0     0   b
1     1   b
2     2   a
3     3   c
4     4   a
5     5   b

In [48]: df2
Out[48]:
  data2 key
0     0   a
1     1   b
2     2   a
3     3   b
4     4   d

In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
  data1 key data2
0     0   b   1.0
1     0   b   3.0
2     1   b   1.0
3     1   b   3.0
4     2   a   0.0
5     2   a   2.0
6     3   c   NaN
7     4   a   0.0
8     4   a   2.0
9     5   b   1.0
10    5   b   3.0
```

Соединение *многие-ко-многим* порождает декартово произведение строк. Поскольку в левом объекте DataFrame было три строки с ключом `'b'`, а в правом – две, то в результирующем объекте таких строк получилось шесть. Метод соединения оказывает влияние только на множество различных ключей в результате:

```
In [50]: pd.merge(df1, df2, how='inner')
```

```
Out[50]:
   data1 key data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      5   b      1
5      5   b      3
6      2   a      0
7      2   a      2
8      4   a      0
9      4   a      2
```

Для соединения по нескольким ключам следует передать список имен столбцов:

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                      'key2': ['one', 'two', 'one'],
   ....:                      'lval': [1, 2, 3]})
```

```
In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                       'key2': ['one', 'one', 'one', 'two'],
   ....:                       'rval': [4, 5, 6, 7]})
```

```
In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[53]:
   key1 key2 lval rval
0  foo  one  1.0  4.0
1  foo  one  1.0  5.0
2  foo  two  2.0  NaN
3  bar  one  3.0  6.0
4  bar  two  NaN  7.0
```

Чтобы определить, какие комбинации ключей появятся в результате при данном выборе метода слияния, полезно представить несколько ключей как массив кортежей, используемый в качестве единственного ключа соединения (хотя на самом деле операция реализована не так).

При соединении по столбцам индексы над переданными объектами DataFrame отбрасываются.

Последний момент, касающийся операций слияния, – обработка одинаковых имен столбцов. Хотя эту проблему можно решить вручную (см. раздел о переименовании меток на осях ниже), у функции `merge` имеется параметр `suffixes`, позволяющий задать строки, которые должны дописываться в конец одинаковых имен в левом и правом объектах DataFrame:

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  key1 key2_x lval key2_y rval
0  foo    one    1    one    4
```

```
1  foo    one    1    one    5
2  foo    two    2    one    4
3  foo    two    2    one    5
4  bar    one    3    one    6
5  bar    one    3    two    7
```

```
In [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[55]:
  key1 key2_left lval key2_right rval
0  foo       one    1        one    4
1  foo       one    1        one    5
2  foo       two    2        one    4
3  foo       two    2        one    5
4  bar       one    3        one    6
5  bar       one    3        two    7
```

В табл. 8.2 приведена справка по аргументам функции `merge`. Соединение с использованием индекса строк DataFrame – тема следующего раздела.

Таблица 8.2. Аргументы функции `merge`

Аргумент	Описание
`left`	Объект DataFrame в левой части операции слияния
`right`	Объект DataFrame в правой части операции слияния
`how`	Допустимые значения: `'inner'`, `'outer'`, `'left'`, `'right'`
`on`	Имена столбцов, по которым производится соединение. Должны присутствовать в обоих объектах DataFrame. Если не заданы и не указаны никакие другие ключи соединения, то используются имена столбцов, общих для обоих объектов
`left_on`	Столбцы левого DataFrame, используемые как ключи соединения
`right_on`	Столбцы правого DataFrame, используемые как ключи соединения
`left_index`	Использовать индекс строк левого DataFrame в качестве его ключа соединения (или нескольких ключей в случае мультииндекса)
`right_index`	То же, что `left_index`, но для правого DataFrame
`sort`	Сортировать слитые данные лексикографически по ключам соединения; по умолчанию `True`. Иногда при работе с большими наборами данных лучше отключить
`suffixes`	Кортеж строк, которые дописываются в конец совпадающих имен столбцов; по умолчанию (`'_x'`, `'_y'`). Например, если в обоих объектах DataFrame встречается столбец `'data'`, то в результирующем объекте появятся столбцы `'data_x'` и `'data_y'`
`copy`	Если равен `False`, то в некоторых особых случаях разрешается не копировать данные в результирующую структуру. По умолчанию данные копируются всегда
`indicator`	Добавляет специальный столбец `_merge`, который сообщает об источнике каждой строки; он может принимать значения `'left_only'`, `'right_only'` или `'both'` в зависимости от того, как строка попала в результат соединения

Соединение по индексу

Иногда ключ (или ключи) соединения находится в индексе объекта DataFrame. В таком случае можно задать параметр `left_index=True` или `right_index=True` (или то и другое), чтобы указать, что в качестве ключа соединения следует использовать индекс:

```
In [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....: 'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [58]: left1
Out[58]:
  key value
0   a     0
1   b     1
2   a     2
3   a     3
4   b     4
5   c     5

In [59]: right1
Out[59]:
  group_val
a       3.5
b       7.0

In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
  key value group_val
0   a     0       3.5
2   a     2       3.5
3   a     3       3.5
1   b     1       7.0
4   b     4       7.0
```

По умолчанию соединение производится по пересекающимся ключам, но можно вместо пересечения выполнить объединение, указав внешнее соединение:

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[61]:
   key  value group_val
0    a      0       3.5
2    a      2       3.5
3    a      3       3.5
1    b      1       7.0
4    b      4       7.0
5    c      5       NaN
```

В случае иерархически индексированных данных ситуация немного усложняется:

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
   ....:                                 'Nevada', 'Nevada'],
   ....:                        'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                        'data': np.arange(5.)})
```

```
In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
   ....:                       index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',
   ....:                               'Ohio', 'Ohio'],
   ....:                              [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                       columns=['event1', 'event2'])
```

```
In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002
```

```
In [65]: righth
Out[65]:
             event1 event2
Nevada 2001       0      1
       2000       2      3
Ohio   2000       4      5
       2000       6      7
       2001       8      9
       2002      10     11
```


В этом случае необходимо перечислить столбцы, по которым производится соединение, в виде списка (обратите внимание на обработку повторяющихся значений в индексе, когда `how='outer'`):

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[66]:
   data    key1 key2 event1 event2
0   0.0    Ohio 2000      4      5
0   0.0    Ohio 2000      6      7
1   1.0    Ohio 2001      8      9
2   2.0    Ohio 2002     10     11
3   3.0  Nevada 2001      0      1
```



```
In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:           right_index=True, how='outer')
Out[67]:
   data    key1 key2 event1 event2
0   0.0    Ohio 2000    4.0    5.0
0   0.0    Ohio 2000    6.0    7.0
1   1.0    Ohio 2001    8.0    9.0
2   2.0    Ohio 2002   10.0   11.0
3   3.0  Nevada 2001    0.0    1.0
4   4.0  Nevada 2002    NaN    NaN
4   NaN  Nevada 2000    2.0    3.0
```

Употребление индексов в обеих частях соединения тоже не проблема:

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....:                      index=['a', 'c', 'e'],
   ....:                      columns=['Ohio', 'Nevada'])

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
....:                         index=['b', 'c', 'd', 'e'],
....:                         columns=['Missouri', 'Alabama'])

In [70]: left2
Out[70]:
   Ohio  Nevada
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0

In [71]: right2
Out[71]:
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0

In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
Out[72]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

В классе DataFrame есть и более удобный метод экземпляра `join` для слияния по индексу. Его также можно использовать для комбинирования нескольких объектов DataFrame, обладающих одинаковыми или похожими индексами, но непересекающимися столбцами. В предыдущем примере можно было бы написать:

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

Отчасти из-за необходимости поддерживать совместимость (с очень старыми версиями pandas) метод `join` объекта DataFrame выполняет левое внешнее соединение, в точности сохраняя индекс строк левого фрейма. Он также

поддерживает соединение с индексом переданного DataFrame по одному из столбцов вызывающего:

```
In [74]: left1.join(right1, on='key')
Out[74]:
   key  value group_val
0    a      0       3.5
1    b      1       7.0
2    a      2       3.5
3    a      3       3.5
4    b      4       7.0
5    c      5       NaN
```

Наконец, в случае простых операций слияния индекса с индексом можно передать список объектов DataFrame методу `join` в качестве альтернативы использованию более общей функции `concat`, которая описана ниже:

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....: index=['a', 'c', 'e', 'f'],
   ....: columns=['New York', 'Oregon'])

In [76]: another
Out[76]:
   New York Oregon
a       7.0    8.0
c       9.0   10.0
e      11.0   12.0
f      16.0   17.0

In [77]: left2.join([right2, another])
Out[77]:
   Ohio Nevada Missouri Alabama New York Oregon
a   1.0   2.0       NaN     NaN      7.0    8.0
c   3.0   4.0       9.0    10.0      9.0   10.0
e   5.0   6.0      13.0    14.0     11.0   12.0

In [78]: left2.join([right2, another], how='outer')
Out[78]:
   Ohio Nevada Missouri Alabama New York Oregon
a   1.0    2.0      NaN     NaN      7.0    8.0
b   NaN    NaN      7.0     8.0      NaN    NaN
c   3.0    4.0      9.0    10.0      9.0   10.0
d   NaN    NaN     11.0    12.0      NaN    NaN
e   5.0    6.0     13.0    14.0     11.0   12.0
f   NaN    NaN      NaN     NaN     16.0   17.0
```

Конкатенация вдоль оси

Еще одну операцию комбинирования данных разные авторы называют по-разному: конкатенация, связывание или укладка. В библиотеке NumPy имеется функция `concatenate` для выполнения этой операции над массивами:

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
array([[ 0, 1, 2, 3, 0, 1, 2, 3],
       [ 4, 5, 6, 7, 4, 5, 6, 7],
       [ 8, 9, 10, 11, 8, 9, 10, 11]])
```

В контексте объектов pandas, Series и DataFrame наличие помеченных осей позволяет обобщить конкатенацию массивов. В частности, нужно решить следующие вопросы:

- если объекты по-разному проиндексированы по другим осям, следует ли объединять различные элементы на этих осях, или нужно использовать только общие значения (пересечение)?
- нужно ли иметь возможность идентифицировать группы в результирующем объекте?
- содержит ли «ось конкатенации» данные, которые необходимо сохранить? Во многих случаях подразумеваемые по умолчанию целочисленные метки в объекте DataFrame в процессе конкатенации лучше отбросить.

Функция `concat` в pandas дает согласованные ответы на эти вопросы. Я покажу, как она работает, на примерах. Допустим, имеются три объекта Series с непересекающимися индексами:

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

Если передать их функции `concat` списком, то она склеит данные и индексы:

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a  0
b  1
c  2
d  3
e  4
f  5
g  6
dtype: int64
```

По умолчанию concat работает вдоль оси axis=0, порождая новый объект Series. Но если передать параметр axis=1, то результатом будет DataFrame (в нем axis=1 – ось столбцов):

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
     0    1    2
a    0  NaN  NaN
b    1  NaN  NaN
c  NaN    2  NaN
d  NaN    3  NaN
e  NaN    4  NaN
f  NaN  NaN    5
g  NaN  NaN    6
```

В данном случае на другой оси нет перекрытия, и она, как видно, является отсортированным объединением (внешним соединением) индексов. Но можно образовать и пересечение индексов, если передать параметр join='inner':

```
In [87]: s4 = pd.concat([s1, s3])

In [88]: s4
Out[88]:
a   0
b   1
f   5
g   6
dtype: int64

In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
    0 1
a 0.0 0
b 1.0 1
f NaN 5
g NaN 6

In [90]: pd.concat([s1, s4], axis=1, join='inner')
Out[90]:
  0 1
a 0 0
b 1 1
```

В последнем примере метки 'f' и 'g' пропали, поскольку был задан аргумент join='inner'.

Можно даже задать, какие метки будут использоваться на других осях – с помощью параметра join_axes:

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[91]:
     0    1
a  0.0  0.0
```

```
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

Проблема может возникнуть из-за того, что в результирующем объекте не видно, конкатенацией каких объектов он получен. Допустим, что вы на самом деле хотите построить иерархический индекс на оси конкатенации. Для этого используется аргумент keys:

```
In [92]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [93]: result
Out[93]:
one    a  0
       b  1
two    a  0
       b  1
three  f  5
       g  6

In [94]: result.unstack()
Out[94]:
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

При комбинировании Series вдоль оси axis=1 элементы списка keys становятся заголовками столбцов объекта DataFrame:

```
In [95]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
Out[95]:
   one  two  three
a  0.0  NaN    NaN
b  1.0  NaN    NaN
c  NaN  2.0    NaN
d  NaN  3.0    NaN
e  NaN  4.0    NaN
f  NaN  NaN    5.0
g  NaN  NaN    6.0
```

Эта логика обобщается и на объекты DataFrame:

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   ....:                    columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   ....:                    columns=['three', 'four'])

In [98]: df1
Out[98]:
   one two
a    0   1
```

```
b    2   3
c    4   5

In [99]: df2
Out[99]:
   three four
a      5    6
c      7    8

In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[100]:
  level1    level2
    one two  three four
a     0   1    5.0   6.0
b     2   3    NaN   NaN
c     4   5    7.0   8.0
```

Если передать не список, а словарь объектов, то роль аргумента keys будут играть ключи словаря:

```
In [101]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[101]:
   level1       level2
      one  two   three  four
a       0    1     5.0   6.0
b       2    3     NaN   NaN
c       4    5     7.0   8.0
```

Дополнительные аргументы управляют созданием иерархического индекса (см. табл. 8.3). Например, можно поименовать созданные уровни на оси с помощью аргумента names:

```
In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
   ....:           names=['upper', 'lower'])
Out[78]:
upper   level1       level2
lower      one  two   three  four
a            0    1     5.0   6.0
b            2    3     NaN   NaN
c            4    5     7.0   8.0
```

Последнее замечание касается объектов DataFrame, в которых индекс строк не имеет смысла в контексте анализа:

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
```

```
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
```

```
In [106]: df2
Out[106]:
          b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

В таком случае можно передать параметр `ignore_index=True`:

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985       NaN  3.248944
4  0.302614 -0.577087       NaN  0.124121
```

Таблица 8.3. Аргументы функции `concat`

Аргумент	Описание
`objs`	Список или словарь конкатенируемых объектов pandas. Единственный обязательный аргумент
`axis`	Ось, вдоль которой производится конкатенация, по умолчанию 0
`join`	Допустимые значения: `'inner'`, `'outer'`, по умолчанию `'outer'`; следует ли пересекать (`inner`) или объединять (`outer`) индексы вдоль других осей
`join_axes`	Какие конкретно индексы использовать для других n – 1 осей вместо выполнения пересечения или объединения
`keys`	Значения, которые ассоциируются с конкатенируемыми объектами и образуют иерархический индекс вдоль оси конкатенации. Может быть список или массив произвольных значений, а также массив кортежей или список массивов (если в параметре `levels` передаются массивы для нескольких уровней)
`levels`	Конкретные индексы, которые используются на одном или нескольких уровнях иерархического индекса, если задан параметр `keys`
`names`	Имена создаваемых уровней иерархического индекса, если заданы параметры `keys` и (или) `levels`
`verify_integrity`	Проверить новую ось в конкатенированном объекте на наличие дубликатов и, если они имеются, возбудить исключение. По умолчанию `False` – дубликаты разрешены
`ignore_index`	Не сохранять индексы вдоль оси конкатенации, а вместо этого создать новый индекс `range(total_length)`

Комбинирование перекрывающихся данных

Есть еще одна ситуация, которую нельзя выразить как слияние или конкатенацию. Речь идет о двух наборах данных, индексы которых полностью или частично пересекаются. В качестве пояснительного примера рассмотрим функцию NumPy `where`, которая выражает векторный аналог if-else:

```
In [108]: a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
   .....:               index=['f', 'e', 'd', 'c', 'b', 'a'])

In [109]: b = pd.Series(np.arange(len(a), dtype=np.float64),
   .....:               index=['f', 'e', 'd', 'c', 'b', 'a'])

In [110]: b[-1] = np.nan

In [111]: a
Out[111]:
f   NaN
e   2.5
d   NaN
c   3.5
b   4.5
a   NaN
dtype: float64

In [112]: b
Out[112]:
f   0.0
e   1.0
d   2.0
c   3.0
b   4.0
a   NaN
dtype: float64

In [113]: np.where(pd.isnull(a), b, a)
Out[113]: array([ 0. , 2.5, 2. , 3.5, 4.5, nan])
```

У объекта Series имеется метод `combine_first`, который выполняет эквивалент этой операции плюс обычное для pandas выравнивание данных:

```
In [114]: b[:-2].combine_first(a[2:])
Out[114]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
dtype: float64
```

В случае DataFrame метод `combine_first` делает то же самое для каждого столбца, так что можно считать, что он подставляет вместо данных, отсутствующих в вызывающем объекте, данные из объекта, переданного в аргументе:

```
In [115]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
   ....:                      'b': [np.nan, 2., np.nan, 6.],
   ....:                      'c': range(2, 18, 4)})
```

```
In [116]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
   ....: 'b': [np.nan, 3., 4., 6., 8.]})

In [117]: df1.combine_first(df2)
Out[117]:
    a    b   c
0 1.0  NaN   2
1 NaN  2.0   6
2 5.0  NaN  10
3 NaN  6.0  14

In [118]: df2
Out[118]:
    a    b
0 5.0  NaN
1 4.0  3.0
2 NaN  4.0
3 3.0  6.0
4 7.0  8.0

In [119]: df1.combine_first(df2)
Out[119]:
    a    b    c
0 1.0  NaN  2.0
1 4.0  2.0  6.0
2 5.0  4.0 10.0
3 3.0  6.0 14.0
4 7.0  8.0  NaN
```

8.3. Изменение формы и поворот

Существует ряд фундаментальных операций реорганизации табличных данных. Иногда их называют *изменением формы* (reshape), а иногда – *поворотом* (pivot).

Изменение формы с помощью иерархического индексирования

Иерархическое индексирование дает естественный способ реорганизовать данные в DataFrame. Есть два основных действия:

- `stack` – это «поворот», который переносит данные из столбцов в строки;
- `Unstack` – обратный поворот, который переносит данные из строк в столбцы.

Проиллюстрирую эти операции примерами. Рассмотрим небольшой DataFrame, в котором индексы строк и столбцов – массивы строк.

```
In [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   ....:           index=pd.Index(['Ohio', 'Colorado'], name='state'),
   ....:           columns=pd.Index(['one', 'two', 'three'], name='number'))
```

```
In [121]: data
Out[121]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Метод stack поворачивает таблицу, так что столбцы оказываются строками, и в результате получается объект Series:

```
In [122]: result = data.stack()
```

```
In [123]: result
Out[123]:
state     number
Ohio      one     0
          two     1
          three   2
Colorado  one     3
          two     4
          three   5
dtype: int64
```

Имея иерархически проиндексированный объект Series, мы можем восстановить DataFrame методом unstack:

```
In [124]: result.unstack()
Out[124]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

По умолчанию поворачивается самый внутренний уровень (как и в случае stack). Но можно повернуть и любой другой, если указать номер или имя уровня:

```
In [125]: result.unstack(0)
Out[125]:
state  Ohio Colorado
number
one       0        3
two       1        4
three     2        5
```

```
In [126]: result.unstack('state')
Out[126]:
state  Ohio Colorado
number
one       0        3
two       1        4
three     2        5
```

При обратном повороте могут появиться отсутствующие данные, если не каждое значение на указанном уровне присутствует в каждой подгруппе:

```
In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])

In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [130]: data2
Out[130]:
one  a  0
     b  1
     c  2
     d  3
two  c  4
     d  5
     e  6
dtype: int64

In [131]: data2.unstack()
Out[131]:
      a     b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

При выполнении поворота отсутствующие данные по умолчанию отфильтровываются, поэтому операция обратима:

```
In [132]: data2.unstack()
Out[132]:
      a     b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0

In [133]: data2.unstack().stack()
Out[133]:
one a  0.0
    b  1.0
    c  2.0
    d  3.0
two c  4.0
    d  5.0
    e  6.0
dtype: float64

In [134]: data2.unstack().stack(dropna=False)
Out[134]:
one a  0.0
    b  1.0
    c  2.0
    d  3.0
    e  NaN
```

```
two a  NaN
    b  NaN
    c  4.0
    d  5.0
    e  6.0
dtype: float64
```

В случае обратного поворота DataFrame поворачиваемый уровень становится самым нижним уровнем результирующего объекта:

```
In [135]: df = pd.DataFrame({'left': result, 'right': result + 5},
   .....:                   columns=pd.Index(['left', 'right'], name='side'))

In [136]: df
Out[136]:
Side         left right
state    number
Ohio     one    0     5
         Two    1     6
         three  2     7
Colorado one    3     8
         two    4     9
         three  5    10

In [137]: df.unstack('state')
Out[137]:
side    left          right
state   Ohio Colorado  Ohio Colorado
number
one        0        3     5        8
two        1        4     6        9
three      2        5     7       10
```

При вызове stack можно указать имя поворачиваемой оси:

```
In [138]: df.unstack('state').stack('side')
Out[138]:
state         Colorado Ohio
number side
one    left          3    0
       right         8    5
two    left          4    1
       right         9    6
three  left          5    2
       right        10    7
```

Поворот из «длинного» в «широкий» формат

Стандартный способ хранения нескольких временных рядов в базах данных и в CSV-файлах – так называемый *длинный* формат (*в столбик*). Загрузим

демонстрационные данные и займемся переформатированием временных рядов и другими операциями очистки данных:

```
In [139]: data = pd.read_csv('examples/macrodata.csv')

In [140]: data.head()
Out[140]:
    year quarter  realgdp realcons  realinv realgovt realdpi    cpi \
0 1959.0     1.0 2710.349   1707.4  286.898  470.045  1886.9  28.98
1 1959.0     2.0 2778.801   1733.7  310.859  481.301  1919.7  29.15
2 1959.0     3.0 2775.488   1751.8  289.226  491.260  1916.4  29.35
3 1959.0     4.0 2785.204   1753.7  299.356  484.052  1931.3  29.37
4 1960.0     1.0 2847.699   1770.5  331.722  462.199  1955.5  29.54
     m1 tbilrate unemp      pop  infl realint
0 139.7     2.82   5.8  177.146  0.00    0.00
1 141.7     3.08   5.1  177.830  2.34    0.74
2 140.5     3.82   5.3  178.657  2.74    1.09
3 140.0     4.33   5.6  179.386  0.27    4.06
4 139.6     3.50   5.2  180.007  2.31    1.19

In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....:                          name='date')

In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to_timestamp('D', 'end')

In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

С функцией `PeriodIndex` мы поближе познакомимся в главе 11. В двух словах, она объединяет столбцы `year` и `quarter`, создавая специальный тип временного интервала. Теперь `ldata` выглядит так:

```
In [146]: ldata[:10]
Out[146]:
        date    item    value
0 1959-03-31 realgdp 2710.349
1 1959-03-31    infl    0.000
2 1959-03-31   unemp    5.800
3 1959-06-30 realgdp 2778.801
4 1959-06-30    infl    2.340
5 1959-06-30   unemp    5.100
6 1959-09-30 realgdp 2775.488
7 1959-09-30    infl    2.740
8 1959-09-30   unemp    5.300
9 1959-12-31 realgdp 2785.204
```

Это и называется *длинным* форматом для нескольких временных рядов или других данных наблюдений с двумя и более ключами (в данном случае клю-

чами являются дата и показатель item). Каждая строка таблицы соответствует одному наблюдению.

Так данные часто хранятся в реляционных базах данных типа MySQL, поскольку при наличии фиксированной схемы (совокупность имен и типов данных столбцов) количество различных значений в столбце item может увеличиваться или уменьшаться при добавлении или удалении данных. В примере выше пара столбцов date и item обычно выступает в роли первичного ключа (в терминологии реляционных баз данных), благодаря которому обеспечивается целостность данных и упрощаются многие операции соединения. Иногда с данными в таком формате трудно работать; предпочтительнее иметь объект DataFrame, содержащий по одному столбцу на каждое уникальное значение item и проиндексированный временными метками в столбце date. Метод pivot объекта DataFrame именно такое преобразование и выполняет:

```
In [147]: pivoted = ldata.pivot('date', 'item', 'value')

In [148]: pivoted
Out[148]:
item        infl    realgdp unemp
date
1959-03-31  0.00   2710.349   5.8
1959-06-30  2.34   2778.801   5.1
1959-09-30  2.74   2775.488   5.3
1959-12-31  0.27   2785.204   5.6
1960-03-31  2.31   2847.699   5.2
1960-06-30  0.14   2834.390   5.2
1960-09-30  2.70   2839.022   5.6
1960-12-31  1.21   2802.616   6.3
1961-03-31 -0.40   2819.264   6.8
1961-06-30  1.47   2872.005   7.0
...          ...        ...   ...
2007-06-30  2.75  13203.977   4.5
2007-09-30  3.45  13321.109   4.7
2007-12-31  6.38  13391.249   4.8
2008-03-31  2.82  13366.865   4.9
2008-06-30  8.53  13415.266   5.4
2008-09-30 -3.16  13324.600   6.0
2008-12-31 -8.79  13141.920   6.9
2009-03-31  0.94  12925.410   8.1
2009-06-30  3.37  12901.504   9.2
2009-09-30  3.56  12990.341   9.6
[203 rows x 3 columns]
```

Первые два аргумента – столбцы, которые будут выступать в роли индексов строк и столбцов, а последний необязательный аргумент – столбец, в котором находятся данные, вставляемые в DataFrame. Допустим, что имеется два столбца значений, форму которых требуется изменить одновременно:

```
In [149]: ldata['value2'] = np.random.randn(len(ldata))

In [150]: ldata[:10]
Out[150]:
                 date     item     value     value2
0 1959-03-31 00:00:00  realgdp  2710.349   1.669025
1 1959-03-31 00:00:00     infl     0.000  -0.438570
2 1959-03-31 00:00:00    unemp     5.800  -0.539741
3 1959-06-30 00:00:00  realgdp  2778.801   0.476985
4 1959-06-30 00:00:00     infl     2.340   3.248944
5 1959-06-30 00:00:00    unemp     5.100  -1.021228
6 1959-09-30 00:00:00  realgdp  2775.488  -0.577087
7 1959-09-30 00:00:00     infl     2.740   0.124121
8 1959-09-30 00:00:00    unemp     5.300   0.302614
9 1959-12-31 00:00:00  realgdp  2785.204   0.523772
```

Опустив последний аргумент, получим DataFrame с иерархическими столбцами:

```
In [151]: pivoted = ldata.pivot('date', 'item')

In [152]: pivoted[:5]
Out[152]:
           value                      value2
item        infl   realgdp  unemp       infl    realgdp      unemp
date
1959-03-31  0.00  2710.349    5.8  -0.438570   1.669025  -0.539741
1959-06-30  2.34  2778.801    5.1   3.248944   0.476985  -1.021228
1959-09-30  2.74  2775.488    5.3   0.124121  -0.577087   0.302614
1959-12-31  0.27  2785.204    5.6   0.000940   0.523772   1.343810
1960-03-31  2.31  2847.699    5.2  -0.831154  -0.713544  -2.370232

In [153]: pivoted['value'][:5]
Out[153]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

Отметим, что метод `pivot` – это не более чем сокращенный способ создания иерархического индекса с помощью `set_index` и последующего вызова `unstack`:

```
In [154]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [155]: unstacked[:7]
Out[155]:
value value2
item infl realgdp unemp infl realgdp unemp
date
```

```
           value                          value2
item        infl    realgdp  unemp       infl    realgdp      unemp
date
1959-03-31  0.00  2710.349    5.8  -0.438570   1.669025  -0.539741
1959-06-30  2.34  2778.801    5.1   3.248944   0.476985  -1.021228
1959-09-30  2.74  2775.488    5.3   0.124121  -0.577087   0.302614
1959-12-31  0.27  2785.204    5.6   0.000940   0.523772   1.343810
1960-03-31  2.31  2847.699    5.2  -0.831154  -0.713544  -2.370232
1960-06-30  0.14  2834.390    5.2  -0.970736  -1.541996  -1.307030
1960-09-30  2.70  2839.022    5.6   0.377984   0.286350  -0.753887
```

Поворот из «широкого» в «длинный» формат

Обратной к `pivot` операцией является `pandas.melt`. Вместо того чтобы преобразовывать один столбец в несколько в новом объекте DataFrame, она объединяет несколько столбцов в один, порождая DataFrame длиннее входного. Рассмотрим пример:

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....:                    'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6],
   .....:                    'C': [7, 8, 9]})
```

```
In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

Столбец `'key'` может быть индикатором группы, а остальные столбцы – значениями данных. При использовании функции `pandas.melt` необходимо указать, какие столбцы являются индикаторами группы (если таковые имеются). Будем считать, что в данном случае `'key'` – единственный индикатор группы:

```
In [159]: melted = pd.melt(df, ['key'])
```

```
In [160]: melted
Out[160]:
  key variable value
0 foo        A     1
1 bar        A     2
2 baz        A     3
3 foo        B     4
4 bar        B     5
5 baz        B     6
6 foo        C     7
7 bar        C     8
8 baz        C     9
```

Применив pivot, можем вернуться к исходной форме:

```
In [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped
Out[162]:
variable A  B  C
key
bar      2  5  8
baz      3  6  9
foo      1  4  7
```

Поскольку в результате работы pivot из столбца создается индекс, используемый как метки строк, возможно, понадобится вызвать reset_index, чтобы восстановить из индекса столбец:

```
In [163]: reshaped.reset_index()
Out[163]:
variable key  A  B  C
0        bar  2  5  8
1        baz  3  6  9
2        foo  1  4  7
```

Можно также указать, какое подмножество столбцов следует использовать как значения:

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
  key variable value
0 foo        A     1
1 bar        A     2
2 baz        A     3
3 foo        B     4
4 bar        B     5
5 baz        B     6
```

Функцию pandas.melt можно использовать и без идентификаторов групп:

```
In [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
  variable value
0        A     1
1        A     2
2        A     3
3        B     4
4        B     5
5        B     6
6        C     7
7        C     8
8        C     9

In [166]: pd.melt(df, value_vars=['key', 'A', 'B'])
```

```
Out[166]:
  variable value
0      key   foo
1      key   bar
2      key   baz
3        A     1
4        A     2
5        A     3
6        B     4
7        B     5
8        B     6
```

8.4. Заключение

Теперь, вооружившись базовыми знаниями о применении pandas для импорта, очистки и реорганизации данных, мы готовы перейти к визуализации с помощью matplotlib. Но мы еще вернемся к pandas, когда начнем обсуждать дополнительные средства аналитики.

Глава 9. Построение графиков и визуализация

Информативная визуализация (называемая также *построением графиков*) – одна из важнейших задач анализа данных. Она может быть частью процесса исследования, например применяться для выявления выбросов, определения необходимых преобразований данных или поиска идей для построения моделей. В других случаях построение интерактивной визуализации для вебсайта может быть конечной целью. Для Python имеется много дополнительных библиотек статической и динамической визуализации, но я буду использовать в основном matplotlib (http://matplotlib.sourceforge.net) и надстроенные над ней библиотеки.

Matplotlib – это пакет для построения графиков (главным образом двумерных) полиграфического качества. Проект был основан Джоном Хантером в 2002 году с целью реализовать на Python интерфейс, аналогичный MATLAB. Впоследствии сообщества matplotlib и IPython совместно работали над тем, чтобы упростить интерактивное построение графиков из оболочки IPython (а теперь и Jupyter-блокнотов). Matplotlib поддерживает разнообразные графические интерфейсы пользователя во всех операционных системах, а также умеет экспортировать графические данные во всех векторных и растровых форматах: PDF, SVG, JPG, PNG, BMP, GIF и т. д. С его помощью я построил почти все рисунки для этой книги, за исключением нескольких диаграмм.

Со временем над matplotlib было надстроено много дополнительных библиотек визуализации. Одну из них, seaborn (http://seaborn.pydata.org/), мы будем изучать в этой главе.

Для проработки приведенных в данной главе примеров кода проще всего воспользоваться интерактивным построением графиков в Jupyter-блокноте. Чтобы настроить этот режим, выполните в Jupyter-блокноте такую команду:

```
%matplotlib notebook
```

9.1. Краткое введение в API библиотеки matplotlib

При работе с matplotlib мы будем использовать следующее соглашение об импорте:

```
In [11]: import matplotlib.pyplot as plt
```

После выполнения команды `%matplotlib notebook` в Jupyter (или просто `%matplotlib` в IPython) уже можно создать простой график. Если все настроено правильно, то должна появиться прямая линия, показанная на рис. 9.1:

```
In [12]: import numpy as np

In [13]: data = np.arange(10)

In [14]: data
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)
```

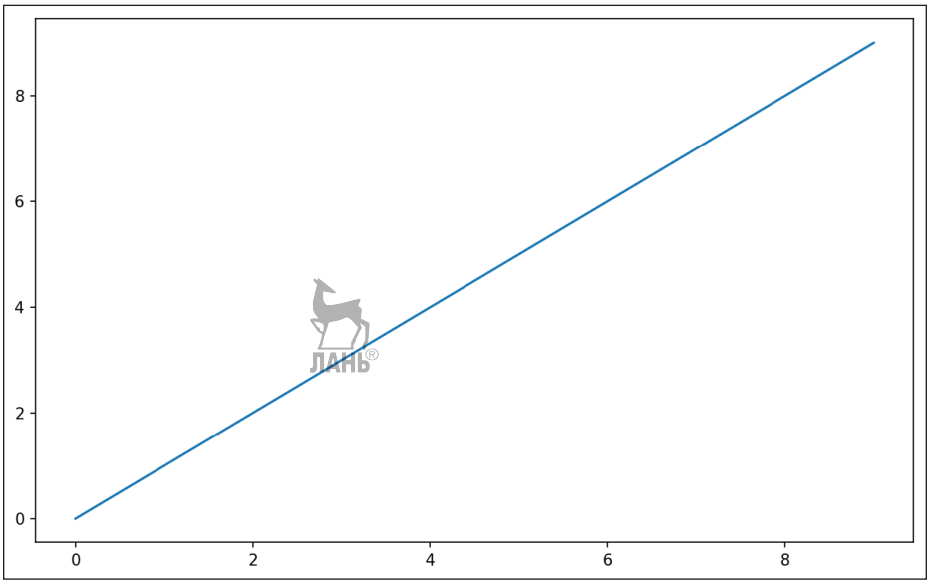


Рис. 9.1. Простой линейный график

Библиотеки типа seaborn и встроенные в pandas функции построения графиков берут на себя многие рутинные детали, но если предусмотренных в них параметров вам недостаточно, то придется разбираться с API библиотеки matplotlib.

В этой книге не хватит места для рассмотрения функциональности matplotlib во всей полноте, но достаточно показать, что делать, и дальше вы все освоите самостоятельно. Для изучения продвинутых возможностей нет ничего лучше галереи и документации matplotlib.

Рисунки и подграфики

Графики в matplotlib «живут» внутри объекта рисунка `Figure`. Создать новый рисунок можно методом `plt.figure`:

```
In [16]: fig = plt.figure()
```

В IPython появится пустое окно графика, но в Jupyter не появится ничего, пока мы не выполним еще несколько команд. У команды `plt.figure()` имеется ряд параметров, в частности `figsize` гарантирует, что при сохранении рисунка на диске у него будут определенные размер и отношение сторон.

Нельзя создать график, имея пустой рисунок. Сначала нужно создать один или несколько подграфиков с помощью метода `add_subplot`:

```
In [17]: ax1 = fig.add_subplot(2, 2, 1)
```

Это означает, что рисунок будет расчерчен сеткой 2×2, и мы выбираем первый из четырех подграфиков (нумерация начинается с 1). Если создать следующие два подграфика, то получится как на рис. 9.2.

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)

In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

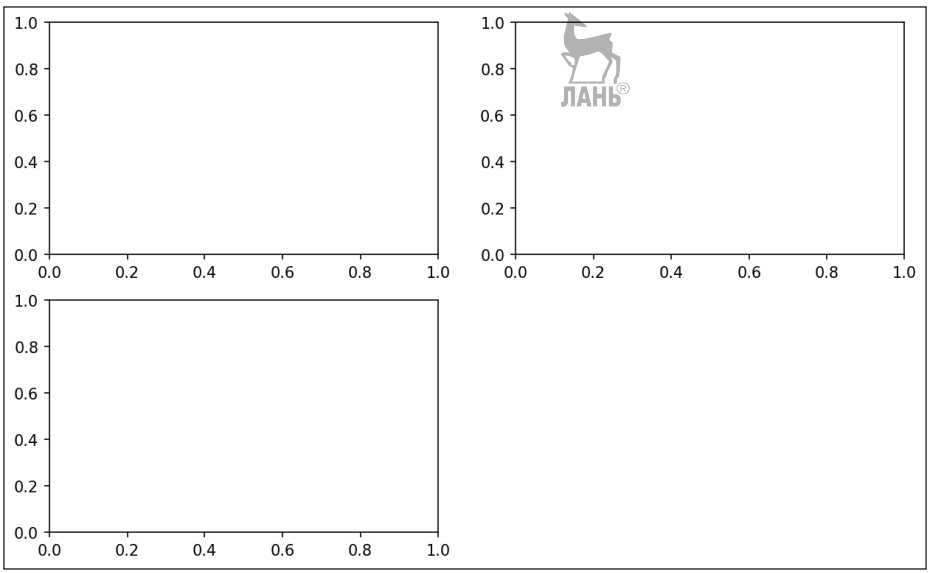


Рис. 9.2. Пустой рисунок matplotlib с тремя подграфиками

Один из нюансов работы с Jupyter-блокнотами заключается в том, что графики сбрасываются после вычисления каждой ячейки, поэтому если графики достаточно сложны, то следует помещать все команды построения в одну ячейку блокнота.

В примере ниже все команды находятся в одной ячейке:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

При выполнении команды построения графика, например `plt.plot([1.5, 3.5, -2, 1.6])`, matplotlib рисует на последнем использованном рисунке и подграфике (при необходимости создав то и другое) и тем самым маскирует создание рисунка и подграфика. Следовательно, выполнив показанную ниже команду, мы получим картину, изображенную на рис. 9.3:

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

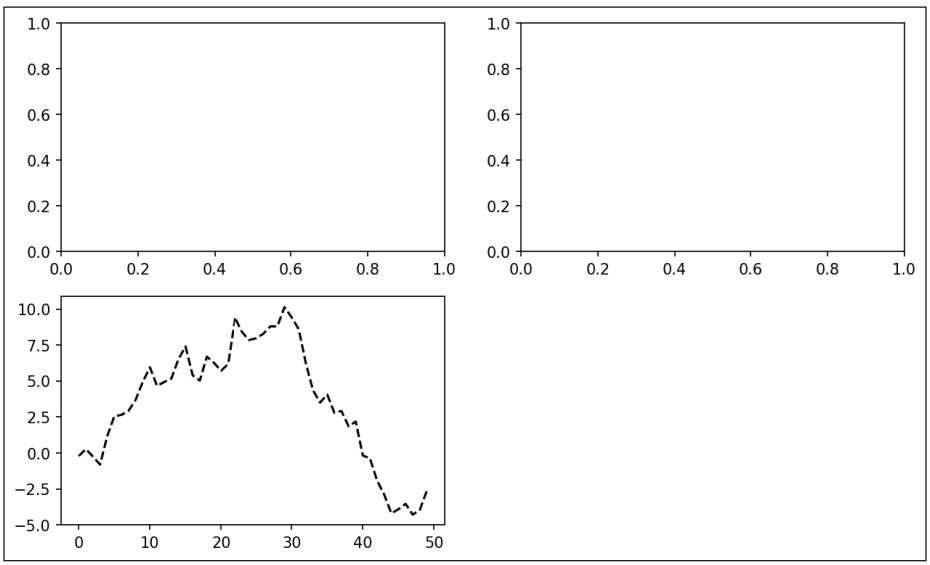


Рис. 9.3. Визуализация данных после построения одного графика

Параметр *стиля* `'k--'` говорит matplotlib, что график нужно рисовать черной штриховой линией. Метод `fig.add_subplot` возвращает объект `AxesSubplot`, который позволяет рисовать в другом пустом подграфике, вызывая его методы экземпляра (см. рис. 9.4):

```
In [21]: _ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)

In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

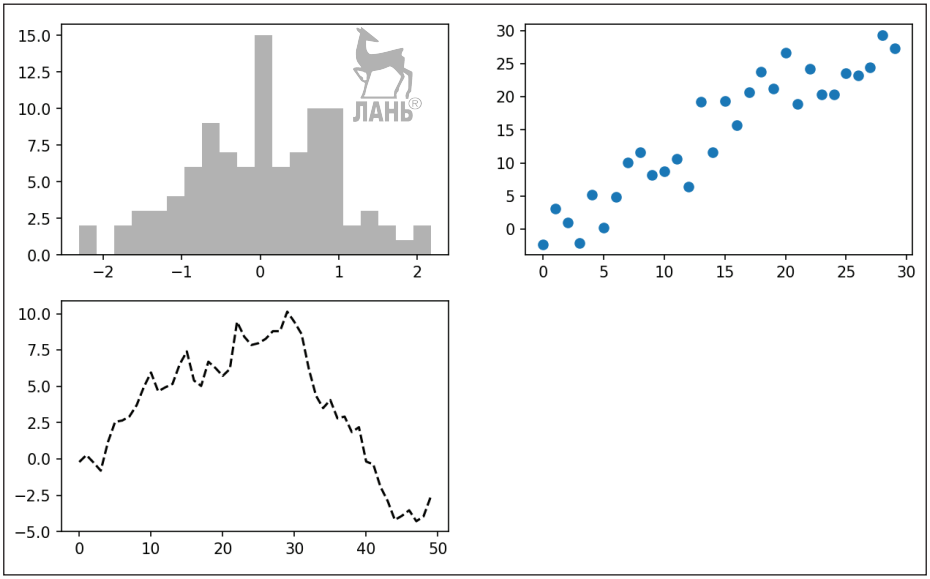


Рис. 9.4. Визуализация данных
после построения дополнительных графиков

Полный перечень типов графиков имеется в документации по matplotlib.

Поскольку создание рисунка с несколькими подграфиками, расположенными определенным образом, – типичная задача, существует вспомогательный метод `plt.subplots`, который создает новый рисунок и возвращает массив NumPy, содержащий созданные в нем объекты подграфиков:

```
In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626374048>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb62625db00>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6262f6c88>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6261a36a0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb626181860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6260fd4e0>]], dtype
=object)
```

Это очень полезно, потому что к массиву `axes` вполне можно обращаться как к двумерному массиву, например `axes[0, 1]`. Можно также указать, что подграфики должны иметь общую ось x или y, задав параметры `sharex` и `sharey` соответственно. Особенно это удобно, когда надо сравнить данные в одном масштабе; иначе matplotlib автоматически и независимо выбирает масштаб графика. Подробнее об этом методе см. табл. 9.1.

Таблица 9.1. Параметры метода pyplot.subplots

Аргумент	Описание
nrows	Число строк в сетке подграфиков
ncols	Число столбцов в сетке подграфиков
sharex	Все подграфики должны иметь одинаковые риски на оси X (настройка xlim отражается на всех подграфиках)
sharey	Все подграфики должны иметь одинаковые риски на оси Y (настройка ylim отражается на всех подграфиках)
subplot_kw	Словарь ключевых слов для создания подграфиков
**fig_kw	Дополнительные ключевые слова используются при создании рисунка, например plt.subplots(2, 2,figsize=(8, 6))

Задание свободного места вокруг подграфиков

По умолчанию matplotlib оставляет пустое место вокруг каждого подграфика и между подграфиками. Размер этого места определяется высотой и шириной графика, так что если изменить размер графика программно или вручную (изменив размер окна), то график автоматически перестроится. Величину промежутка легко изменить с помощью метода subplots_adjust объекта Figure, который также доступен в виде функции верхнего уровня:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

Параметры wspace и hspace определяют, какой процент от ширины (соответственно высоты) рисунка должен составлять промежуток между подграфиками. В примере ниже я задал нулевой промежуток (рис. 9.5):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(randn(500), bins=50, color='k', alpha=0.5)
    plt.subplots_adjust(wspace=0, hspace=0)
```

Вы, наверное, заметили, что риски на осях наложились друг на друга. matplotlib это не проверяет, поэтому если такое происходит, то вам придется самостоятельно подкорректировать риски, явно указав их положения и надписи (как это сделать, мы узнаем в следующих разделах).

Цвета, маркеры и стили линий

Главная функция matplotlib – plot – принимает массивы координат x и y, а также необязательную строку, в которой закодированы цвет и стиль линии. Например, чтобы нарисовать график зависимости y от x зеленой штриховой линией, нужно выполнить следующий вызов:

```
ax.plot(x, y, 'g--')
```

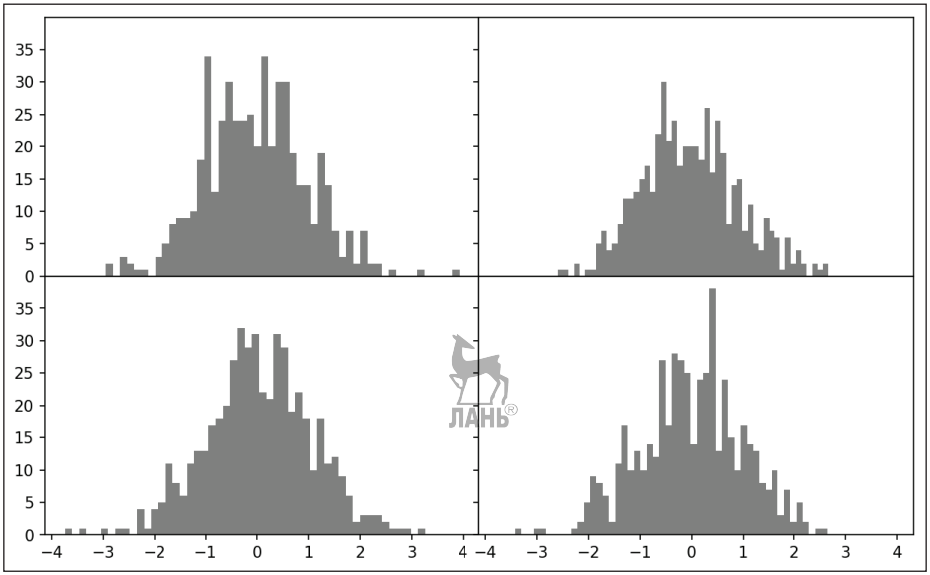


Рис. 9.5. Визуализация данных,
в которой подграфики не разделены промежутками

Такой способ задания цвета и стиля линий в виде строки не более чем удобство; на практике, когда графики строятся из программы, лучше не запутывать код строковыми обозначениями стиля. Этот график можно было бы описать и более понятно:

```
ax.plot(x, y, linestyle='--', color='g')
```

Существует ряд сокращений для наиболее употребительных цветов, но вообще любой цвет можно представить своим RGB-значением (например, `'#CECECE'`). Полный перечень стилей линий имеется в строке документации для функции `plot` (в IPython или Jupyter введите `plot?`).

Линейные графики могут быть также снабжены *маркерами*, обозначающими точки, по которым построен график. Поскольку matplotlib создает непрерывный линейный график, производя интерполяцию между точками, иногда не ясно, где же находятся исходные точки. Маркер можно задать в строке стиля: сначала цвет, потом тип маркера и в конце стиль линии (рис. 9.6):

То же самое можно записать явно:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

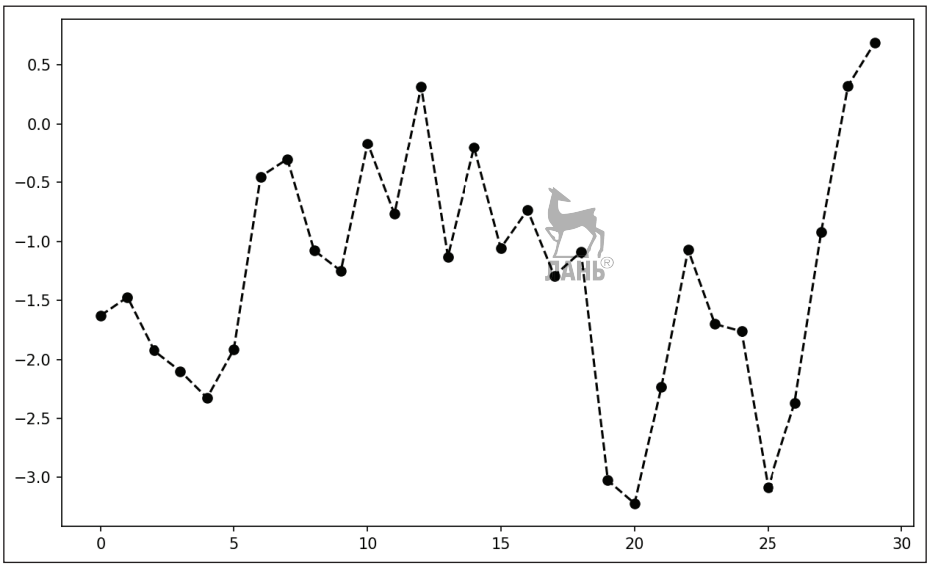


Рис. 9.6. Линейный график с примером маркеров

По умолчанию на линейных графиках соседние точки соединяются отрезками прямой, т. е. производится линейная интерполяция. Параметр `drawstyle` позволяет изменить этот режим:

```
In [33]: data = randn(30).cumsum()

In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>]

In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>]

In [36]: plt.legend(loc='best')
```

Вы, наверное, обратили внимание на строчки вида `<matplotlib.lines.Line2D at ...>`. Matplotlib возвращает объекты, ссылающиеся на только что добавленную часть графика. Обычно эту информацию можно смело игнорировать. В данном случае, поскольку мы передавали функции `plot` аргумент `label`, мы можем с помощью метода `plt.legend` нанести на график надпись, описывающую каждую линию.

Для создания надписи необходимо вызвать метод `plt.legend` (или `ax.legend`, если вы сохранили ссылки на оси) вне зависимости от того, передавали вы аргумент `label` при построении графика или нет.

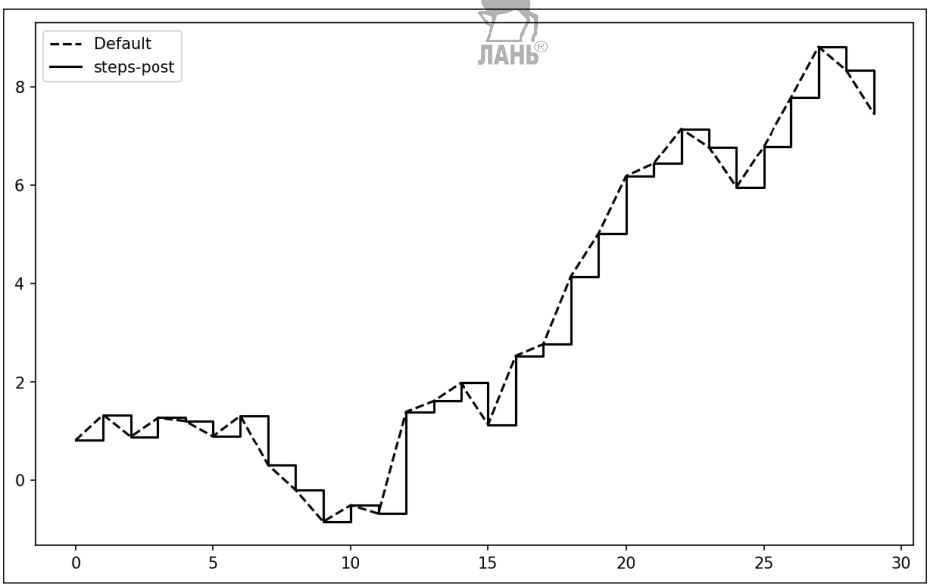


Рис. 9.7. Линейный график с различными значениями параметра `drawstyle`

Риски, метки и надписи

Для оформления большинства графиков существует два основных способа: процедурный интерфейс `pyplot` (который будет понятен пользователям MATLAB) и собственный объектно-ориентированный matplotlib API.

Интерфейс `pyplot`, предназначенный для интерактивного использования, состоит из методов `xlim`, `xticks` и `xticklabels`. Они управляют размером области, занятой графиком, положением и метками рисок соответственно. Использовать их можно двумя способами:

- при вызове без аргументов возвращается текущее значение параметра. Например, метод `plt.xlim()` возвращает текущий диапазон значений по оси X;
- при вызове с аргументами устанавливается новое значение параметра. Например, в результате вызова `plt.xlim([0, 10])` диапазон значений по оси X устанавливается от 0 до 10.

Все подобные методы действуют на активный или созданный последним объект `AxesSubplot`. Каждому из них соответствуют два метода самого объекта подграфика; в случае `xlim` это методы `ax.get_xlim` и `ax.set_xlim`. Я предпочитаю пользоваться методами экземпляра подграфика, чтобы код получался понятнее (в особенности когда работаю с несколькими подграфиками), но вы, конечно, вольны выбирать то, что вам больше нравится.

Задание названия графика, названий осей, рисок и их меток

Чтобы проиллюстрировать оформление осей, я создам простой рисунок и в нем график случайного блуждания (рис. 9.8):

```
In [37]: fig = plt.figure()

In [38]: ax = fig.add_subplot(1, 1, 1)

In [39]: ax.plot(np.random.randn(1000).cumsum())
```

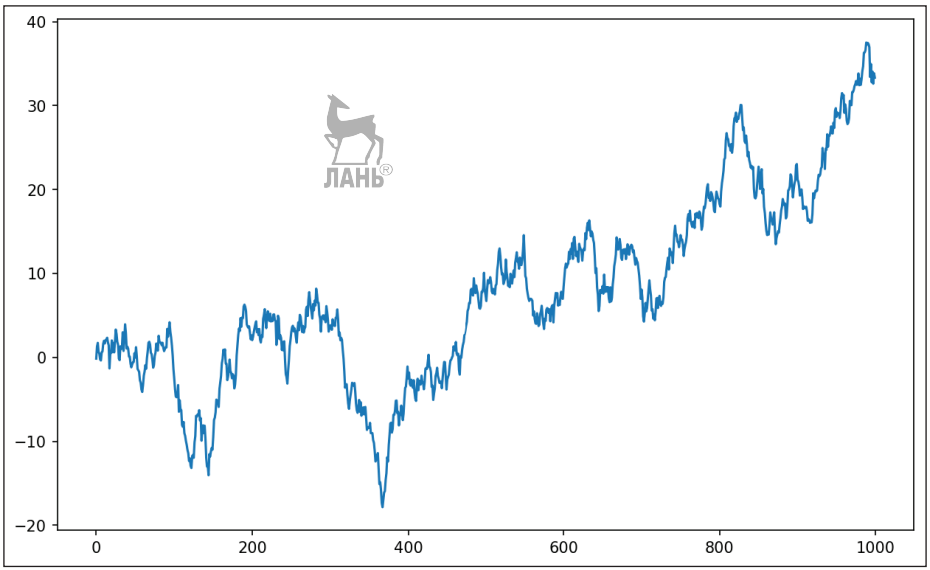


Рис. 9.8. Простой график для иллюстрации рисок (с метками)

Для изменения рисок на оси X проще всего воспользоваться методами `set_xticks` и `set_xticklabels`. Первый говорит matplotlib, где в пределах диапазона значений данных ставить риски; по умолчанию их числовые значения изображаются также и в виде меток. Но можно задать и другие метки с помощью метода `set_xticklabels`:

```
In [40]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
In [41]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
   ....:                             rotation=30, fontsize='small')
```

Аргумент `rotation` устанавливает угол наклона меток рисок к оси x равным 30°. Наконец, метод `set_xlabel` именует ось x, а метод `set_title` задает название подграфика (см. окончательный результат на рис. 9.9):

```
In [42]: ax.set_title('My first matplotlib plot')
Out[42]: <matplotlib.text.Text at 0x7fb624d055f8>

In [43]: ax.set_xlabel('Stages')
```

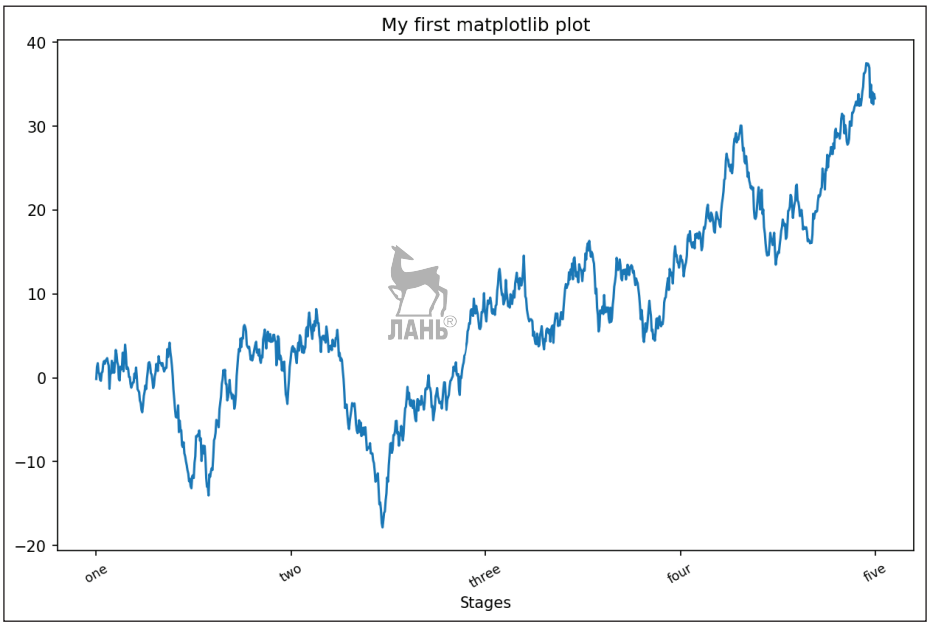


Рис. 9.9. Простой график для иллюстрации рисок

Модификация оси y производится точно так же с заменой `x` на `y`. В классе оси имеется метод `set`, позволяющий задавать сразу несколько свойств графика. Так, предыдущий пример можно записать в следующем виде:

```
props = {
    'title': 'My first matplotlib plot',
    'xlabel': 'Stages'
}
ax.set(**props)
```

Добавление пояснительных надписей

Пояснительная надпись – еще один важный элемент оформления графика. Добавить ее можно двумя способами. Проще всего передать аргумент `label` при добавлении каждого нового графика:

```
In [44]: from numpy.random import randn

In [45]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>]

In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>]

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>]
```

После этого можно вызвать метод `ax.legend()` или `plt.legend()`, и он автоматически создаст пояснительную надпись. Получившийся график показан на рис. 9.10:

```
In [49]: ax.legend(loc='best')
```

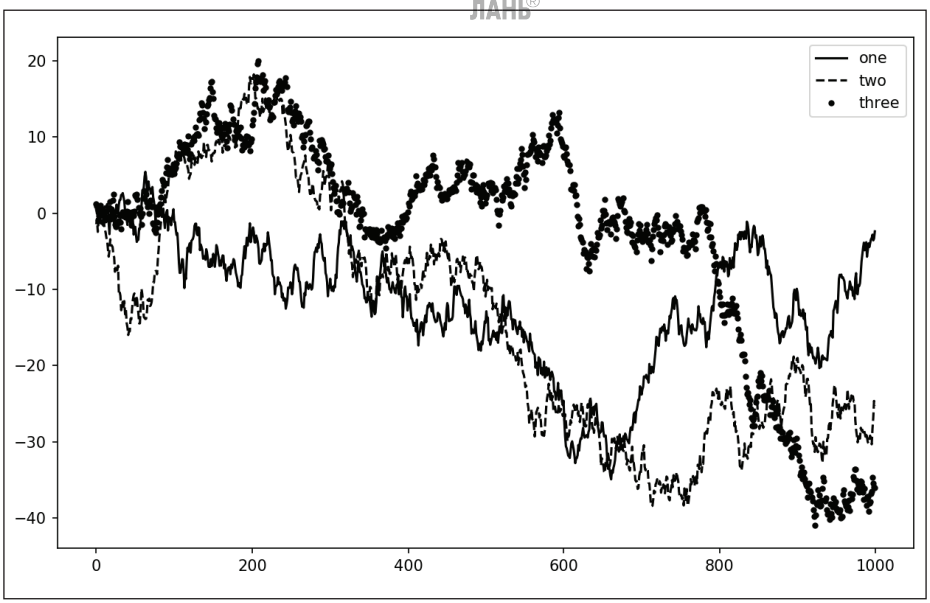


Рис. 9.10. Простой график с тремя линиями и пояснительной надписью

Аргумент `loc` метода `legend` может принимать и другие значения. Дополнительные сведения см. в строке документации (введите `ax.legend?`).

Аргумент `loc` говорит, где поместить надпись. Если вам все равно, задавайте значение `'best'`, потому что тогда место будет выбрано так, чтобы по возможности не загораживать сам график. Чтобы исключить из надписи один или несколько элементов, не задавайте параметр `label` вовсе или задайте `label='_nolegend_'`.

Аннотации и рисование в подграфике

Помимо стандартных типов графиков, разрешается наносить на график свои аннотации, которые могут содержать текст, стрелки и другие фигуры. Для добавления аннотаций и текста предназначены функции `text`, `arrow` и `annotate`. Функция `text` наносит на график текст начиная с точки с заданными координатами (`x`, `y`), с факультативной стилизацией:

```
ax.text(x, y, 'Hello world!',
        family='monospace', fontsize=10)
```

В аннотациях могут встречаться текст и стрелки. В качестве примера построим график цен закрытия по индексу S&P 500 начиная с 2007 года (данные получены с сайта Yahoo! Finance) и аннотируем его некоторыми важными датами, относящимися к финансовому кризису 2008–2009 годов. Проще всего воспроизвести этот код, введя его в одну ячейку Jupyter-блокнота. Результат изображен на рис. 9.11.

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True)
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 50),
                xytext=(date, spx.asof(date) + 200),
                arrowprops=dict(facecolor='black'),
                horizontalalignment='left', verticalalignment='top')

# Оставить только диапазон 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in 2008-2009 financial crisis')
```

Отметим несколько важных моментов. Метод `ax.annotate` умеет рисовать метку в позиции, заданной координатами x и y. Мы воспользовались методами `set_xlim` и `set_ylim`, чтобы вручную задать нижнюю и верхнюю границы графика, а не полагаться на умолчания matplotlib. Наконец, метод `ax.set_title` задает название графика в целом.

В галерее matplotlib в Сети есть много других поучительных примеров аннотаций.

Для рисования фигур требуется больше усилий. В matplotlib имеются объекты, соответствующие многим стандартным фигурам, они называются *патчами* (patches). Часть из них, например `Rectangle` и `Circle`, находится в модуле `matplotlib.pyplot`, а весь набор – в модуле `matplotlib.patches`.

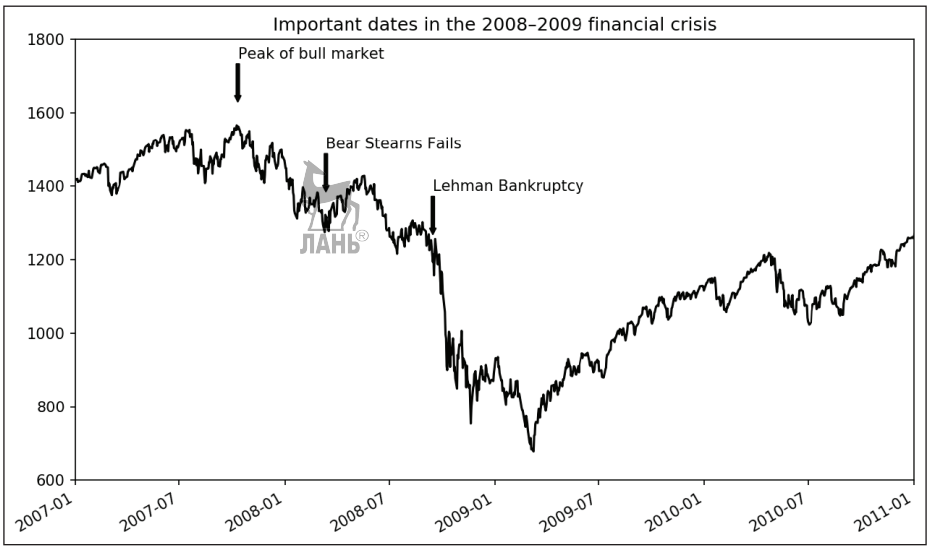


Рис. 9.11. Важные даты, относящиеся к финансовому кризису 2008–2009 годов

Чтобы поместить на график фигуру, мы создаем объект патча `shp` и добавляем его в подграфик, вызывая метод `ax.add_patch(shp)` (см. рис. 9.12):

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

Заглянув в код многих знакомых типов графиков, вы увидите, что они составлены из патчей.

Сохранение графиков в файле

Активный рисунок можно сохранить в файле методом `plt.savefig`. Этот метод эквивалентен методу экземпляра рисунка `savefig`. Например, чтобы сохранить рисунок в формате SVG, достаточно указать только имя файла:

```
plt.savefig('figpath.svg')
```

Формат выводится из расширения имени файла. Если бы мы задали файл с расширением `.pdf`, то рисунок был бы сохранен в формате PDF. При публикации графики я часто использую два параметра: `dpi` (разрешение в точках на

дюйм) и bbox_inches (размер пустого места вокруг рисунка). Чтобы получить тот же самый график в формате PNG с минимальным обрамлением и разрешением 400 DPI, нужно было бы написать:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

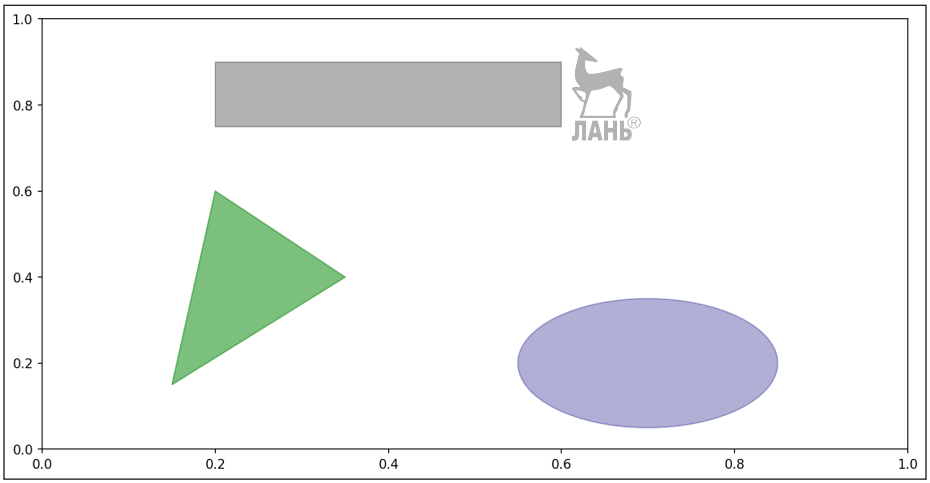


Рис. 9.12. Визуализация данных, составленная из трех разных патчей

Метод savefig может писать не только на диск, а в любой похожий на файл объект, например StringIO:

```
from io import StringIO
buffer = StringIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

В табл. 9.2 перечислены некоторые другие аргументы метода savefig.

Таблица 9.2. Аргументы метода Figure.savefig

Аргумент	Описание
fname	Строка, содержащая путь к файлу или похожий на файл объект Python. Формат рисунка определяется по расширению имени файла, например: PDF для .pdf и PNG для .png
dpi	Разрешение рисунка в точках на дюйм; по умолчанию 100, но может настраиваться
facecolor, edgecolor	Цвет фона рисунка вне области, занятой подграфиками. По умолчанию 'w' (белый)
format	Явно заданный формат файла ('png', 'pdf', 'svg', 'ps', 'eps' и т. д.)
bbox_inches	Какую часть рисунка сохранять. Если задано значение 'tight', то метод пытается обрезать все пустое место вокруг рисунка

Конфигурирование matplotlib

В начальной конфигурации matplotlib заданы цветовые схемы и умолчания, ориентированные главным образом на подготовку рисунков к публикации. По счастью, почти все аспекты поведения по умолчанию можно сконфигурировать с помощью обширного набора глобальных параметров, определяющих размер рисунка, промежутки между подграфиками, цвета, размеры шрифтов, стили сетки и т. д. Есть два основных способа работы с системой конфигурирования matplotlib. Первый – программный, с помощью метода `rc`. Например, чтобы глобально задать размер рисунка равным 10×10, нужно написать:

```
plt.rc('figure', figsize=(10, 10))
```

Первый аргумент `rc` – настраиваемый компонент, например: `'figure'`, `'axes'`, `'xtick'`, `'ytick'`, `'grid'`, `'legend'` и т. д. Вслед за ним идут позиционные аргументы, задающие параметры этого компонента. В программе описывать параметры проще всего в виде словаря:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size' : 'small'}
plt.rc('font', **font_options)
```

Если требуется более тщательная настройка, то можно воспользоваться входящим в состав matplotlib конфигурационным файлом *matplotlibrc* в каталоге *matplotlib/mpl-data*, где перечислены все параметры. Если вы настроите этот файл и поместите его в свой домашний каталог под именем *.matplotlibrc*, то он будет загружаться при каждом использовании matplotlib.

В следующем разделе мы увидим, что в пакете seaborn имеется несколько встроенных тем, или *стилей*, надстроенных над конфигурационной системой matplotlib.

9.2. Построение графиков с помощью pandas и seaborn

Библиотека matplotlib – средство довольно низкого уровня. График собирается из базовых компонентов: способ отображения данных (тип графика: линейный график, столбчатая диаграмма, коробчатая диаграмма, диаграмма рассеяния, контурный график и т. д.), пояснительная надпись, название, метки рисок и прочие аннотации.

В библиотеке pandas может быть несколько столбцов данных, а с ними метки строк и метки столбцов. В саму pandas встроены методы построения, упрощающие создание визуализаций объектов DataFrame и Series. Существует другая библиотека, seaborn, разработанная Майклом Уэскомом (Michael Waskom) для создания статистических графиков. Seaborn упрощает создание многих типов визуализаций.

В результате импорта seaborn изменяются принятые в matplotlib по умолчанию схемы цветов и стили графиков – чтобы повысить удобочитаемость и улучшить эстетическое восприятие. Даже если вы не собираетесь использовать API seaborn, все равно стоит импортировать ее, чтобы улучшить внешний вид графиков matplotlib.

Линейные графики

У объектов Series и DataFrame имеется метод `plot`, который умеет строить графики разных типов. По умолчанию он строит линейные графики (см. рис. 8.13):

```
In [60]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))

In [61]: s.plot()
```

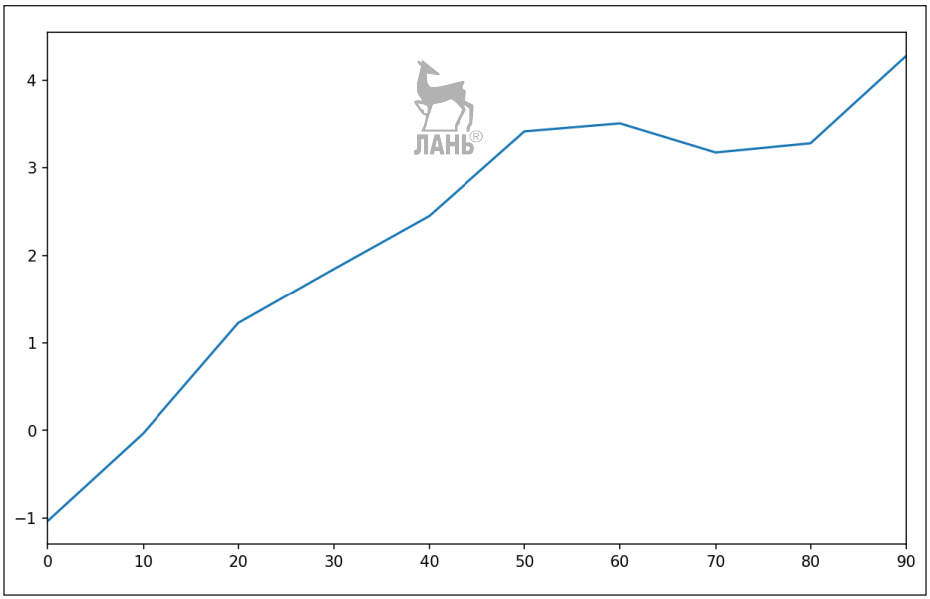


Рис. 9.13. Простой пример графика для объекта Series

Индекс объекта Series передается matplotlib для нанесения рисок на ось x, но это можно отключить, задав параметр `use_index=False`. Риски и диапазон значений на оси x можно настраивать с помощью параметров `xticks` и `xlim`, а на оси y – с помощью параметров `yticks` и `ylim`. Полный перечень параметров метода `plot` приведен в табл. 9.3. О некоторых я расскажу в этом разделе, а остальные оставлю вам для самостоятельного изучения.

Большинство методов построения графиков в pandas принимает необязательный параметр `ax` – объект подграфика matplotlib. Это позволяет гибко расположить подграфики в сетке.

Таблица 9.3. Параметры метода Series.plot

Аргумент	Описание
`label`	Метка для пояснительной надписи на графике
`ax`	Объект подграфика matplotlib, внутри которого строится график. Если параметр не задан, то используется активный подграфик
`style`	Строка стиля, например `'ko--'`, которая передается matplotlib
`alpha`	Уровень непрозрачности графика (число от 0 до 1)
`kind`	Может принимать значения `'line'`, `'bar'`, `'barh'`, `'kde'`
`logy`	Использовать логарифмический масштаб по оси y
`use_index`	Брать метки рисок из индекса объекта
`rot`	Угол поворота меток рисок (от 0 до 360)
`xticks`	Значения рисок на оси x
`yticks`	Значения рисок на оси y
`xlim`	Границы по оси x (например, [0, 10])
`ylim`	Границы по оси y
`grid`	Отображать координатную сетку (по умолчанию включено)

Метод `plot` объекта DataFrame строит отдельные графики каждого столбца внутри одного подграфика и автоматически создает пояснительную надпись (см. рис. 9.14).

```
In [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
   ....: columns=['A', 'B', 'C', 'D'],
   ....: index=np.arange(0, 100, 10))

In [63]: df.plot()
```

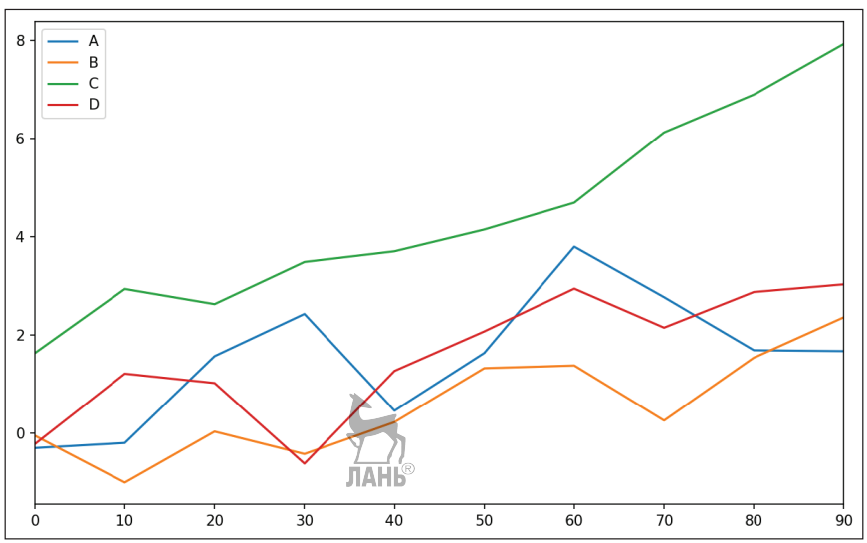


Рис. 9.14. Простой пример графика для объекта DataFrame

Атрибут plot содержит «семейство» методов для различных типов графиков. Например, df.plot() эквивалентно df.plot.line(). Некоторые из этих методов мы рассмотрим ниже.

Дополнительные именованные аргументы метода plot без изменения передаются соответствующей функции matplotlib, поэтому, внимательно изучив API matplotlib, вы сможете настраивать графики более точно.

У объекта DataFrame есть ряд параметров, которые гибко описывают обработку столбцов. Например, поясняют, где нужно строить их графики – внутри одного и того же подграфика или внутри разных подграфиков. Все они перечислены в табл. 9.4.

Таблица 9.4. Параметры метода DataFrame.plot

Аргумент	Описание
subplots	Рисовать график каждого столбца DataFrame в отдельном подграфике
sharex	Если subplots=True, то совместно использовать ось x, объединяя риски и границы
sharey	Если subplots=True, то совместно использовать ось y
figsize	Размеры создаваемого рисунка в виде кортежа
title	Название графика в виде строки
legend	Помещать в подграфик пояснительную надпись (по умолчанию True)
sort_columns	Строить графики столбцов в алфавитном порядке; по умолчанию используется существующий порядок столбцов

О построении графиков временных рядов см. главу 11.

Столбчатые диаграммы

Методы plot.bar() и plot.barh() строят соответственно вертикальную и горизонтальную столбчатые диаграммы. В этом случае индекс Series или DataFrame будет использоваться для нанесения рисок на ось x (bar) или y (barh) (см. рис. 9.15):

```
In [64]: fig, axes = plt.subplots(2, 1)

In [65]: data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))

In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb62493d470>

In [67]: data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

Аргументы color='k' и alpha=0.7 задают цвет диаграмм (черный) и частичную прозрачность столбиков.

В случае DataFrame значения каждой строки объединяются в группы столбиков, расположенные поодаль друг от друга (рис. 9.16).

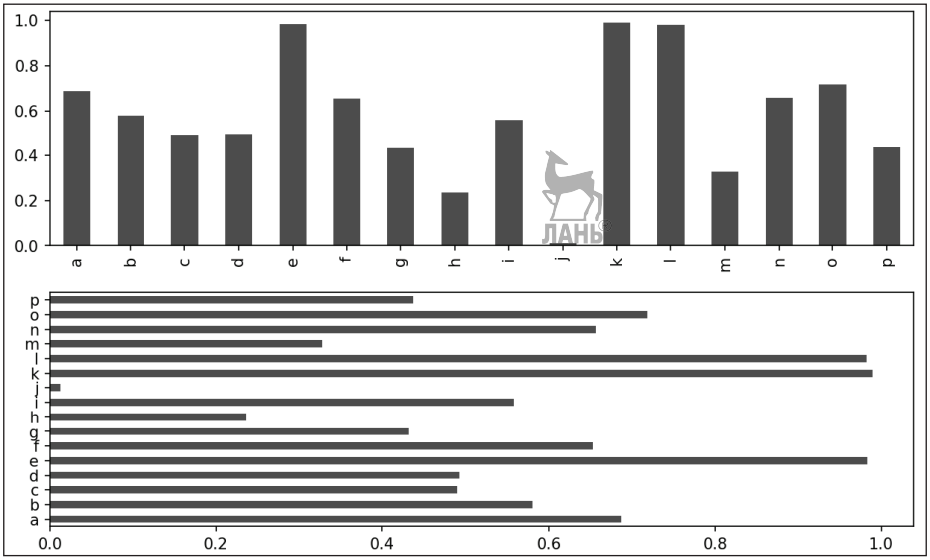


Рис. 9.15. Примеры горизонтальной и вертикальной столбчатых диаграмм

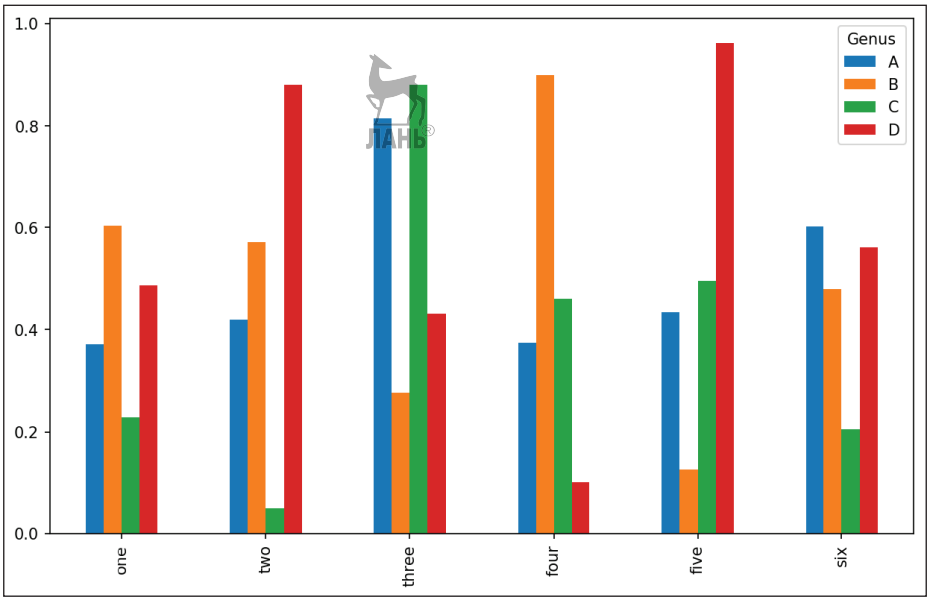


Рис. 9.16. Столбчатая диаграмма для DataFrame

```
In [69]: df = DataFrame(np.random.rand(6, 4),
   ....:                index=['one', 'two', 'three', 'four', 'five', 'six'],
   ....:                columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
```

```
In [70]: df
Out[70]:
Genus         A         B         C         D
one    0.370670  0.602792  0.229159  0.486744
two    0.420082  0.571653  0.049024  0.880592
three  0.814568  0.277160  0.880316  0.431326
four   0.374020  0.899420  0.460304  0.100843
five   0.433270  0.125107  0.494675  0.961825
six    0.601648  0.478576  0.205690  0.560547

In [71]: df.plot.bar()
```

Обратите внимание, что название столбцов DataFrame – «Genus» – используется в заголовке пояснительной надписи.

Для построения составной столбчатой диаграммы по объекту DataFrame нужно задать параметр `stacked=True`, тогда столбики, соответствующие значению в каждой строке, будут приставлены друг к другу (рис. 9.17):

```
In [73]: df.plot(kind='barh', stacked=True, alpha=0.5)
```

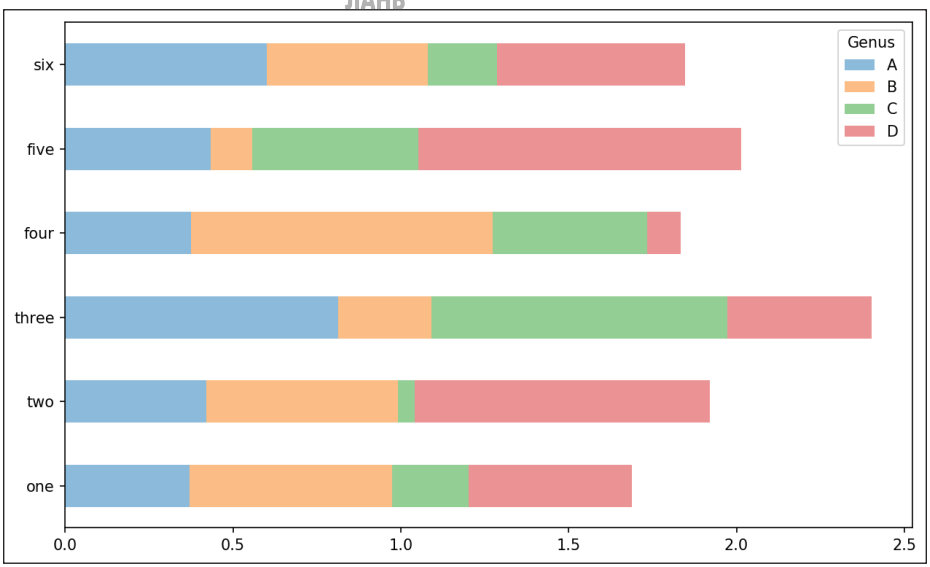


Рис. 9.17. Составная столбчатая диаграмма для DataFrame

Столбчатые диаграммы полезны для визуализации частоты значений в объекте Series с применением метода `value_counts`: `s.value_counts().plot.bar()`.

Допустим, мы хотим построить составную столбчатую диаграмму, показывающую процентную долю данных, относящихся к каждому значению количества гостей в ресторане в каждый день. Я загружаю данные методом `read_csv` и выполняю кросс-табуляцию по дню и количеству гостей:

```
In [75]: tips = pd.read_csv('examples/tips.csv')

In [76]: party_counts = pd.crosstab(tips.day, tips.size)

In [77]: party_counts
Out[77]:
size  1   2   3   4  5  6
day
Fri   1  16   1   1  0  0
Sat   2  53  18  13  1  0
Sun   0  39  15  18  3  1
Thur  1  48   4   5  1  3

# Группы, насчитывающие 1 и 6 гостей, редки
In [71]: party_counts = party_counts.ix[:, 2:5]
```

Затем нормирую значения, так чтобы сумма в каждой строке была равна 1, и строю график (рис. 9.18):

```
# Нормировка на сумму 1
In [79]: party_pcts = party_counts.div(party_counts.sum(1), axis=0)

In [80]: party_pcts
Out[80]:
size         2         3         4         5
day
Fri   0.888889  0.055556  0.055556  0.000000
Sat   0.623529  0.211765  0.152941  0.011765
Sun   0.520000  0.200000  0.240000  0.040000
Thur  0.827586  0.068966  0.086207  0.017241

In [81]: party_pcts.plot.bar()
```

Как видим, в выходные количество гостей в одной группе увеличивается.

Если перед построением графика данные необходимо как-то агрегировать, то пакет `seaborn` может существенно упростить жизнь. Посмотрим, как посчитать процент чаевых в зависимости от дня (результат показан на рис. 9.19):

```
In [83]: import seaborn as sns

In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])

In [85]: tips.head()
Out[85]:
  total_bill   tip smoker  day    time size   tip_pct
0      16.99  1.01     No  Sun  Dinner    2  0.063204
1      10.34  1.66     No  Sun  Dinner    3  0.191244
2      21.01  3.50     No  Sun  Dinner    3  0.199886
3      23.68  3.31     No  Sun  Dinner    2  0.162494
4      24.59  3.61     No  Sun  Dinner    4  0.172069

In [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

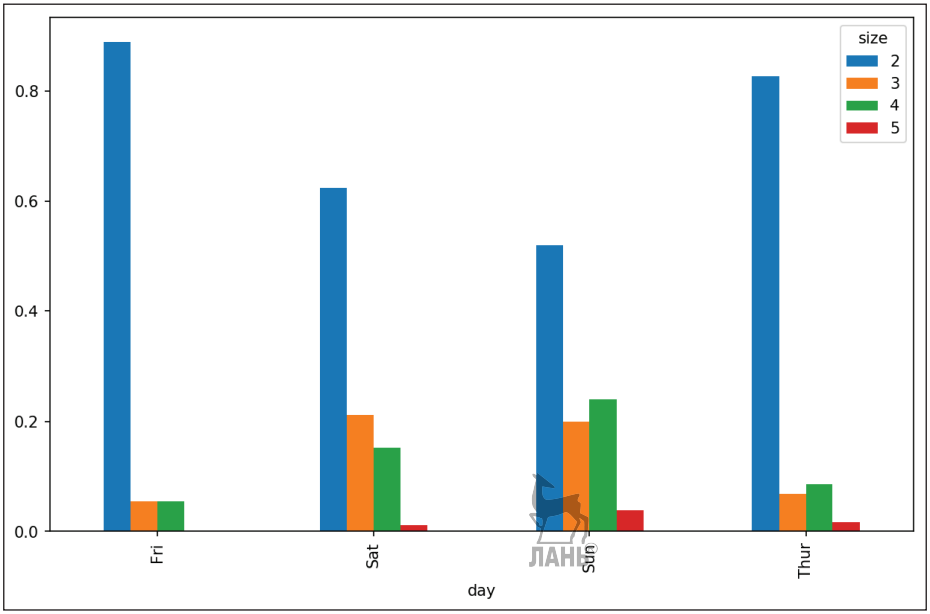


Рис. 9.18. Распределение по количеству гостей в группе в каждый день недели

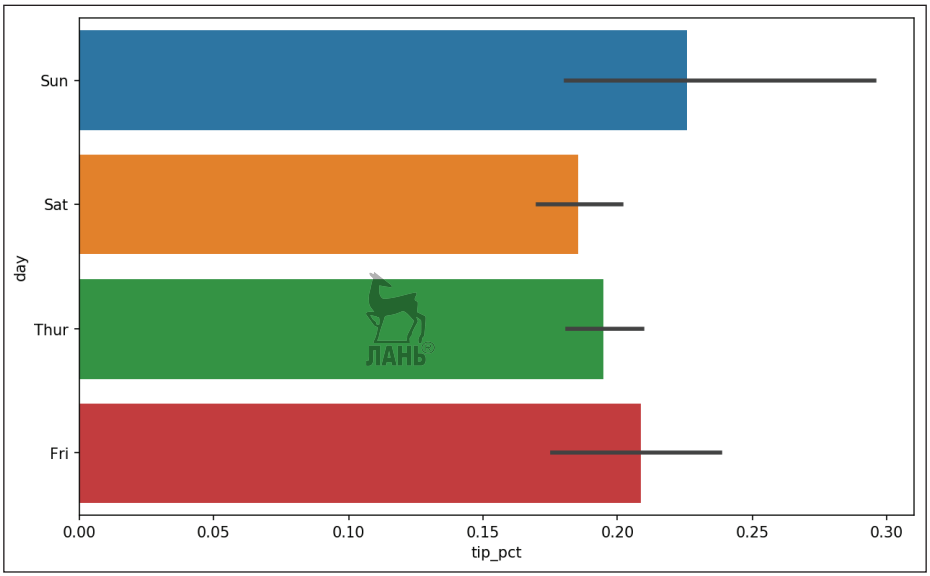


Рис. 9.19. Процент чаевых в зависимости от дня с доверительными интервалами

Функции построения графиков из библиотеки seaborn принимают аргумент `data`, в роли которого может выступать объект pandas DataFrame. Остальные аргументы относятся к именам столбцов. Поскольку для каждого значения в `day` имеется несколько наблюдений, столбики отражают среднее значение `tip_pct`. Черные линии поверх столбиков представляют 95%-ные доверительные интервалы (эту величину можно настроить, задав дополнительные аргументы).

Функция `seaborn.barplot` принимает аргумент `hue`, позволяющий произвести разбиение по дополнительному дискретному значению (рис. 9.20):

```
In [88]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

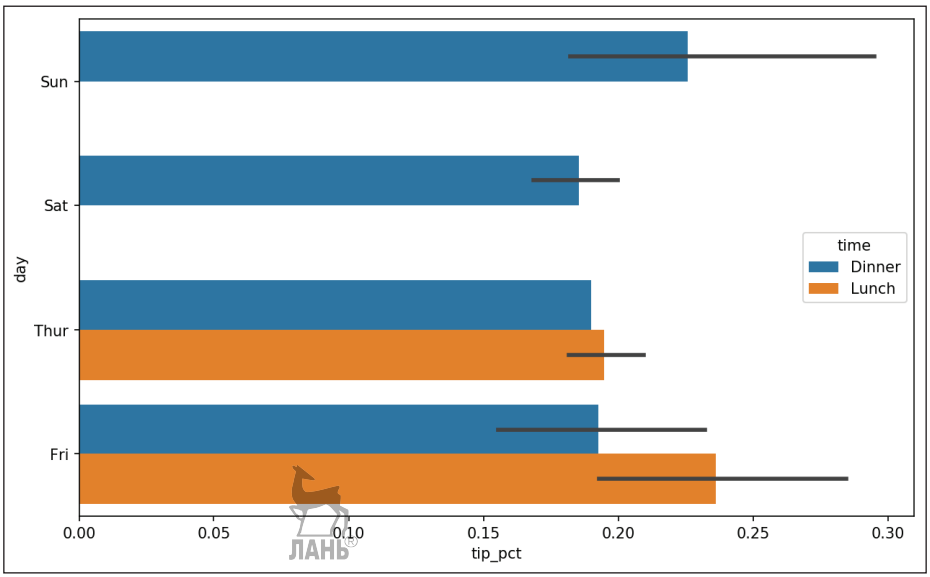


Рис. 9.20. Процент чаевых в зависимости от дня и времени суток

Обратите внимание, что seaborn автоматически изменила внешний вид диаграмм: цветовую палитру по умолчанию, цвет фона и цвета линий сетки. Менять внешний вид графиков позволяет функция `seaborn.set`:

```
In [90]: sns.set(style="whitegrid")
```

Гистограммы и графики плотности

Гистограмма, с которой все мы хорошо знакомы, – это разновидность столбчатой диаграммы, показывающая дискретизированное представление частоты. Результаты измерений распределяются по дискретным интервалам равной ширины, а на гистограмме отображается количество точек в каждом интервале. На примере приведенных выше данных о чаевых от гостей ресто-

рана мы можем с помощью метода hist объекта Series построить гистограмму распределения процента чаевых от общей суммы счета (рис. 9.21):

```
In [92]: tips['tip_pct'].plot.hist(bins=50)
```

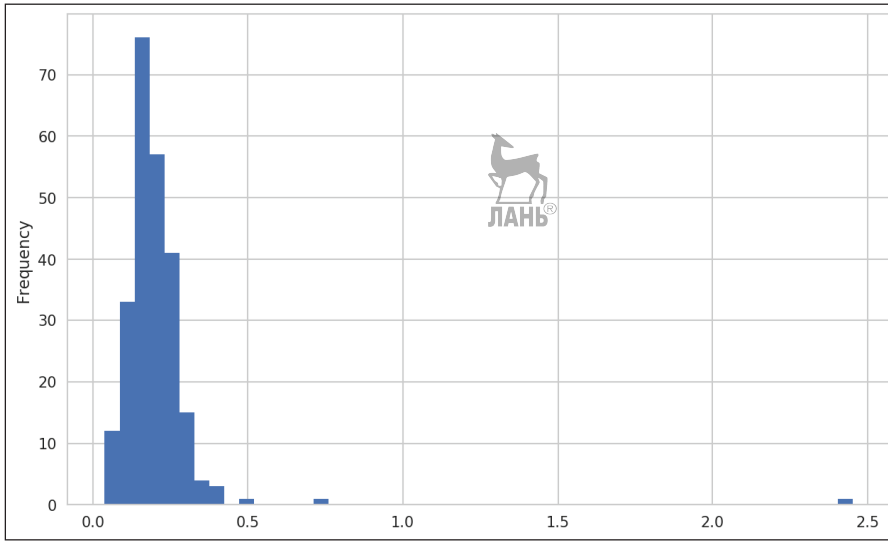


Рис. 9.21. Гистограмма процента чаевых

С гистограммой тесно связан *график плотности*, который строится на основе оценки непрерывного распределения вероятности по результатам измерений. Обычно стремятся аппроксимировать это распределение комбинацией *ядер*, т. е. более простых распределений, например нормального (гауссова). Поэтому графики плотности еще называют графиками ядерной оценки плотности (KDE – kernel density estimate). Функция plot с параметром kind='kde' строит график плотности, применяя стандартный метод комбинирования нормальных распределений (рис. 9.22):

```
In [94]: tips['tip_pct'].plot. density()
```

Seaborn еще упрощает построение гистограмм и графиков плотности благодаря методу distplot, который может строить одновременно гистограмму и непрерывную оценку плотности. В качестве примера рассмотрим бимодальное распределение, содержащее выборки из двух разных стандартных нормальных распределений (рис. 9.23):

```
In [96]: comp1 = np.random.normal(0, 1, size=200)

In [97]: comp2 = np.random.normal(10, 2, size=200)

In [98]: values = pd.Series(np.concatenate([comp1, comp2]))

In [99]: sns.distplot(values, bins=100, color='k')
```

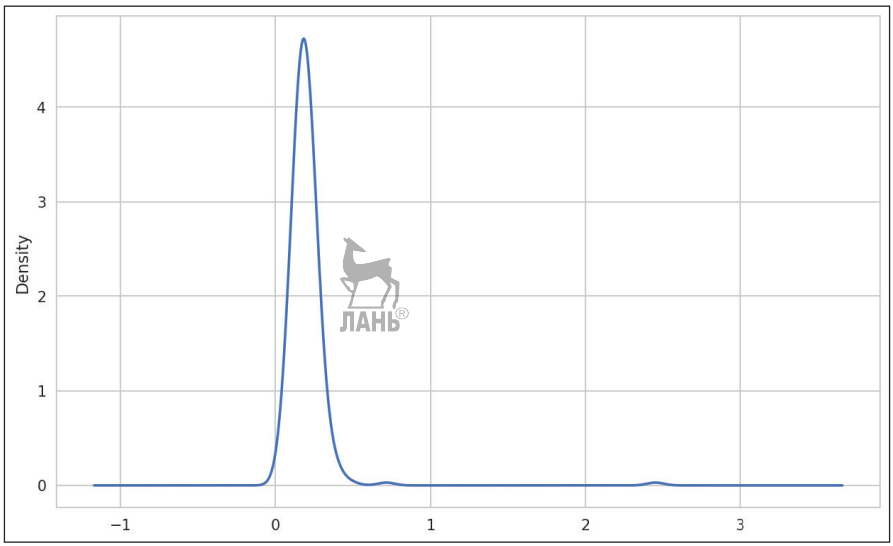


Рис. 9.22. График плотности процента чаевых

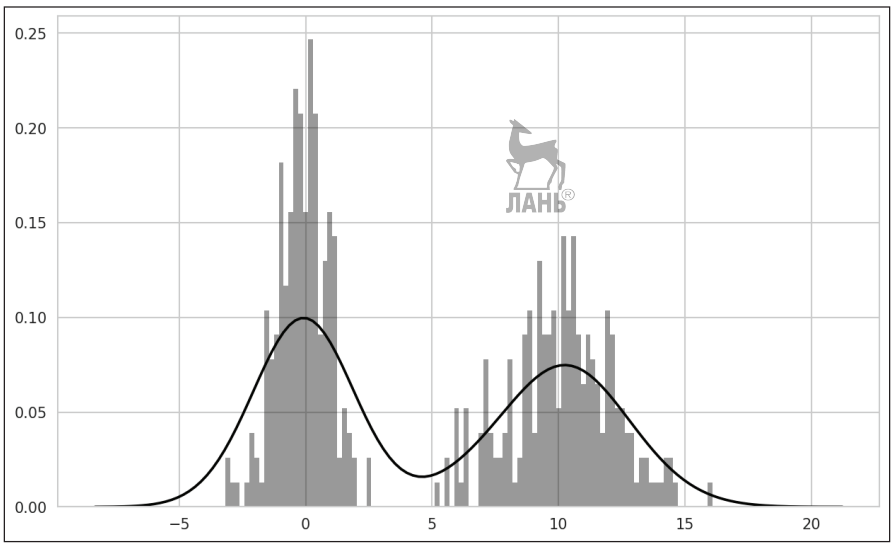


Рис. 9.23. Нормированная гистограмма
и оценка плотности смеси нормальных распределений

Диаграммы рассеяния

Диаграмма рассеяния, или точечная диаграмма, – полезный способ исследования соотношения между двумя одномерными рядами данных. Для демонстрации я загрузил набор данных `macrodata` из проекта statsmodels, выбрал несколько переменных и вычислил логарифмические разности:

```
In [100]: macro = pd.read_csv('examples/macrodata.csv')

In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

In [102]: trans_data = np.log(data).diff().dropna()

In [103]: trans_data[-5:]
Out[103]:
          cpi        m1   tbilrate     unemp
198 -0.007904  0.045361  -0.396881  0.105361
199 -0.021979  0.066753  -2.277267  0.139762
200  0.002340  0.010286   0.606136  0.160343
201  0.008419  0.037461  -0.200671  0.127339
202  0.008894  0.012202  -0.405465  0.042560
```

Затем мы можем использовать метод `regplot` из библиотеки seaborn, чтобы построить диаграмму рассеяния и аппроксимирующую ее прямую линейной регрессии (рис. 9.24):

```
In [105]: sns.regplot('m1', 'unemp', data=trans_data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb613720be0>

In [106]: plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))
```

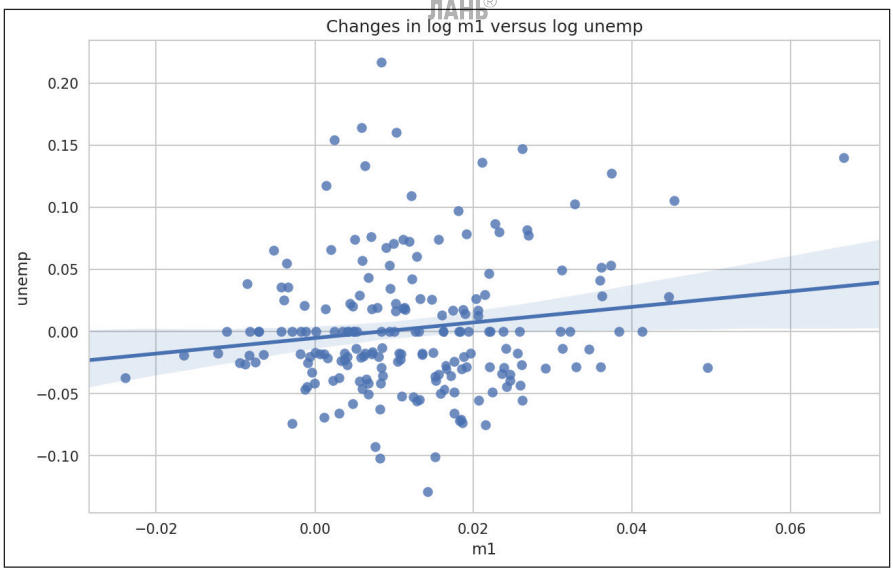


Рис. 9.24. Диаграмма рассеяния с прямой регрессии, построенная средствами seaborn

В разведочном анализе данных полезно видеть все диаграммы рассеяния для группы переменных; это называется *диаграммой пар*, или *матрицей диаграмм рассеяния*. Построение такого графика с нуля – довольно утомительное занятие, поэтому в seaborn имеется функция `pairplot`, которая поддерживает размещение гистограмм или графиков оценки плотности каждой переменной вдоль диагонали (см. результирующий график на рис. 9.25):

```
In [107]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```

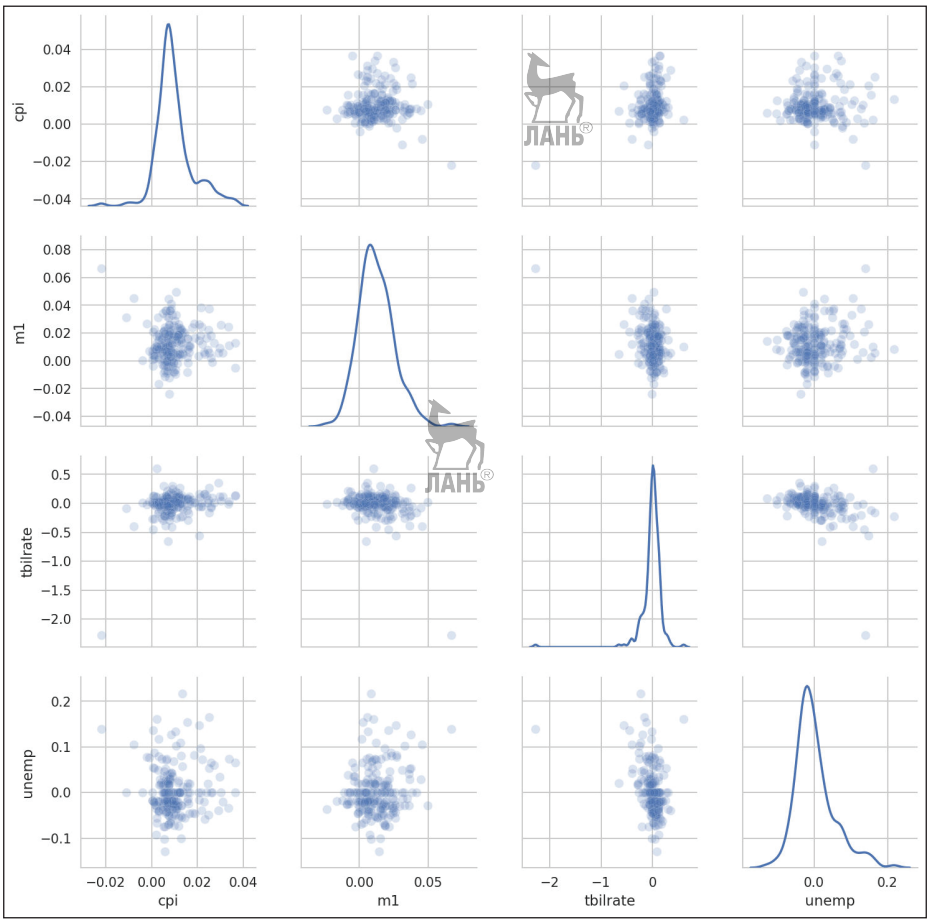


Рис. 9.25. Матрица диаграмм рассеяния
для набора данных macrodata из проекта statsmodels

Обратите внимание на аргумент `plot_kws`. Он позволяет передавать конфигурационные параметры отдельным вызовам построения во внедиагональных элементах. Дополнительные сведения о конфигурационных параметрах см. в строке документации `seaborn.pairplot`.

Фасетные сетки и категориальные данные

Как быть с наборами данных, в которых имеются дополнительные группировочные измерения? Один из способов визуализировать данные с большим числом категориальных переменных – воспользоваться *фасетной сеткой*. В seaborn имеется полезная функция `factorplot`, которая упрощает построение разнообразных фасетных графиков (рис. 9.26):

```
In [108]: sns.factorplot(x='day', y='tip_pct', hue='time', col='smoker',
   .....:                kind='bar', data=tips[tips.tip_pct < 1])
```

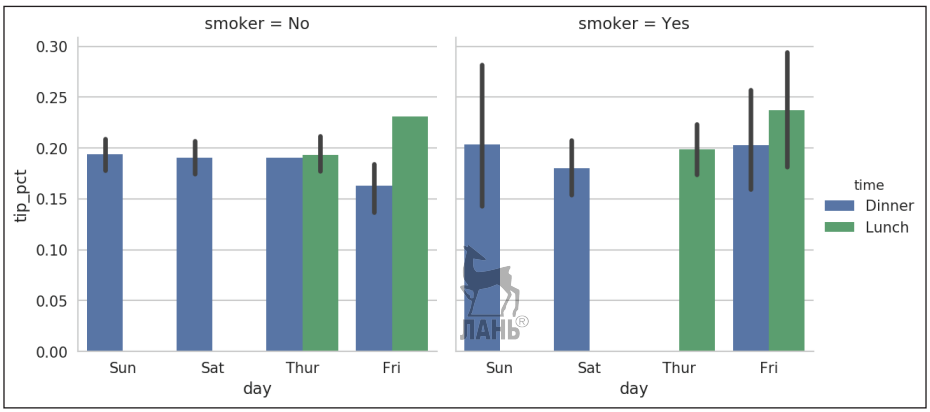


Рис. 9.26. Процент чаевых в зависимости от дня, времени суток и курения

Вместо того чтобы использовать для группировки по `'time'` столбики разных цветов внутри фасеты, мы можем расширить фасетную сетку, добавив по одной строке на каждое значение `time` (рис. 9.27):

```
In [109]: sns.factorplot(x='day', y='tip_pct', row='time',
   .....:                col='smoker',
   .....:                kind='bar', data=tips[tips.tip_pct < 1])
```

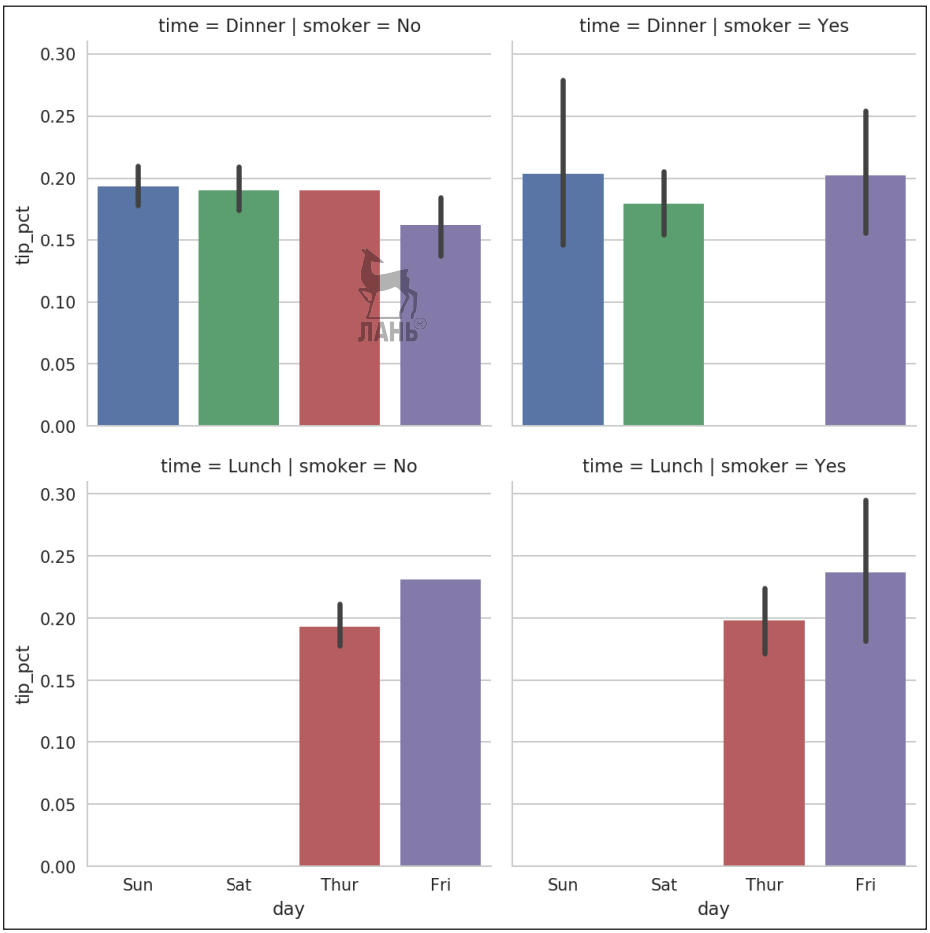


Рис. 9.27. Процент чаевых в зависимости от дня; фасеты по времени суток и курению

Функция `factorplot` поддерживает и другие типы графиков, которые могут оказаться полезны в зависимости от того, что мы пытаемся показать. Например, диаграммы размаха (на которых изображаются медиана, квартили и выбросы) могут оказаться эффективным средством визуализации (рис. 9.28):

```
In [110]: sns.factorplot(x='tip_pct', y='day', kind='box',
   .....:                data=tips[tips.tip_pct < 0.5])
```

С помощью более общего класса `seaborn.FacetGrid` можно создавать собственные фасетные сетки. Дополнительные сведения см. в документации по seaborn: https://seaborn.pydata.org/.

9.3. Другие средства визуализации для Python

Как обычно бывает в проектах с открытым исходным кодом, в средствах создания графики для Python нехватки не ощущается (их слишком много, чтобы все перечислить). Начиная с 2010 года усилия разработчиков были сосредоточены на создании интерактивной графики для публикации в вебе. Благодаря таким инструментам, как Bokeh (http://bokeh.pydata.org/) и Plotly (https://github.com/plotly/plotly.py), стало возможно описывать на Python динамичную интерактивную графику, ориентированную на отображение в браузере.

Если вы создаете статическую графику для печати или для веба, то рекомендую начать с matplotlib и таких дополнительных библиотек, как pandas и seaborn. Если же требования к визуализации иные, то полезно изучить какой-нибудь из других имеющихся в Сети инструментов. Настоятельно советую изучать экосистему, потому что она продолжает развиваться и устремлена в будущее.

9.4. Заключение

В этой главе нашей целью было познакомиться с основными средствами визуализации на основе pandas, matplotlib и seaborn. Если наглядное представление результатов анализа данных важно для вашей работы, то рекомендую почитать еще что-нибудь об эффективной визуализации данных. Это активная область исследований, поэтому вы найдете немало отличных учебных ресурсов как в Сети, так и в реальности.

В следующей главе займемся агрегированием данных и операциями группировки в pandas.

Глава 10. Агрегирование данных и групповые операции

Разбиение набора данных на группы и применение некоторой функции к каждой группе, будь то в целях агрегирования или преобразования, зачастую является одной из важнейших операций анализа данных. После загрузки, слияния и подготовки набора данных обычно вычисляют статистику по группам или, возможно, *сводные таблицы* для построения отчета или визуализации. В библиотеке pandas имеется гибкий и быстрый механизм `groupby`, который позволяет формировать продольные и поперечные срезы, а также агрегировать наборы данных естественным образом.

Одна из причин популярности реляционных баз данных и языка SQL (структурированного языка запросов) – простота соединения, фильтрации, преобразования и агрегирования данных. Однако в том, что касается групповых операций, языки запросов типа SQL несколько ограничены. Как мы увидим, выразительность и мощь языка Python и библиотеки pandas позволяют выполнять гораздо более сложные групповые операции с помощью функций, принимающих произвольный объект pandas или массив NumPy. В этой главе мы изучим следующие возможности:

- разделение объекта pandas на части по одному или нескольким ключам (в виде функций, массивов или имен столбцов объекта DataFrame);
- вычисление групповых статистик, например общего количества, среднего, стандартного отклонения или определенной пользователем функции;
- применение переменного множества функций к каждому столбцу DataFrame;
- применение внутригрупповых преобразований или иных манипуляций, например нормировки, линейной регрессии, ранжирования или выборки подмножества;

- вычисление сводных таблиц и перекрестное табулирование;
- квантильный анализ и другие виды статистического анализа групп.

Частный случай механизма `groupby`, агрегирование временных рядов, в этой книге называется *передискретизацией* и рассматривается отдельно в главе 11.

10.1. Механизм GroupBy

Хэдли Уикхэм (Hadley Wickham), автор многих популярных пакетов на языке программирования R, предложил для групповых операций термин *разделение-применение-объединение*, который, как мне кажется, удачно описывает процесс. На первом этапе данные, хранящиеся в объекте pandas, будь то Series, DataFrame или что-то еще, *разделяются* на группы по одному или нескольким указанным вами *ключам*. Разделение производится вдоль одной оси объекта. Например, DataFrame можно группировать по строкам (`axis=0`) или по столбцам (`axis=1`). Затем к каждой группе *применяется* некоторая функция, которая порождает новое значение. Наконец, результаты применения всех функций *объединяются* в результирующий объект. Форма результирующего объекта обычно зависит от того, что именно проделывается с данными. На рис. 10.1 показан схематический пример простого группового агрегирования.

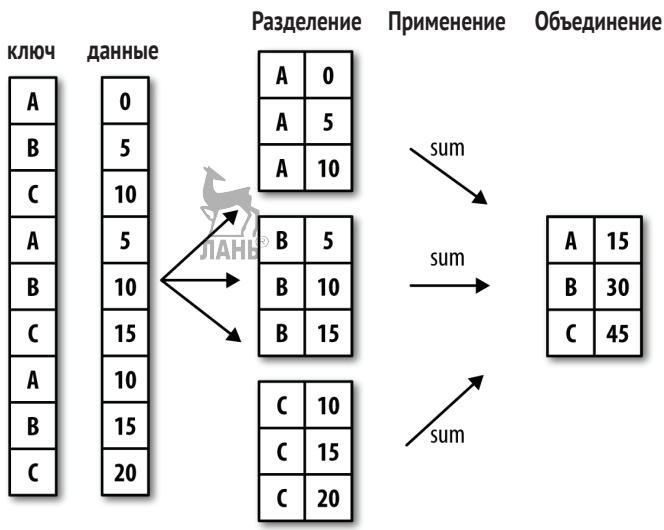


Рис. 10.1. Пример группового агрегирования

Ключи группировки могут задаваться по-разному и необязательно должны быть одного типа:

- список или массив значений той же длины, что ось, по которой производится группировка;
- значение, определяющее имя столбца объекта DataFrame;
- словарь или объект Series, определяющий соответствие между значениями на оси группировки и именами групп;
- функция, которой передается индекс оси или отдельные метки из этого индекса.

Отметим, что последние три метода – просто различные способы порождения массива значений, используемого далее для разделения объекта на группы. Не пугайтесь, если это кажется слишком абстрактным. В данной главе будут приведены многочисленные примеры каждого метода. Для начала рассмотрим очень простой табличный набор данных, представленный в виде объекта DataFrame:

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....:                    'key2' : ['one', 'two', 'one', 'two', 'one'],
   ....:                    'data1' : np.random.randn(5),
   ....:                    'data2' : np.random.randn(5)})
```

```
In [11]: df
Out[11]:
       data1     data2  key1  key2
0  -0.204708  1.393406     a   one
1   0.478943  0.092908     a   two
2  -0.519439  0.281746     b   one
3  -0.555730  0.769023     b   two
4   1.965781  1.246435     a   one
```

Пусть требуется вычислить среднее по столбцу `data1`, используя метки групп в столбце `key1`. Сделать это можно несколькими способами. Первый – взять столбец `data1` и вызвать метод `groupby`, передав ему столбец (объект Series) `key1`:

```
In [12]: grouped = df['data1'].groupby(df['key1'])
```

```
In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy at 0x7faa31537390>
```

Переменная `grouped` – это объект `GroupBy`. Пока что он не вычислил ничего, кроме промежуточных данных о групповом ключе `df['key1']`. Идея в том, что этот объект хранит всю информацию, необходимую для последующего применения некоторой операции к каждой группе. Например, чтобы вычислить средние по группам, мы можем вызвать метод `mean` объекта GroupBy:

```
In [14]: grouped.mean()
Out[14]:
         key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

Позже я подробнее объясню, что происходит при вызове .mean(). Важно, что данные (объект Series) агрегированы по групповому ключу и в результате создан новый объект Series, индексированный уникальными значениями в столбце key1. Получившийся индекс назван 'key1', потому что так именовался столбец df['key1'] объекта DataFrame.

Если бы мы передали несколько массивов в виде списка, то получили бы другой результат:

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [16]: means
Out[16]:
   key1       key2
a   one   0.880536
    two   0.478943
b   one  -0.519439
    two  -0.555730
Name: data1, dtype: float64
```

В этом случае данные сгруппированы по двум ключам, а у результирующего объекта Series имеется иерархический индекс, который состоит из уникальных пар значений ключей, встретившихся в исходных данных:

```
In [20]: means.unstack()
Out[20]:
key2     one        two
key1
a   0.880536   0.478943
b  -0.519439  -0.555730
```

В этом примере групповыми ключами были объекты Series, но можно было бы использовать и произвольные массивы правильной длины:

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005   0.478943
            2006  -0.519439
Ohio        2005  -0.380219
            2006   1.965781
Name: data1, dtype: float64
```

Часто информация о группировке находится в том же объекте DataFrame, что и группируемые данные. В таком случае в качестве групповых ключей можно передать имена столбцов (не важно, что они содержат – строки, числа или другие объекты Python):

```
In [21]: df.groupby('key1').mean()
```

```
Out[21]:
       data1     data2
key1
a   0.746672  0.910916
b  -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
           data1     data2
key1 key2
a  one  0.880536  1.319920
   two  0.478943  0.092908
b  one -0.519439  0.281746
   two -0.555730  0.769023
```

Вероятно, вы обратили внимание, что в первом случае – `df.groupby('key1').mean()` – результат не содержал столбца `key2`. Поскольку `df['key2']` содержит нечисловые данные, говорят, что это *посторонний столбец*, и в результат не включают. По умолчанию агрегируются все числовые столбцы, хотя можно выбрать и некоторое их подмножество, как мы вскоре увидим.

Вне зависимости от цели использования `groupby` у объекта GroupBy есть полезный метод `size`, который возвращает объект Series, содержащий размеры групп:

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1 key2
a   one    2
    two    1
b   one    1
    two    1
dtype: int64
```

Обратите внимание, что данные, соответствующие отсутствующим значениям группового ключа, исключаются из результата.

Обход групп

Объект GroupBy поддерживает итерирование, в результате которого генерируется последовательность 2-кортежей, содержащих имя группы и блок данных. Рассмотрим следующий небольшой набор данных:

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print name
   ....:     print group
   ....:
a
       data1     data2  key1  key2
0  -0.204708  1.393406     a   one
```

```
1   0.478943  0.092908     a   two
4   1.965781  1.246435     a   one
b
       data1     data2  key1  key2
2  -0.519439  0.281746     b   one
3  -0.555730  0.769023     b   two
```

В случае нескольких ключей первым элементом кортежа будет кортеж, содержащий значения ключей:

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print k1, k2
   ....:     print group
   ....:
('a', 'one')
       data1     data2  key1  key2
0  -0.204708  1.393406     a   one
4   1.965781  1.246435     a   one
('a', two')
       data1     data2  key1  key2
1   0.478943  0.092908     a   two
('b', 'one')
       data1     data2  key1  key2
2  -0.519439  0.281746     b   one
('b', two')
       data1     data2  key1  key2
3  -0.555730  0.769023     b   two
```

Разумеется, только вам решать, что делать с блоками данных. Возможно, пригодится следующий однострочный код, который строит словарь блоков:

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
       data1     data2  key1  key2
2  -0.519439  0.281746     b   one
3  -0.555730  0.769023     b   two
```

По умолчанию метод `groupby` группирует по оси `axis=0`, но можно задать любую другую ось. Например, в нашем примере столбцы объекта `df` можно было бы сгруппировать по `dtype`:

```
In [28]: df.dtypes
Out[28]:
data1   float64
data2   float64
key1    object
key2    object
dtype: object

In [32]: grouped = df.groupby(df.dtypes, axis=1)
```

Группы можно распечатать следующим образом:

```
In [30]: for dtype, group in grouped:
   ....:     print(dtype)
   ....:     print(group)
....:
float64
      data1     data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

Доступ по индексу к объекту GroupBy, полученному группировкой объекта DataFrame путем задания имени столбца или массива имен столбцов, имеет тот же эффект, что *выборка этих столбцов* для агрегирования. Это означает, что

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

– синтаксический сахар для:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Большие наборы данных обычно желательно агрегировать лишь по немногим столбцам. Например, чтобы в приведенном выше примере вычислить среднее только по столбцу `data2` и получить результат в виде DataFrame, можно было бы написать:

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
             data2
key1 key2
a    one  1.319920
     two  0.092908
b    one  0.281746
     two  0.769023
```

В результате этой операции доступа по индексу возвращается сгруппированный DataFrame, если передан список или массив, или сгруппированный Series, если передано только одно имя столбца:

```
In [32]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy at 0x7faa30c78da0>

In [34]: s_grouped.mean()
Out[34]:
key1 key2
a    one  1.319920
     two  0.092908
b    one  0.281746
     two  0.769023
Name: data2, dtype: float64
```

Группировка с помощью словарей и объектов Series

Информацию о группировке можно передавать не только в виде массива. Рассмотрим еще один объект DataFrame:

```
In [35]: people = DataFrame(np.random.randn(5, 5),
   ....:                    columns=['a', 'b', 'c', 'd', 'e'],
   ....:                    index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

In [36]: people.ix[2:3, ['b', 'c']] = np.nan # Добавить несколько значений NA

In [37]: people
Out[37]:
               a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Теперь предположим, что имеется соответствие между столбцами и группами и нужно просуммировать столбцы для каждой группы:

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....:            'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

Из этого словаря нетрудно построить массив и передать его groupby, но можно передать и сам словарь (я включил ключ 'f', чтобы показать, что неиспользуемые ключи группировки не составляют проблемы):

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
            blue       red
Joe     0.503905  1.063885
Steve   1.297183 -1.553778
Wes    -1.021228 -1.116829
```

```
Jim      0.524712   1.770545
Travis  -4.230992  -2.405455
```

То же самое относится к объекту Series, который можно рассматривать как отображение фиксированного размера.

```
In [41]: map_series = Series(mapping)

In [42]: map_series
Out[42]:
a     red
b     red
c    blue
d    blue
e     red
f  orange
dtype: object

In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
        blue   red
Joe        2     3
Steve      2     3
Wes        1     2
Jim        2     3
Travis     2     3
```

Группировка с помощью функций

Использование функции Python – более абстрактный способ определения соответствия групп, по сравнению со словарями или объектами Series. Функция, переданная в качестве группового ключа, будет вызвана по одному разу для каждого значения в индексе, а возвращенные ей значения станут именами групп. Конкретно рассмотрим объект DataFrame из предыдущего раздела, где значениями индекса являются имена людей. Пусть требуется сгруппировать по длине имени. Можно было бы вычислить массив длин строк, но лучше просто передать функцию len:

```
In [44]: people.groupby(len).sum()
Out[44]:
          a         b         c         d         e
3   0.591569 -0.993608  0.798764 -0.791374  2.119639
5   0.886429 -2.001637 -0.371843  1.669025 -0.438570
6  -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Использование вперемежку функций, массивов, словарей и объектов Series вполне допустимо, потому что внутри все преобразуется в массивы:

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two']

In [46]: people.groupby([len, key_list]).min()
```

```
Out[46]:
              a         b         c         d         e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
  two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Группировка по уровням индекса

Наконец, иерархически индексированные наборы данных можно агрегировать по одному из уровней индекса оси. Рассмотрим пример:

```
In [47]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
   ....:                                      [1, 3, 5, 1, 3]], names=['cty', 'tenor'])
```

```
In [48]: hier_df = DataFrame(np.random.randn(4, 5), columns=columns)
```

```
In [49]: hier_df
Out[49]:
cty          US                              JP
tenor         1          3          5          1          3
0      0.560145  -1.265934   0.119827  -1.063512   0.332883
1     -2.359419  -0.199543  -1.541996  -0.970736  -1.307030
2      0.286350   0.377984  -0.753887   0.331286   1.349742
3      0.069877   0.246674  -0.011862   1.004812   1.327195
```

Для группировки по уровню нужно передать номер или имя уровня в именованном параметре `level`:

```
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

10.2. Агрегирование данных

Под агрегированием я обычно понимаю любое преобразование данных, которое порождает скалярные значения из массивов. В примерах выше мы встречали несколько таких преобразований: `mean`, `count`, `min` и `sum`. Вероятно, вам интересно, что происходит при вызове `mean()` для объекта GroupBy. Реализации многих стандартных операций агрегирования, в частности перечисленных в табл. 10.1, оптимизированы. Однако необязательно ограничиваться только этими методами.

Вы можете придумать собственные способы агрегирования и, кроме того, вызвать любой метод, определенный для сгруппированного объекта. Напри-

мер, метод quantile вычисляет выборочные квантили для объекта Series или столбцов объекта DataFrame:

Таблица 10.1. Оптимизированные функции агрегирования

Имя функции	Описание
count	Количество отличных от NA значений в группе
sum	Сумма отличных от NA значений
mean	Среднее отличных от NA значений
median	Медиана отличных от NA значений
std, var	Несмещенное (со знаменателем n – 1) стандартное отклонение и дисперсия
min, max	Минимальное и максимальное отличные от NA значения
prod	Произведение отличных от NA значений
first, last	Первое и последнее отличные от NA значения

Несмотря на то что в классе GroupBy метод quantile не реализован, он есть в классе Series и потому может быть использован. На самом деле объект GroupBy разбивает Series на части, вызывает piece.quantile(0.9) для каждой части, а затем собирает результаты в итоговый объект.

```
In [51]: df
Out[51]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a     1.668413
b    -0.523068
Name: data1, dtype: float64
```

Для использования собственных функций агрегирования передайте функцию, агрегирующую массив, методу aggregate или agg:

```
In [54]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
       data1     data2
key1
a   2.170488  1.300498
b   0.036292  0.487276
```

Отметим, что некоторые методы, например describe, тоже работают, хотя, строго говоря, и не являются операциями агрегирования:

```
In [56]: grouped.describe()
Out[56]:
     data1                                                                \
     count      mean       std       min       25%       50%       75%
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0.478943  1.222362
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0.537585 -0.528512
             data2                                                        \
           max  count      mean       std      min       25%       50%
key1
a     1.965781    3.0  0.910916  0.712217  0.092908  0.669671  1.246435
b    -0.519439    2.0  0.525384  0.344556  0.281746  0.403565  0.525384

          75%       max
key1
a    1.319920 1.393406
b    0.647203 0.769023
```

Что здесь произошло, я объясню подробнее в разделе 10.3.

В общем случае пользовательские функции агрегирования работают гораздо медленнее оптимизированных функций из табл. 10.1. Это объясняется большими накладными расходами (на вызовы функций и реорганизацию данных) при построении промежуточных блоков данных, относящихся к каждой группе.

Применение функций, зависящих от столбца и нескольких функций

Вернемся к набору данных о чаевых, который уже встречался нам ранее. После загрузки функцией read_csv добавим в него столбец процента чаевых tip_pct:

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# Добавить величину чаевых в виде процента от суммы счета
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [59]: tips[:6]
Out[59]:
   total_bill   tip     sex  smoker  day    time   size   tip_pct
0       16.99  1.01  Female      No  Sun   Dinner     2  0.059447
1       10.34  1.66    Male      No  Sun   Dinner     3  0.160542
2       21.01  3.50    Male      No  Sun   Dinner     3  0.166587
3       23.68  3.31    Male      No  Sun   Dinner     2  0.139780
4       24.59  3.61  Female      No  Sun   Dinner     4  0.146808
5       25.29  4.71    Male      No  Sun   Dinner     4  0.186240
```

Как мы уже видели, для агрегирования объекта Series или всех столбцов объекта DataFrame достаточно воспользоваться методом `aggregate`, передав ему требуемую функцию, или вызвать метод `mean`, `std` и им подобный. Однако иногда нужно использовать разные функции в зависимости от столбца или сразу несколько функций. К счастью, сделать это совсем нетрудно, что я и продемонстрирую в следующих примерах. Для начала сгруппирую столбец `tips` по значениям `sex` и `smoker`:

```
In [60]: grouped = tips.groupby(['day', 'smoker'])
```

Отмечу, что в случае описательных статистик типа тех, что приведены в табл. 10.1, можно передать имя функции в виде строки:

```
In [61]: grouped_pct = grouped['tip_pct']

In [62]: grouped_pct.agg('mean')
Out[62]:
day  smoker
Fri  No       0.151650
     Yes      0.174783
Sat  No       0.158048
     Yes      0.147906
Sun  No       0.160113
     Yes      0.187250
Thur No       0.160298
     Yes      0.163863
Name: tip_pct, dtype: float64
```

Если вместо этого передать список функций или имен функций, то будет возвращен объект DataFrame, в котором имена столбцов совпадают с именами функций:

```
In [63]: grouped_pct.agg(['mean', 'std', peak_to_peak])
Out[63]:
                mean       std  peak_to_peak
day  smoker
Fri  No     0.151650  0.028123      0.067349
     Yes    0.174783  0.051293      0.159925
Sat  No     0.158048  0.039767      0.235193
     Yes    0.147906  0.061375      0.290095
Sun  No     0.160113  0.042347      0.193226
     Yes    0.187250  0.154134      0.644685
Thur No     0.160298  0.038774      0.193350
     Yes    0.163863  0.039389      0.151240
```

Здесь мы передали список функций агрегирования методу `agg`, который независимо вычисляет агрегаты для групп данных.

Совершенно необязательно соглашаться с именами столбцов, предложенными объектом GroupBy; в частности, все лямбда-функции называются `'<lambda>'`, поэтому различить их затруднительно (можете убедиться сами, распечатав атрибут функции `__name__`). Поэтому если передать список корте-

жей вида (name, function), то в качестве имени столбца DataFrame будет взят первый элемент кортежа (можно считать, что список 2-кортежей – упорядоченное отображение):

```
In [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[64]:
                foo       bar
day  smoker
Fri  No     0.151650  0.028123
     Yes    0.174783  0.051293
Sat  No     0.158048  0.039767
     Yes    0.147906  0.061375
Sun  No     0.160113  0.042347
     Yes    0.187250  0.154134
Thur No     0.160298  0.038774
     Yes    0.163863  0.039389
```

В случае DataFrame диапазон возможностей шире, поскольку можно задавать список функций, применяемых ко всем столбцам, или разные функции для разных столбцов. Допустим, нам нужно вычислить три одинаковые статистики для столбцов tip_pct и total_bill:

```
In [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped['tip_pct', 'total_bill'].agg(functions)

In [67]: result
Out[67]:
             tip_pct                     total_bill
               count      mean       max   count       mean    max
day  smoker
Fri  No            4  0.151650  0.187735       4  18.420000  22.75
     Yes          15  0.174783  0.263480      15  16.813333  40.17
Sat  No           45  0.158048  0.291990      45  19.661778  48.33
     Yes          42  0.147906  0.325733      42  21.276667  50.81
Sun  No           57  0.160113  0.252672      57  20.506667  48.17
     Yes          19  0.187250  0.710345      19  24.120000  45.35
Thur No           45  0.160298  0.266312      45  17.113111  41.19
    Yes           17  0.163863  0.241255      17  19.190588  43.11
```

Как видите, в результирующем DataFrame имеются иерархические столбцы – точно так же, как если бы мы агрегировали каждый столбец по отдельности, а потом склеили результаты с помощью метода concat, передав ему имена столбцов в качестве аргумента keys:

```
In [68]: result['tip_pct']
Out[68]:
            count      mean       max
day  smoker
Fri  No         4  0.151650  0.187735
     Yes       15  0.174783  0.263480
```

```
Sat  No        45  0.158048  0.291990
     Yes       42  0.147906  0.325733
Sun  No        57  0.160113  0.252672
     Yes       19  0.187250  0.710345
Thur No        45  0.160298  0.266312
     Yes       17  0.163863  0.241255
```

Как и раньше, можно передавать список кортежей, содержащий желаемые имена:

```
In [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]

In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[70]:
               tip_pct              total_bill
          Durchschnitt Abweichung Durchschnitt  Abweichung
day  smoker
Fri  No       0.151650    0.000791    18.420000   25.596333
     Yes      0.174783    0.002631    16.813333   82.562438
Sat  No       0.158048    0.001581    19.661778   79.908965
     Yes      0.147906    0.003767    21.276667  101.387535
Sun  No       0.160113    0.001793    20.506667   66.099980
     Yes      0.187250    0.023757    24.120000  109.046044
Thur No       0.160298    0.001503    17.113111   59.625081
     Yes      0.163863    0.001551    19.190588   69.808518
```

Предположим далее, что требуется применить потенциально различные функции к одному или нескольким столбцам. Делается это путем передачи методу agg словаря, который содержит отображение имен столбцов на любой из рассмотренных выше объектов, задающих функции:

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[71]:
              tip  size
day  smoker
Fri  No      3.50     9
     Yes     4.73    31
Sat  No      9.00   115
     Yes    10.00   104
Sun  No      6.00   167
     Yes     6.50    49
Thur No      6.70   112
     Yes     5.00    40

In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
   ....:              'size' : 'sum'})
Out[72]:
             tip_pct                               size
                 min       max      mean       std  sum
day  smoker
Fri  No     0.120385  0.187735  0.151650  0.028123    9
     Yes    0.103555  0.263480  0.174783  0.051293   31
```

```
Sat  No     0.056797  0.291990  0.158048  0.039767  115
     Yes    0.035638  0.325733  0.147906  0.061375  104
Sun  No     0.059447  0.252672  0.160113  0.042347  167
     Yes    0.065660  0.710345  0.187250  0.154134   49
Thur No     0.072961  0.266312  0.160298  0.038774  112
     Yes    0.090014  0.241255  0.163863  0.039389   40
```

Объект DataFrame будет содержать иерархические столбцы, только если хотя бы к одному столбцу было применено несколько функций.

Возврат агрегированных данных без индексов строк

Во всех рассмотренных выше примерах агрегированные данные сопровождались индексом, иногда иерархическим, составленным из уникальных встретившихся комбинаций групповых ключей. Такое поведение не всегда желательно, поэтому его можно подавить, передав методу `groupby` аргумент `as_index=False`:

```
In [73]: tips.groupby(['day', 'smoker'], as_index=False).mean()
Out[73]:
   day smoker  total_bill       tip      size   tip_pct
0  Fri     No   18.420000  2.812500  2.250000  0.151650
1  Fri    Yes   16.813333  2.714000  2.066667  0.174783
2  Sat     No   19.661778  3.102889  2.555556  0.158048
3  Sat    Yes   21.276667  2.875476  2.476190  0.147906
4  Sun     No   20.506667  3.167895  2.929825  0.160113
5  Sun    Yes   24.120000  3.516842  2.578947  0.187250
6  Thur    No   17.113111  2.673778  2.488889  0.160298
7  Thur   Yes   19.190588  3.030000  2.352941  0.163863
```

Разумеется, для получения данных в таком формате всегда можно вызвать метод `reset_index` результата. Аргумент `as_index=False` просто позволяет избежать некоторых лишних вычислений.

10.3. Метод apply: часть общего принципа разделения-применения-объединения

Самым общим из методов класса GroupBy является `apply`, ему мы и посвятим остаток этого раздела. На рис. 10.2 показано, что `apply` разделяет обрабатываемый объект на части, вызывает для каждой части переданную функцию, а затем пытается конкатенировать все части вместе.

Возвращаясь к набору данных о чаевых, предположим, что требуется выбрать первые пять значений `tip_pct` в каждой группе. Прежде всего нетрудно написать функцию, которая отбирает строки с наибольшими значениями в указанном столбце:

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_index(by=column)[-n:]
```

```
In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip  smoker  day    time  size   tip_pct
109       14.31  4.00     Yes  Sat  Dinner     2  0.279525
183       23.17  6.50     Yes  Sun  Dinner     4  0.280535
232       11.61  3.39      No  Sat  Dinner     2  0.291990
67         3.07  1.00     Yes  Sat  Dinner     1  0.325733
178        9.60  4.00     Yes  Sun  Dinner     2  0.416667
172        7.25  5.15     Yes  Sun  Dinner     2  0.710345
```

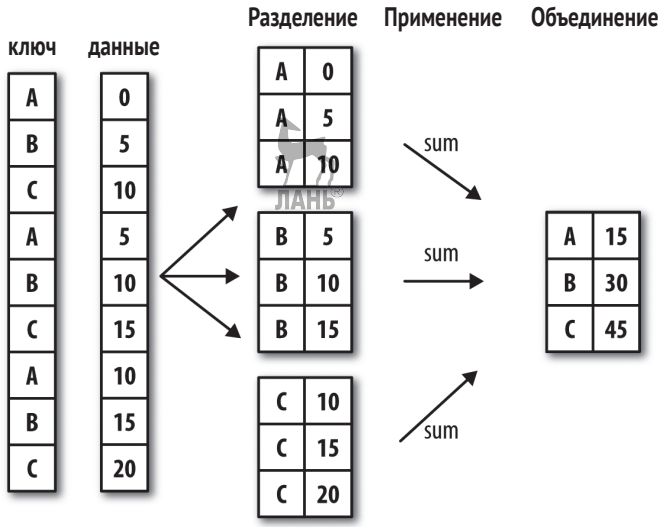


Рис. 10.2. Иллюстрация группового агрегирования

Если теперь сгруппировать, например, по столбцу `smoker` и вызвать метод `apply`, передав ему эту функцию, то получим следующее:

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip  smoker   day    time  size   tip_pct
smoker
No      88       24.71  5.85      No  Thur   Lunch     2  0.236746
        185      20.69  5.00      No   Sun  Dinner     5  0.241663
        51       10.29  2.60      No   Sun  Dinner     2  0.252672
        149       7.51  2.00      No  Thur   Lunch     2  0.266312
        232      11.61  3.39      No   Sat  Dinner     2  0.291990
Yes     109      14.31  4.00     Yes   Sat  Dinner     2  0.279525
        183      23.17  6.50     Yes   Sun  Dinner     4  0.280535
        67        3.07  1.00     Yes   Sat  Dinner     1  0.325733
        178       9.60  4.00     Yes   Sun  Dinner     2  0.416667
        172       7.25  5.15     Yes   Sun  Dinner     2  0.710345
```

Что здесь произошло? Функция top вызывается для каждой части DataFrame, после чего результаты склеиваются методом pandas.concat, а частям сопоставляются метки, совпадающие с именами групп. Поэтому результат имеет иерархический индекс, внутренний уровень которого содержит индексные значения из исходного объекта DataFrame.

Если передать методу apply функцию, которая принимает еще какие-то позиционные или именованные аргументы, то их можно передать вслед за самой функцией:

```
In [77]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
Out[77]:
                  total_bill    tip  smoker  day    time  size   tip_pct
smoker  day
No      Fri  94        22.75   3.25      No  Fri  Dinner    2  0.142857
        Sat  212       48.33   9.00      No  Sat  Dinner    4  0.186220
        Sun  156       48.17   5.00      No  Sun  Dinner    6  0.103799
       Thur  142       41.19   5.00      No Thur   Lunch    5  0.121389
Yes     Fri  95        40.17   4.73     Yes  Fri  Dinner    4  0.117750
        Sat  170       50.81  10.00     Yes  Sat  Dinner    3  0.196812
        Sun  182       45.35   3.50     Yes  Sun  Dinner    3  0.077178
       Thur  197       43.11   5.00     Yes Thur   Lunch    4  0.115982
```

Это лишь простейшие приемы, а вообще, возможности apply ограничены только вашей изобретательностью. Что именно делает переданная функция, решать вам, требуется лишь, чтобы она возвращала объект pandas или скалярное значение. Далее в этой главе будут в основном примеры, показывающие, как решать различные задачи с помощью groupby.

Вы, наверное, помните, что выше я вызывал метод describe от имени объекта GroupBy:

```
In [78]: result = tips.groupby('smoker')['tip_pct'].describe()

In [79]: result
Out[79]:
        count      mean       std       min       25%       50%       75%  \
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625  0.185014
Yes      93.0  0.163196  0.085119  0.035638  0.106771  0.153846  0.195059
            max
smoker
No     0.291990
Yes    0.710345

In [80]: result.unstack('smoker')
Out[80]:
       smoker
count  No      151.000000
       Yes      93.000000
```

```
mean   No        0.159328
       Yes       0.163196
std    No        0.039910
       Yes       0.085119
min    No        0.056797
       Yes       0.035638
25%    No        0.136906
       Yes       0.106771
50%    No        0.155625
       Yes       0.153846
75%    No        0.185014
       Yes       0.195059
max    No        0.291990
       Yes       0.710345
dtype: float64
```

Когда от имени GroupBy вызывается метод типа `describe`, на самом деле выполняются такие предложения:

```
f = lambda x: x.describe()
grouped.apply(f)
```

Подавление групповых ключей

В примерах выше мы видели, что у результирующего объекта имеется иерархический индекс, образованный групповыми ключами и индексами каждой части исходного объекта. Создание этого индекса можно подавить, передав методу `groupby` параметр `group_keys=False`:

```
In [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
     total_bill   tip  smoker  day    time  size   tip_pct
88        24.71  5.85      No Thur   Lunch     2  0.236746
185       20.69  5.00      No  Sun  Dinner     5  0.241663
51        10.29  2.60      No  Sun  Dinner     2  0.252672
149        7.51  2.00      No Thur   Lunch     2  0.266312
232       11.61  3.39      No  Sat  Dinner     2  0.291990
109       14.31  4.00     Yes  Sat  Dinner     2  0.279525
183       23.17  6.50     Yes  Sun  Dinner     4  0.280535
67         3.07  1.00     Yes  Sat  Dinner     1  0.325733
178        9.60  4.00     Yes  Sun  Dinner     2  0.416667
172        7.25  5.15     Yes  Sun  Dinner     2  0.710345
```

Квантильный и интервальный анализы

Напомним, что в главе 8 шла речь о некоторых средствах библиотеки pandas, в том числе о методах `cut` и `qcut`, которые позволяют разложить данные по ящикам, размер которых задан вами или определяется выборочными квантилями. В сочетании с методом `groupby` эти методы позволяют очень просто

подвергнуть набор данных интервальному или квантильному анализу. Рассмотрим простой набор случайных данных и раскладывание по интервалам (ящикам) равной длины с помощью cut:

```
In [82]: frame = pd.DataFrame({'data1': np.random.randn(1000),
   ....:                       'data2': np.random.randn(1000)})

In [83]: quartiles = pd.cut(frame.data1, 4)

In [84]: quartiles[:10]
Out[84]:
0     (-1.23, 0.489]
1    (-2.956, -1.23]
2     (-1.23, 0.489]
3     (0.489, 2.208]
4     (-1.23, 0.489]
5     (0.489, 2.208]
6     (-1.23, 0.489]
7     (-1.23, 0.489]
8     (0.489, 2.208]
9     (0.489, 2.208]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.
208] < (2.208, 3.928]]
```

Объект Categorical, возвращаемый функцией cut, можно передать непосредственно groupby. Следовательно, набор статистик для столбца data2 можно вычислить следующим образом:

```
In [85]: def get_stats(group):
   ....:     return {'min': group.min(), 'max': group.max(),
   ....:             'count': group.count(), 'mean': group.mean()}

In [86]: grouped = frame.data2.groupby(factor)

In [87]: grouped.apply(get_stats).unstack()
Out[87]:
                 count      max      mean       min
data1
(-2.956, -1.23]     95 1.670835 -0.039521 -3.399312
(-1.23, 0.489]     598 3.260383 -0.002051 -2.989741
(0.489, 2.208]     297 2.954439  0.081822 -3.745356
(2.208, 3.928]      10 1.765640  0.024750 -1.929776
```

Это были интервалы одинаковой длины, а чтобы вычислить интервалы равного размера на основе выборочных квантилей, нужно использовать функцию qcut. Я задам параметр labels=False, чтобы получать только номера квантилей:

```
# Вернуть номера квантилей
In [88]: grouping = pd.qcut(frame.data1, 10, labels=False)
```

```
In [89]: grouped = frame.data2.groupby(grouping)

In [90]: grouped.apply(get_stats).unstack()
Out[90]:
   count       max       mean        min
0    100  1.670835  -0.049902  -3.399312
1    100  2.628441   0.030989  -1.950098
2    100  2.527939  -0.067179  -2.925113
3    100  3.260383   0.065713  -2.315555
4    100  2.074345  -0.111653  -2.047939
5    100  2.184810   0.052130  -2.989741
6    100  2.458842  -0.021489  -2.223506
7    100  2.954439  -0.026459  -3.056990
8    100  2.735527   0.103406  -3.745356
9    100  2.377020   0.220122  -2.064111
```

В главе 12 мы поближе познакомимся с типом pandas `Categorical`.

Пример: подстановка зависящих от группы значений вместо отсутствующих

Иногда отсутствующие данные требуется отфильтровать методом `dropna`, а иногда восполнить их, подставив либо фиксированное значение, либо значение, зависящее от данных. Для этой цели предназначен метод `fillna`. Вот, например, как можно заменить отсутствующие значения средним:

```
In [91]: s = Series(np.random.randn(6))

In [92]: s[::2] = np.nan

In [93]: s
Out[93]:
0        NaN
1  -0.125921
2        NaN
3  -0.884475
4        NaN
5   0.227290
dtype: float64

In [94]: s.fillna(s.mean())
Out[94]:
0  -0.261035
1  -0.125921
2  -0.261035
3  -0.884475
4  -0.261035
5   0.227290
dtype: float64
```

А что делать, если подставляемое значение зависит от группы? Один из способов решить задачу – сгруппировать данные и вызвать метод apply, передав ему функцию, которая вызывает fillna для каждого блока данных. Ниже приведены данные о некоторых штатах США с разделением на восточные и западные:

```
In [95]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
   .....:           'Oregon', 'Nevada', 'California', 'Idaho']

In [96]: group_key = ['East'] * 4 + ['West'] * 4

In [97]: data = pd.Series(np.random.randn(8), index=states)

In [98]: data
Out[98]:
Ohio         0.922264
New York    -2.153545
Vermont     -0.365757
Florida     -0.375842
Oregon       0.329939
Nevada       0.981994
California   1.105913
Idaho       -1.613716
dtype: float64
```


Заметим, что выражение ['East'] * 4 порождает список, содержащий четыре копии элементов в ['East']. Складывание списков означает их конкатенацию.

Сделаем так, чтобы некоторые значения отсутствовали:

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [100]: data
Out[100]:
Ohio         0.922264
New York    -2.153545
Vermont           NaN
Florida     -0.375842
Oregon       0.329939
Nevada            NaN
California   1.105913
Idaho             NaN
dtype: float64

In [101]: data.groupby(group_key).mean()
Out[101]:
East  -0.535707
West   0.717926
dtype: float64
```


Чтобы подставить вместо отсутствующих значений групповые средние, нужно поступить так:

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
Ohio        0.922264
New York   -2.153545
Vermont    -0.535707
Florida    -0.375842
Oregon      0.329939
Nevada      0.717926
California  1.105913
Idaho       0.717926
dtype: float64
```

Или, возможно, требуется подставлять вместо отсутствующих значений фиксированные, но зависящие от группы:

```
In [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
Ohio        0.922264
New York   -2.153545
Vermont     0.500000
Florida    -0.375842
Oregon      0.329939
Nevada     -1.000000
California  1.105913
Idaho      -1.000000
dtype: float64
```

Пример: случайная выборка и перестановка

Предположим, что требуется произвести случайную выборку (с возвращением или без оного) из большого набора данных для моделирования методом Монте-Карло или какой-то другой задачи. Существуют разные способы выборки, одни более эффективны, другие – менее; здесь мы воспользуемся методом `sample` для объекта Series.

Для демонстрации сконструируем колоду игральных карт:

```
# Hearts (черви), Spades (пики), Clubs (трефы), Diamonds (бубны)
suits = ['H', 'S', 'C', 'D']
card_val = (range(1, 11) + [10] * 3) * 4
base_names = ['A'] + range(2, 11) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

Теперь у нас есть объект Series длины 52, индекс которого содержит названия карт, а значения – ценность карт в блэкджеке и других играх (для простоты я присвоил тузу значение 1).

```
In [108]: deck[:13]
Out[108]:
AH   1
2H   2
3H   3
4H   4
5H   5
6H   6
7H   7
8H   8
9H   9
10H 10
JH  10
KH  10
QH  10
dtype: int64
```

Исходя из сказанного выше, сдать пять карт из колоды можно следующим образом:

```
In [109]: def draw(deck, n=5):
   .....:     return deck.take(np.random.permutation(len(deck))[:n])

In [110]: draw(deck)
Out[110]:
AD   1
8C   8
5H   5
KC  10
2C   2
dtype: int64
```

Пусть требуется выбрать по две случайные карты каждой масти. Поскольку масть обозначается последним символом названия карты, то можно произвести по ней группировку и воспользоваться методом `apply`:

```
In [111]: get_suit = lambda card: card[-1] # последняя буква обозначает масть

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
C  2C   2
   3C   3
D  KD  10
   8D   8
H  KH  10
   3H   3
```

```
S  2S   2
   4S   4
dtype: int64
```

Можно поступить и по-другому:

```
In [113]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[113]:
KC  10
JC  10
AD   1
5D   5
5H   5
6H   6
7S   7
KS  10
dtype: int64
```


Пример: групповое взвешенное среднее и корреляция

Принцип разделения-применения-объединения, лежащий в основе `groupby`, позволяет легко выразить такие операции между столбцами DataFrame или двумя объектами Series, как вычисление группового взвешенного среднего. В качестве примера возьмем следующий набор данных, содержащий групповые ключи, значения и веса:


```
In [114]: df = DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
   .....:                 'data': np.random.randn(8),
   .....:                 'weights': np.random.rand(8)})
```

```
In [115]: df
Out[115]:
   category       data   weights
0         a   1.561587  0.957515
1         a   1.219984  0.347267
2         a  -0.482239  0.581362
3         a   0.315667  0.217091
4         b  -0.047852  0.894406
5         b  -0.454145  0.918564
6         b  -0.556774  0.277825
7         b   0.253321  0.955905
```

Групповое взвешенное среднее по столбцу `category` равно:

```
In [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])

In [118]: grouped.apply(get_wavg)
Out[118]:
category
a       0.811643
```

```
b      -0.122262
dtype: float64
```

В качестве другого примера рассмотрим набор данных с сайта Yahoo! Finance, содержащий цены дня на некоторые акции и индекс S&P 500 (торговый код SPX):

```
In [119]: close_px = pd.read_csv('examples/stock_px.csv', parse_dates=True,
     ....                         index_col=0)

In [120]: close_px
Out[120]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL   2214 non-null float64
MSFT   2214 non-null float64
XOM    2214 non-null float64
SPX    2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [121]: close_px[-4:]
Out[121]:
              AAPL   MSFT    XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

Было бы интересно вычислить объект DataFrame, содержащий годовые корреляции между суточной доходностью (вычисленной по процентному изменению) и SPX. Один из способов решения этой задачи состоит в том, чтобы сначала создать функцию, которая вычисляет попарную корреляцию между каждым столбцом и столбцом 'SPX':

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

Затем вычислим процентное изменение в close_px с помощью метода pct_change:

```
In [123]: rets = close_px.pct_change().dropna()
```

И наконец, сгруппируем процентные изменения по годам, которые можно извлечь из метки строки однострочной функцией, возвращающей атрибут year метки datetime:

```
In [124]: get_year = lambda x: x.year

In [125]: by_year = rets.groupby(get_year)

In [126]: by_year.apply(spx_corr)
```

```
Out[126]:
          AAPL      MSFT       XOM  SPX
2003  0.541124  0.745174  0.661265    1
2004  0.374283  0.588531  0.557742    1
2005  0.467540  0.562374  0.631010    1
2006  0.428267  0.406126  0.518514    1
2007  0.508118  0.658770  0.786264    1
2008  0.681434  0.804626  0.828303    1
2009  0.707103  0.654902  0.797921    1
2010  0.710105  0.730118  0.839057    1
2011  0.691931  0.800996  0.859975    1
```

Разумеется, ничто не мешает вычислить корреляцию между столбцами. Ниже мы вычисляем годовую корреляцию между Apple и Microsoft:

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[127]:
2003   0.480868
2004   0.259024
2005   0.300093
2006   0.161735
2007   0.417738
2008   0.611901
2009   0.432738
2010   0.571946
2011   0.581987
dtype: float64
```


Пример: групповая линейная регрессия

Следуя той же методике, что в предыдущем примере, мы можем применить `groupby` для выполнения более сложного статистического анализа на группах; главное, чтобы функция возвращала объект pandas или скалярное значение. Например, я могу определить функцию `regress` (воспользовавшись эконометрической библиотекой `statsmodels`), которая вычисляет регрессию обычным методом наименьших квадратов для каждого блока данных:

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

Теперь для вычисления линейной регрессии `AAPL` от суточного оборота `SPX` по годам нужно написать:

```
In [129]: by_year.apply(regress, 'AAPL', ['SPX'])
```

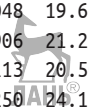
```
Out[129]:
           SPX  intercept
2003  1.195406   0.000710
2004  1.363463   0.004201
2005  1.766415   0.003246
2006  1.645496   0.000080
2007  1.198761   0.003438
2008  0.968016  -0.001110
2009  0.879103   0.002954
2010  1.052608   0.001261
2011  0.806605   0.001514
```

10.4. Сводные таблицы и перекрестное табулирование

Сводная таблица – это средство обобщения данных, применяемое в электронных таблицах и других аналитических программах. Оно агрегирует таблицу по одному или нескольким ключам и строит другую таблицу, в которой одни групповые ключи расположены в строках, а другие – в столбцах. Библиотека pandas позволяет строить сводные таблицы с помощью описанного выше механизма `groupby` в сочетании с операциями изменения формы с применением иерархического индексирования. В классе DataFrame имеется метод `pivot_table`, а на верхнем уровне – функция `pandas.pivot_table`. Помимо удобного интерфейса к `groupby` функция `pivot_table` еще умеет добавлять частичные итоги, которые называются *маргиналами*.

Вернемся к набору данных о чаевых и вычислим таблицу групповых средних (тип агрегирования по умолчанию, подразумеваемый `pivot_table`) по столбцам `day` и `smoker`, расположив их в строках:

```
In [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:
                size       tip   tip_pct total_bill
day  smoker
Fri  No     2.250000  2.812500  0.151650  18.420000
     Yes    2.066667  2.714000  0.174783  16.813333
Sat  No     2.555556  3.102889  0.158048  19.661778
     Yes    2.476190  2.875476  0.147906  21.276667
Sun  No     2.929825  3.167895  0.160113  20.506667
     Yes    2.578947  3.516842  0.187250  24.120000
Thur No     2.488889  2.673778  0.160298  17.113111
     Yes    2.352941  3.030000  0.163863  19.190588
```



Это можно было бы легко сделать и с помощью `groupby`. Пусть теперь требуется агрегировать только столбцы `tip_pct` и `size`, добавив еще группировку по `time`. Я помещу средние по `smoker` в столбцы таблицы, а по `day` – в строки:

```
In [131]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....:                  columns='smoker')
Out[131]:
                size             tip_pct
smoker             No       Yes        No       Yes
time    day
Dinner  Fri   2.000000  2.222222  0.139622  0.165347
        Sat   2.555556  2.476190  0.158048  0.147906
        Sun   2.929825  2.578947  0.160113  0.187250
        Thur  2.000000       NaN  0.159744       NaN
Lunch   Fri   3.000000  1.833333  0.187735  0.188937
        Thur  2.500000  2.352941  0.160311  0.163863
```

Эту таблицу можно было бы дополнить, включив частичные итоги, для чего следует задать параметр `margins=True`. Тогда будут добавлены строка и столбец с меткой `All`, значениями в которых будут групповые статистики по всем данным на одном уровне.

```
In [132]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....:                  columns='smoker', margins=True)
Out[132]:
                size                      tip_pct
smoker             No       Yes       All        No       Yes       All
time    day
Dinner  Fri   2.000000  2.222222  2.166667  0.139622  0.165347  0.158916
        Sat   2.555556  2.476190  2.517241  0.158048  0.147906  0.153152
        Sun   2.929825  2.578947  2.842105  0.160113  0.187250  0.166897
        Thur  2.000000       NaN  2.000000  0.159744       NaN  0.159744
Lunch   Fri   3.000000  1.833333  2.000000  0.187735  0.188937  0.188765
        Thur  2.500000  2.352941  2.459016  0.160311  0.163863  0.161301
All           2.668874  2.408602  2.569672  0.159328  0.163196  0.160803
```

Здесь столбцы `All` содержат средние без учета того, является гость курящим или некурящим, а строка `All` – средние по обоим уровням группировки.

Для применения другой функции агрегирования ее нужно передать в параметре `aggfunc`. Например, передача `'count'` или `len` даст таблицу сопряженности размеров групп (счетчики или частоты):

```
In [133]: tips.pivot_table('tip_pct', index=['time', 'smoker'], columns='day',
   .....:                  aggfunc=len, margins=True)
Out[133]:
day            Fri   Sat   Sun  Thur    All
time    smoker
Dinner  No     3.0  45.0  57.0   1.0  106.0
        Yes    9.0  42.0  19.0   NaN   70.0
Lunch   No     1.0   NaN   NaN  44.0   45.0
        Yes    6.0   NaN   NaN  17.0   23.0
All           19.0  87.0  76.0  62.0  244.0
```

Для восполнения отсутствующих комбинаций можно задать параметр `fill_value`:

```
In [134]: tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'],
   .....: columns='day', aggfunc='mean', fill_value=0)
Out[134]:
day                     Fri       Sat       Sun      Thur
time   size smoker
Dinner 1    No     0.000000  0.137931  0.000000  0.000000
            Yes    0.000000  0.325733  0.000000  0.000000
       2    No     0.139622  0.162705  0.168859  0.159744
            Yes    0.171297  0.148668  0.207893  0.000000
       3    No     0.000000  0.154661  0.152663  0.000000
            Yes    0.000000  0.144995  0.152660  0.000000
       4    No     0.000000  0.150096  0.148143  0.000000
            Yes    0.117750  0.124515  0.193370  0.000000
       5    No     0.000000  0.000000  0.206928  0.000000
            Yes    0.000000  0.106572  0.065660  0.000000
...                     ...       ...       ...       ...
Lunch  1    No     0.000000  0.000000  0.000000  0.181728
            Yes    0.223776  0.000000  0.000000  0.000000
       2    No     0.000000  0.000000  0.000000  0.166005
            Yes    0.181969  0.000000  0.000000  0.158843
       3    No     0.187735  0.000000  0.000000  0.084246
            Yes    0.000000  0.000000  0.000000  0.204952
       4    No     0.000000  0.000000  0.000000  0.138919
            Yes    0.000000  0.000000  0.000000  0.155410
       5    No     0.000000  0.000000  0.000000  0.121389
       6    No     0.000000  0.000000  0.000000  0.173706
[21 rows x 4 columns]
```

В табл. 9.2 приведена сводка аргументов метода `pivot_table`.

Таблица 9.2. Аргументы метода `pivot_table`

Параметр	Описание
`values`	Имя (или имена) одного или нескольких столбцов, по которым производится агрегирование. По умолчанию агрегируются все числовые столбцы
`index`	Имена столбцов или другие групповые ключи для группировки по строкам результирующей сводной таблицы
`columns`	Имена столбцов или другие групповые ключи для группировки по столбцам результирующей сводной таблицы
`aggfunc`	Функция агрегирования или список таких функций; по умолчанию `'mean'`. Можно задать произвольную функцию, допустимую в контексте `groupby`
`fill_value`	Чем заменять отсутствующие значения в результирующей таблице
`dropna`	Если `True`, не включать столбцы, в которых все значения отсутствуют
`margins`	Добавлять частичные итоги и общий итог по строкам и столбцам (по умолчанию `False`)

Таблицы сопряженности

Таблица сопряженности, или перекрестная таблица (cross-tabulation, или для краткости crosstab), – частный случай сводной таблицы, в которой представлены групповые частоты. Приведем пример:

```
In [138]: data
Out[138]:
  Sample Nationality    Handedness
0      1         USA  Right-handed
1      2       Japan   Left-handed
2      3         USA  Right-handed
3      4       Japan  Right-handed
4      5       Japan   Left-handed
5      6       Japan  Right-handed
6      7         USA  Right-handed
7      8         USA   Left-handed
8      9       Japan  Right-handed
9     10         USA  Right-handed
```

В ходе анализа-обследования мы могли бы обобщить эти данные по национальности и праворукости/леворукости. Для этой цели можно использовать метод `pivot_table`, но функция `pandas.crosstab` удобнее:

```
In [139]: pd.crosstab(data.Nationality, data.Handedness, margins=True)
Out[139]:
Handedness     Left-handed  Right-handed  All
Nationality
Japan                    2             3    5
USA                      1             4    5
All                      3             7   10
```

Каждый из первых двух аргументов `crosstab` может быть массивом, объектом Series или списком массивов. Например, в случае данных о чаевых:

```
In [140]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[140]:
smoker       No Yes  All
time   day
Dinner Fri    3   9   12
       Sat   45  42   87
       Sun   57  19   76
       Thur   1   0    1
Lunch  Fri    1   6    7
       Thur  44  17   61
All         151  93  244
```

10.5. Заключение

Уверенное владение средствами группировки, имеющимися в pandas, поможет вам как при очистке данных, так и в процессе моделирования или статистического анализа. В главе 14 мы рассмотрим дополнительные примеры использования `groupby` для реальных данных.

А в следующей главе обратимся к временным рядам.

Глава 11. Временные ряды

Временные ряды – важная разновидность структурированных данных. Они встречаются во многих областях, в том числе в финансах, экономике, экологии, нейронауках и физике. Любые результаты наблюдений или измерений в разные моменты времени образуют временной ряд. Для многих временных рядов характерна *фиксированная частота*, т. е. интервалы между соседними точками одинаковы – измерения производятся, например, один раз в 15 секунд, 5 минут или в месяц. Но временные ряды могут быть и *нерегулярными*, когда интервалы времени между соседними точками различаются. Как разметить временной ряд и обращаться к нему, зависит от приложения. Существуют следующие варианты:

- *временные метки*, конкретные моменты времени;
- фиксированные *периоды*, например январь 2007 года или весь 2010 год;
- временные *интервалы*, обозначаемые метками начала и конца; периоды можно считать частными случаями интервалов;
- время эксперимента или истекшее время; каждая временная метка измеряет время, прошедшее с некоторого начального момента. Например, результаты измерения диаметра печенья с момента помещения теста в духовку.

В этой главе меня будут интересовать в основном временные ряды трех первых видов, хотя многие методы применимы и к экспериментальным временным рядам, когда индекс может содержать целые или вещественные значения, обозначающие время, прошедшее с начала эксперимента. Простейший и самый распространенный вид временных рядов – ряды, индексированные временной меткой.

pandas поддерживает также индексы, построенные по приращению времени, это полезный способ представления времени эксперимента или истекшего времени. Такие индексы в книге не рассматриваются, но вы можете прочитать о них в документации (http://pandas.pydata.org/).

В библиотеке pandas имеется стандартный набор инструментов и алгоритмов для работы с временными рядами. Он позволяет эффективно работать с очень большими рядами, легко строить продольные и поперечные срезы, агрегировать и производить передискретизацию регулярных и нерегулярных временных рядов. Как нетрудно догадаться, многие из этих инструментов особенно полезны в финансовых и эконометрических приложениях, но никто не мешает применять их, например, к анализу журналов сервера.

11.1. Типы данных и инструменты, относящиеся к дате и времени

В стандартной библиотеке Python имеются типы данных для представления даты и времени, а также средства, относящиеся к календарю. Начинать изучение надо с модулей `datetime`, `time` и `calendar`. Особенно широко используется тип `datetime.datetime`, или просто `datetime`:

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 9, 25, 14, 5, 52, 72973)

In [13]: now.year, now.month, now.day
Out[13]: (2017, 9, 25)
```

В объекте типа `datetime` хранятся дата и время с точностью до микросекунды. Класс `datetime.timedelta` представляет интервал времени между двумя объектами `datetime`:

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

Можно прибавить (или вычесть) объект `timedelta` или его произведение на целое число к объекту `datetime` и получить в результате новый объект того же типа, представляющий соответственно сдвинутый момент времени:

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)

In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)
```

```
In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

Сводка типов данных в модуле `datetime` приведена в табл. 11.1. Хотя в этой главе речь пойдет преимущественно о типах данных в pandas и высокоуровневых операциях с временными рядами, вы, без сомнения, встретите основанные на `datetime` типы и во многих других приложениях, написанных на Python.

Таблица 11.1. Типы в модуле datetime

Тип	Описание
`date`	Хранит дату (год, месяц, день) по григорианскому календарю
`time`	Хранит время суток (часы, минуты, секунды и микросекунды)
`datetime`	Хранит дату и время
`timedelta`	Представляет разность между двумя значениями типа `datetime` (дни, секунды и микросекунды)
`tzinfo`	Базовый тип для хранения информации о часовых поясах

Преобразование между строкой и datetime

Объекты типа `datetime` и входящего в pandas типа `Timestamp`, с которым мы вскоре познакомимся, можно представить в виде отформатированной строки с помощью метода `str` или `strftime`, которому передается спецификация формата:

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

Полный перечень форматных кодов приведен в табл. 11.2 (повторение списка из главы 2).

Таблица 11.2. Спецификации формата даты в классе datetime (совместимы со стандартом ISO C89)

Спецификатор	Описание
`%Y`	Год с четырьмя цифрами
`%y`	Год с двумя цифрами
`%m`	Номер месяца с двумя цифрами [01, 12]
`%d`	Номер дня с двумя цифрами [01, 31]
`%H`	Час (в 24-часовом формате) [00, 23]
`%I`	Час (в 12-часовом формате) [01, 12]
`%M`	Минута с двумя цифрами [01, 59]

Таблица 11.2 (окончание)

Спецификатор	Описание
%S	Секунда [00, 61] (секунды 60 и 61 високосные)
%w	День недели в виде целого числа [0 (воскресенье), 6]
%U	Номер недели в году [00, 53]. Первым днем недели считается воскресенье, а дни, предшествующие первому воскресенью, относятся к неделе 0
%W	Номер недели в году [00, 53]. Первым днем недели считается понедельник, а дни, предшествующие первому понедельнику, относятся к неделе 0
%z	Часовой пояс UTC в виде +HHMM или –HHMM; пустая строка, если часовой пояс не учитывается
%F	Сокращение для %Y-%m-%d, например 2012-4-18
%D	Сокращение для %m/%d/%y, например 04/18/12

Многие из этих кодов используются для преобразования строк в даты методом datetime.strptime:

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]
```

Метод datetime.strptime прекрасно работает, когда формат даты известен. Однако каждый раз задавать формат даты, особенно общеупотребительный, надоедает. В таком случае можно воспользоваться методом parser.parse из стороннего пакета dateutil (он автоматически устанавливается вместе с pandas):

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

dateutil умеет разбирать практически любое представление даты, понятное человеку:

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

Если, как бывает в других локалях, день предшествует месяцу, то следует задать параметр dayfirst=True:

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

Библиотека pandas, вообще говоря, ориентирована на работу с массивами дат, используемых как в качестве осевого индекса, так и столбца в DataFrame.

Метод to_datetime разбирает различные представления даты. Стандартные форматы, например ISO8601, разбираются очень быстро.

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='dat
etime64[ns]', freq=None)
```

Кроме того, этот метод умеет обрабатывать значения, которые следует считать отсутствующими (None, пустая строка и т. д.):

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'],
dtype='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isnull(idx)
Out[38]: array([False, False, True], dtype=bool)
```

NaT (Not a Time – не время) – применяемое в pandas значение для индикации отсутствующей временной метки.

Класс dateutil.parser – полезный, но не идеальный инструмент. В частности, он распознает строки, которые не на всякий взгляд являются датами. Например, строка '42' будет разобрана как текущая календарная дата в 2042 году.

У объектов datetime имеется также ряд зависимых от локали параметров форматирования для других стран и языков. Например, сокращенные названия месяцев в системе с немецкой или французской локалью будут не такие, как в системе с английской локалью. Полный перечень см. в табл. 11.3.

Таблица 11.3. Спецификации формата даты, зависящие от локали

Спецификатор	Описание
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Полная дата и время, например «Tue 01 May 2012 04:20:57 PM»
%p	Локализованный эквивалент AM или PM
%x	Дата в формате, соответствующем локали. Например, в США 1 мая 2012 будет представлено в виде «05/01/2012»
%X	Время в формате, соответствующем локали, например «04:24:12 PM»

11.2. Основы работы с временными рядами

Самый простой вид временного ряда в pandas – объект Series, индексированный временными метками, которые часто представляются внешними по отношению к pandas Python-строками или объектами `datetime`:

```
In [39]: from datetime import datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
   ....: datetime(2011, 1, 7), datetime(2011, 1, 8),
   ....: datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [41]: ts = pd.Series(np.random.randn(6), index=dates)

In [42]: ts
Out[42]:
2011-01-02  -0.204708
2011-01-05   0.478943
2011-01-07  -0.519439
2011-01-08  -0.555730
2011-01-10   1.965781
2011-01-12   1.393406
dtype: float64
```

Под капотом объекты `datetime` помещаются в объект типа `DatetimeIndex`:

```
In [43]: ts.index
Out[43]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

Как и для других объектов Series, арифметические операции над временными рядами с различными индексами автоматически приводят к выравниванию дат:

```
In [44]: ts + ts[::2]
Out[44]:
2011-01-02   -0.409415
2011-01-05         NaN
2011-01-07   -1.038877
2011-01-08         NaN
2011-01-10    3.931561
2011-01-12         NaN
dtype: float64
```

Напомним, что конструкция `ts[::2]` выбирает каждый второй элемент `ts`.

В pandas временные метки хранятся в типе данных NumPy `datetime64` с наносекундным разрешением:

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

Скалярные значения в индексе DatetimeIndex – это объекты pandas типа Timestamp:

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: <Timestamp: 2011-01-02 00:00:00>
```

Объект Timestamp можно использовать всюду, где допустим объект datetime. Кроме того, в нем можно хранить информацию о частоте (если имеется), и он умеет преобразовывать часовые пояса и производить другие манипуляции. Подробнее об этом будет рассказано ниже.

Индексирование, выборка, подмножества

TimeSeries – подкласс Series и потому ведет себя точно так же по отношению к индексированию и выборке данных по метке:

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.51943871505673811
```

В качестве дополнительного удобства можно передать строку, допускающую интерпретацию в виде даты:

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
Out[51]: 1.9657805725027142
```

Для выборки срезов из длинных временных рядов можно передать только год или год и месяц:

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....: index=pd.date_range('1/1/2000', periods=1000))

In [53]: longer_ts
Out[53]:
2000-01-01  0.092908
2000-01-02  0.281746
2000-01-03  0.769023
2000-01-04  1.246435
2000-01-05  1.007189
2000-01-06 -1.296221
2000-01-07  0.274992
2000-01-08  0.228913
2000-01-09  1.352917
2000-01-10  0.886429
                ...
2002-09-17 -0.139298
2002-09-18 -1.159926
```

```
2002-09-19  0.618965
2002-09-20  1.373890
2002-09-21 -0.983505
2002-09-22  0.930944
2002-09-23 -0.811676
2002-09-24 -1.830156
2002-09-25 -0.138730
2002-09-26  0.334088
Freq: D, Length: 1000, dtype: float64
```

```
In [54]: longer_ts['2001']
Out[54]:
2001-01-01  1.599534
2001-01-02  0.474071
2001-01-03  0.151326
2001-01-04 -0.542173
2001-01-05 -0.475496
2001-01-06  0.106403
2001-01-07 -1.308228
2001-01-08  2.173185
2001-01-09  0.564561
2001-01-10 -0.190481
               ...
2001-12-22  0.000369
2001-12-23  0.900885
2001-12-24 -0.454869
2001-12-25 -0.864547
2001-12-26  1.129120
2001-12-27  0.057874
2001-12-28 -0.433739
2001-12-29  0.092698
2001-12-30 -1.397820
2001-12-31  1.457823
Freq: D, Length: 365, dtype: float64
```

Здесь строка '2001' интерпретируется как год, и выбирается такой период времени. Это будет работать и тогда, когда указан месяц:

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01 -0.622547
2001-05-02  0.936289
2001-05-03  0.750018
2001-05-04 -0.056715
2001-05-05  2.300675
2001-05-06  0.569497
2001-05-07  1.489410
2001-05-08  1.264250
2001-05-09 -0.761837
```

```
2001-05-10 -0.331617
               ...
2001-05-22  0.503699
2001-05-23 -1.387874
2001-05-24  0.204851
2001-05-25  0.603705
2001-05-26  0.545680
2001-05-27  0.235477
2001-05-28  0.111835
2001-05-29 -1.251504
2001-05-30 -2.949343
2001-05-31  0.634634
Freq: D, Length: 31, dtype: float64
```

Выборка срезов с помощью объектов `datetime` тоже работает:

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10  1.965781
2011-01-12  1.393406
dtype: float64
```

Поскольку временные ряды обычно упорядочены хронологически, при формировании срезов можно указывать временные метки, отсутствующие в самом ряду, т. е. выполнять запрос по диапазону:

```
In [57]: ts
Out[57]:
2011-01-02 -0.204708
2011-01-05  0.478943
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10  1.965781
2011-01-12  1.393406
dtype: float64

In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07 -0.519439
2011-01-08 -0.555730
2011-01-10  1.965781
dtype: float64
```

Как и раньше, можно задать дату в виде строки, объекта `datetime` или `Timestamp`. Напомню, что такое формирование среза порождает представление исходного временного ряда, как и для массивов NumPy. Это означает, что никакие данные не копируются, а модификация среза отражается на исходных данных.

Существует эквивалентный метод экземпляра truncate, который возвращает срез Series между двумя датами:

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02  -0.204708
2011-01-05   0.478943
2011-01-07  -0.519439
2011-01-08   -0.555730
dtype: float64
```

Все вышеперечисленное справедливо и для объекта DataFrame, индексированного по строкам:

```
In [60]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

In [61]: long_df = pd.DataFrame(np.random.randn(100, 4),
   ....:                        index=dates,
   ....:                        columns=['Colorado', 'Texas',
   ....:                                 'New York', 'Ohio'])

In [62]: long_df.loc['5-2001']
Out[62]:
            Colorado      Texas   New York       Ohio
2001-05-02  -0.006045   0.490094  -0.277186  -0.707213
2001-05-09  -0.560107   2.735527   0.927335   1.513906
2001-05-16   0.538600   1.273768   0.667876  -0.969206
2001-05-23   1.676091  -0.817649   0.050188   1.951312
2001-05-30   3.260383   0.963301   1.201206  -1.852001
```

Временные ряды с неуникальными индексами

В некоторых приложениях бывает, что несколько результатов измерений имеют одну и ту же временную метку, например:

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',
   .....:                          '1/3/2000'])

In [64]: dup_ts = Series(np.arange(5), index=dates)

In [65]: dup_ts
Out[65]:
2000-01-01   0
2000-01-02   1
2000-01-02   2
2000-01-02   3
2000-01-03   4
dtype: int64
```

Узнать о том, что индекс не уникален, можно, опросив его свойство is_unique:

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

При доступе к такому временному ряду по индексу будет возвращено либо скалярное значение, либо срез – в зависимости от того, является временная метка уникальной или нет:

```
In [67]: dup_ts['1/3/2000'] # метка уникальна
Out[67]: 4

In [68]: dup_ts['1/2/2000'] # метка повторяется
Out[68]:
2000-01-02   1
2000-01-02   2
2000-01-02   3
dtype: int64
```

Пусть требуется агрегировать данные с неуникальными временными метками. Одно из возможных решений – воспользоваться методом `groupby` с параметром `level=0`:

```
In [69]: grouped = dup_ts.groupby(level=0)

In [70]: grouped.mean()
Out[70]:
2000-01-01   0
2000-01-02   2
2000-01-03   4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01   1
2000-01-02   3
2000-01-03   1
dtype: int64
```

11.3. Диапазоны дат, частоты и сдвиг

Вообще говоря, временные ряды pandas не предполагаются регулярными, т. е. частота в них не фиксирована. Для многих приложений это вполне приемлемо. Но иногда желательно работать с постоянной частотой, например день, месяц, 15 минут, даже если для этого приходится вставлять в ряд отсутствующие значения. По счастью, pandas поддерживает полный набор частот и средства для передискретизации, выведения частот и генерации диапазонов дат с фиксированной частотой. Например, временной ряд из нашего примера можно преобразовать в ряд с частотой один день с помощью метода `resample`:

```
In [72]: ts
Out[72]:
2011-01-02  -0.204708
```

```
2011-01-05   0.478943
2011-01-07  -0.519439
2011-01-08  -0.555730
2011-01-10   1.965781
2011-01-12   1.393406
dtype: float64
```

```
In [73]: resampler = ts.resample('D')
```

Строка 'D' интерпретируется как суточная частота (daily).

Преобразование частоты, или *передискретизация*, – настолько обширная тема, что мы посвятим ей отдельный раздел – раздел 11.6 ниже. А сейчас я покажу, как работать с базовой частотой и кратными ей.

Генерация диапазонов дат

Раньше я уже пользовался методом pandas.date_range без объяснений, и вы, наверное, догадались, что он порождает объект DatetimeIndex указанной длины с определенной частотой:

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')
```

```
In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

По умолчанию метод date_range генерирует временные метки с частотой один день. Если вы передаете ему только начальную или конечную дату, то должны задать также количество генерируемых периодов:

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
```

```
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

Начальная и конечная даты определяют строгие границы для сгенерированного индекса по датам. Например, если требуется индекс по датам, содержащий последний рабочий день каждого месяца, то следует передать в качестве частоты значение `'BM'` (табл. 11.4), и тогда будут включены только даты, попадающие внутрь или на границу интервала:

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

Таблица 11.4. Базовые частоты временных рядов (список неполный)

Обозначение	Тип смещения	Описание
`D`	`Day`	Ежедневно
`B`	`BusinessDay`	Каждый рабочий день
`H`	`Hour`	Ежечасно
`T` или `min`	`Minute`	Ежеминутно
`S`	`Second`	Ежесекундно
`L` или `ms`	`Milli`	Каждую миллисекунду
`U`	`Micro`	Каждую микросекунду
`M`	`MonthEnd`	Последний календарный день месяца
`BM`	`BusinessMonthEnd`	Последний рабочий день месяца
`MS`	`MonthBegin`	Первый календарный день месяца
`BMS`	`BusinessMonthBegin`	Первый рабочий день месяца
`W-MON`, `W-TUE`, …	`Week`	Еженедельно в указанный день: `MON`, `TUE`, `WED`, `THU`, `FRI`, `SAT`, `SUN`
`WOM-1MON`, `WOM-2MON`, …	`WeekOfMonth`	Указанный день первой, второй, третьей или четвертой недели месяца. Например, `WOM-3FRI` означает третью пятницу каждого месяца
`Q-JAN`, `Q-FEB`, …	`QuarterEnd`	Ежеквартально с привязкой к последнему календарному дню каждого месяца, считая, что год заканчивается в указанном месяце: `JAN`, `FEB`, `MAR`, `APR`, `MAY`, `JUN`, `JUL`, `AUG`, `SEP`, `OCT`, `NOV`, `DEC`

Таблица 11.4 (окончание)

Обозначение	Тип смещения	Описание
BQ-JAN, BQ-FEB, …	BusinessQuarterEnd	Ежеквартально с привязкой к последнему рабочему дню каждого месяца, считая, что год заканчивается в указанном месяце
QS-JAN, QS-FEB, …	QuarterBegin	Ежеквартально с привязкой к первому календарному дню каждого месяца, считая, что год заканчивается в указанном месяце
BQS-JAN, BQS-FEB, …	BusinessQuarterBegin	Ежеквартально с привязкой к первому рабочему дню каждого месяца, считая, что год заканчивается в указанном месяце
A-JAN, A-FEB, …	YearEnd	Ежегодно с привязкой к последнему календарному дню указанного месяца: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
BA-JAN, BA-FEB, …	BusinessYearEnd	Ежегодно с привязкой к последнему рабочему дню указанного месяца
AS-JAN, AS-FEB, …	YearBegin	Ежегодно с привязкой к первому календарному дню указанного месяца
BAS-JAN, BAS-FEB, …	BusinessYearBegin	Ежегодно с привязкой к первому рабочему дню указанного месяца

По умолчанию метод date_range сохраняет время (если оно было задано) начальной и конечной временных меток:

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods=5)
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

Иногда начальная или конечная дата содержит время, но требуется сгенерировать *нормализованный* набор временных меток, в которых время совпадает с полуночью. Для этого задайте параметр normalize:

```
In [80]: pd.date_range('2012-05-02 12:56:31', periods=5, normalize=True)
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
               '2012-05-06'],
              dtype='datetime64[ns]', freq='D')
```

Частоты и смещения дат

Частота в pandas состоит из *базовой частоты* и кратности. Базовая частота обычно обозначается строкой. Например, 'M' означает раз в месяц, а 'H' – раз в час. Для каждой базовой частоты определен объект, называемый *смещением даты* (date offset). Так, частоту «раз в час» можно представить классом Hour:

```
In [81]: from pandas.tseries.offsets import Hour, Minute
```

```
In [82]: hour = Hour()

In [83]: hour
Out[83]: <Hour>
```

Для определения кратности смещения нужно задать целое число:

```
In [84]: four_hours = Hour(4)

In [85]: four_hours
Out[85]: <4 * Hours>
```

В большинстве приложений не приходится создавать такие объекты явно, достаточно использовать их строковые обозначения вида `'H'` или `'4H'`. Наличие целого числа перед базовой частотой создает кратную частоту:

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

Операция сложения позволяет объединить несколько смещений:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

Можно также задать частоту в виде строки `'1h30min'`, что приводит к тому же результату, что и выше:

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
               '2000-01-01 09:00:00', '2000-01-01 10:30:00',
               '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

Некоторые частоты описывают неравноотстоящие моменты времени. Например, значение частот `'M'` (конец календарного месяца) и `'BM'` (последний рабочий день месяца) зависит от числа дней в месяце, а в последнем случае также от того, заканчивается месяц рабочим или выходным днем. За неимением лучшего термина я называю такие смещения *привязанными*.

В табл. 11.4 перечислены имеющиеся в pandas коды частот и классы смещений дат.

Пользователь может определить собственный класс частоты для реализации логики работы с датами, отсутствующей в pandas, однако подробности этого процесса выходят за рамки книги.

Даты, связанные с неделей месяца

Полезный класс частот – «неделя месяца», обозначается строкой, начинающейся с WOM. Он позволяет получить, например, третью пятницу каждого месяца:

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

Сдвиг данных (с опережением и с запаздыванием)

Под сдвигом понимается перемещение данных назад и вперед по временной оси. У объектов Series и DataFrame имеется метод shift для наивного сдвига в обе стороны без модификации индекса:


```
In [91]: ts = pd.Series(np.random.randn(4),
   ....: index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [92]: ts
Out[92]:
2000-01-31  -0.066748
2000-02-29   0.838639
2000-03-31  -0.117388
2000-04-30  -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31        NaN
2000-02-29        NaN
2000-03-31  -0.066748
2000-04-30   0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31  -0.117388
2000-02-29  -0.517795
```

```
2000-03-31        NaN
2000-04-30        NaN
Freq: M, dtype: float64
```

При таком сдвиге в начало или в конец временного ряда вдвигаются отсутствующие данные.

Типичное применение shift – вычисление относительных изменений временного ряда или нескольких временных рядов и представление их в виде столбцов объекта DataFrame. Это выражается следующим образом:

```
ts / ts.shift(1) - 1
```

Поскольку наивный сдвиг не изменяет индекс, некоторые данные отбрасываются. Но если известна частота, то ее можно передать методу shift, чтобы сдвинуть вперед временные метки, а не сами данные:

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31  -0.066748
2000-04-30   0.838639
2000-05-31  -0.117388
2000-06-30  -0.517795
Freq: M, dtype: float64
```

Можно указывать и другие частоты, что позволяет очень гибко смещать данные в прошлое и в будущее:

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03  -0.066748
2000-03-03   0.838639
2000-04-03  -0.117388
2000-05-03  -0.517795
dtype: float64
```

```
In [97]: ts.shift(1, freq='90T')
Out[97]:
2000-01-31 01:30:00  -0.066748
2000-02-29 01:30:00   0.838639
2000-03-31 01:30:00  -0.117388
2000-04-30 01:30:00  -0.517795
Freq: M, dtype: float64
```

Здесь T обозначает минуты.

Сдвиг дат с помощью смещений

Совместно с объектами datetime и Timestamp можно использовать также смещения дат pandas:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd
```

```
In [99]: now = datetime(2011, 11, 17)
```

```
In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

В случае привязанного смещения, например MonthEnd, первое сложение с ним *продвинет* дату до следующей даты с соответствующей привязкой:

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')

In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Привязанные смещения можно использовать и для явного сдвига даты вперед и назад с помощью методов rollforward и rollback соответственно:

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

У смещений дат есть интересное применение совместно с функцией groupby:

```
In [106]: ts = pd.Series(np.random.randn(20),
   .....: index=pd.date_range('1/15/2000', periods=20, freq='4d'))

In [107]: ts
Out[107]:
2000-01-15  -0.116696
2000-01-19   2.389645
2000-01-23  -0.932454
2000-01-27  -0.229331
2000-01-31  -1.140330
2000-02-04   0.439920
2000-02-08  -0.823758
2000-02-12  -0.520930
2000-02-16   0.350282
2000-02-20   0.204395
2000-02-24   0.133445
2000-02-28   0.327905
2000-03-03   0.072153
2000-03-07   0.131678
2000-03-11  -1.297459
2000-03-15   0.997747
2000-03-19   0.870955
2000-03-23  -0.991253
2000-03-27   0.151699
2000-03-31   1.266151
Freq: 4D, dtype: float64
```

```
In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31  -0.005833
2000-02-29   0.015894
2000-03-31   0.150209
dtype: float64
```

Разумеется, проще и быстрее добиться того же результата с помощью метода `resample` (гораздо подробнее о нем будет сказано в разделе 11.6):

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31  -0.005833
2000-02-29   0.015894
2000-03-31   0.150209
Freq: M, dtype: float64
```

11.4. Часовые пояса

Работа с часовым поясами традиционно считается одной из самых неприятных сторон манипулирования временными рядами. Поэтому многие пользователи предпочитают иметь дело с временными рядами в *координированном универсальном времени* (*UTC*), которое пришло на смену гринвичскому времени и теперь является международным стандартом. Часовые пояса выражаются в виде смещений от UTC; например, в Нью-Йорке время отстает от UTC на 4 часа в летний период и на 5 часов в остальное время года.

В Python информация о часовых поясах берется из сторонней библиотеки `pytz` (ее можно установить с помощью pip или conda), которая является оберткой вокруг *базы данных Олсона*, где собраны все сведения о мировых часовых поясах. Это особенно важно для исторических данных, потому что даты перехода на летнее время (и даже смещения от UTC) многократно менялись по прихоти местных правительств. В США даты перехода на летнее время с 1900 года менялись много раз!

Подробные сведения о библиотеке `pytz` можно найти в документации к ней. Но поскольку pandas инкапсулирует функциональность `pytz`, то можете спокойно игнорировать весь ее API, кроме названий часовых поясов. А эти названия можно узнать как интерактивно, так и из документации:

```
In [110]: import pytz

In [111]: pytz.common_timezones[-5:]
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

Чтобы получить объект часового пояса от `pytz`, используется функция `pytz.timezone`:

```
In [112]: tz = pytz.timezone('America/New_York')

In [113]: tz
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Методы из библиотеки pandas принимают как названия часовых зон, так и эти объекты. Я рекомендую использовать названия.

Локализация и преобразование

По умолчанию временные ряды в pandas *не учитывают часовые пояса*. Рассмотрим следующий ряд:

```
In [114]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')

In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [116]: ts
Out[116]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64
```

Поле `tz` в индексе равно `None`:

```
In [117]: print(ts.index.tz)
None
```

Но при генерировании диапазонов дат можно и указать часовой пояс:

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

Для преобразования даты из инвариантного формата в *локализованный* служит метод `tz_localize`:

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize('UTC')

In [121]: ts_utc
```

```
Out[121]:
2012-03-09 09:30:00+00:00   -0.202469
2012-03-10 09:30:00+00:00    0.050718
2012-03-11 09:30:00+00:00    0.639869
2012-03-12 09:30:00+00:00    0.597594
2012-03-13 09:30:00+00:00   -0.797246
2012-03-14 09:30:00+00:00    0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

После локализации временного ряда для его преобразования в другой часовой пояс нужно вызвать метод `tz_convert`:

```
In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
2012-03-09 04:30:00-05:00   -0.202469
2012-03-10 04:30:00-05:00    0.050718
2012-03-11 05:30:00-04:00    0.639869
2012-03-12 05:30:00-04:00    0.597594
2012-03-13 05:30:00-04:00   -0.797246
2012-03-14 05:30:00-04:00    0.472879
Freq: D, dtype: float64
```

Приведенный выше временной ряд охватывает дату перехода на летнее время в часовом поясе `America/New_York`, мы могли бы локализовать его для часового пояса, а затем преобразовать, скажем, в UTC или в берлинское время:

```
In [124]: ts_eastern = ts.tz_localize('America/New_York')

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00   -0.202469
2012-03-10 14:30:00+00:00    0.050718
2012-03-11 13:30:00+00:00    0.639869
2012-03-12 13:30:00+00:00    0.597594
2012-03-13 13:30:00+00:00   -0.797246
2012-03-14 13:30:00+00:00    0.472879
Freq: D, dtype: float64

In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
2012-03-09 15:30:00+01:00   -0.202469
2012-03-10 15:30:00+01:00    0.050718
2012-03-11 14:30:00+01:00    0.639869
2012-03-12 14:30:00+01:00    0.597594
```

```
2012-03-13 14:30:00+01:00   -0.797246
2012-03-14 14:30:00+01:00    0.472879
Freq: D, dtype: float64
```

tz_localize и tz_convert являются также методами экземпляра DatetimeIndex:

```
In [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```

При локализации наивных временных меток проверяется также однозначность и существование моментов времени в окрестности даты перехода на летнее время.

Операции над объектами Timestamp с учетом часового пояса

По аналогии с временными рядами и диапазонами дат можно локализовать и отдельные объекты Timestamp, включив в них информацию о часовом поясе, а затем преобразовывать из одного пояса в другой:

```
In [128]: stamp = pd.Timestamp('2011-03-12 04:00')

In [129]: stamp_utc = stamp.tz_localize('utc')

In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: <Timestamp: 2011-03-11 23:00:00-0500, tz=America/New_York>
```

Часовой пояс можно задать и при создании объекта Timestamp:

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')

In [132]: stamp_moscow
Out[132]: <Timestamp: 2011-03-12 04:00:00+0300, tz=Europe/Moscow>
```

В объектах Timestamp, учитывающих часовой пояс, хранится временной штамп UTC в виде числа секунд от «эпохи» UNIX (1 января 1970 года); это значение инвариантно относительно преобразования из одного пояса в другой:

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert('America/New_York').value
Out[134]: 1299902400000000000
```

При выполнении арифметических операций над объектами pandas DateOffset всюду, где возможно, учитывается переход на летнее время. Ниже мы конструируем временные метки в моменты, предшествующие переходу на летнее время (в прямом и обратном направлениях). Сначала за 30 минут до перехода:

```
In [135]: from pandas.tseries.offsets import Hour

In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')

In [137]: stamp
Out[137]: <Timestamp: 2012-03-12 01:30:00-0400, tz=US/Eastern>

In [138]: stamp + Hour()
Out[138]: <Timestamp: 2012-03-12 02:30:00-0400, tz=US/Eastern>
```

Затем за 90 минут до перехода на летнее время:

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [140]: stamp
Out[140]: <Timestamp: 2012-11-04 00:30:00-0400, tz=US/Eastern>

In [141]: stamp + 2 * Hour()
Out[141]: <Timestamp: 2012-11-04 01:30:00-0500, tz=US/Eastern>
```

Операции между датами из разных часовых поясов

Если комбинируются два временных ряда с разными часовыми поясами, то в результате получится UTC. Поскольку во внутреннем представлении временные метки хранятся в UTC, то операция не требует никаких преобразований:

```
In [142]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')

In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [144]: ts
Out[144]:
2012-03-07 09:30:00   0.522356
2012-03-08 09:30:00  -0.546348
2012-03-09 09:30:00  -0.733537
2012-03-12 09:30:00   1.302736
2012-03-13 09:30:00   0.022199
2012-03-14 09:30:00   0.364287
2012-03-15 09:30:00  -0.922839
2012-03-16 09:30:00   0.312656
2012-03-19 09:30:00  -1.128497
2012-03-20 09:30:00  -0.333488
Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz_localize('Europe/London')

In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [147]: result = ts1 + ts2

In [148]: result.index
Out[148]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
               '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
```

```
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5. Периоды и арифметика периодов

Периоды – это промежутки времени: дни, месяцы, кварталы, годы. Этот тип данных представлен классом `Period`, конструктор которого принимает строку или число и частоту из приведенной выше таблицы:

```
In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p
Out[150]: Period('2007', 'A-DEC')
```

В данном случае объект `Period` представляет промежуток времени от 1 января 2007 года до 31 декабря 2007 года включительно. Сложение и вычитание периода и целого числа дают тот же результат, что сдвиг на величину, кратную частоте периода:

```
In [151]: p + 5
Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

Если у двух периодов одинаковая частота, то их разностью является количество единиц частоты между ними:

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

Регулярные диапазоны периодов строятся с помощью функции `period_range`:

```
In [154]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')

In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '2000-06'],
dtype='period[M]', freq='M')
```

В классе `PeriodIndex` хранится последовательность периодов, он может служить осевым индексом в любой структуре данных pandas:

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
2000-01  -0.514551
2000-02  -0.559782
2000-03  -0.783408
2000-04  -1.797685
2000-05  -0.172670
2000-06   0.680215
Freq: M, dtype: float64
```

Если имеется массив строк, то можно обратиться к самому классу PeriodIndex:

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]', freq='Q-DEC')
```

Преобразование частоты периода

Периоды и объекты PeriodIndex можно преобразовать с изменением частоты, воспользовавшись методом asfreq. Для примера предположим, что имеется годовой период, который мы хотим преобразовать в месячный, начинающийся или заканчивающийся на границе года. Это довольно просто:

```
In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p
Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')
Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')
Out[163]: Period('2007-12', 'M')
```

Можно рассматривать Period('2007', 'A-DEC') как курсор, указывающий на промежуток времени, поделенный на периоды продолжительностью один месяц. Это проиллюстрировано рис. 11.1. Для *финансового года*, заканчивающегося в любом месяце, кроме декабря, месячные подпериоды вычисляются по-другому:

```
In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p
Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')
Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')
Out[167]: Period('2007-06', 'M')
```

Когда производится преобразование из большей частоты в меньшую, объемлющий период определяется в зависимости от того, куда попадает подпериод. Например, если частота равна A-JUN, то месяц Aug-2007 фактически является частью периода 2008:

```
In [168]: p = pd.Period('2007-08', 'M')

In [169]: p.asfreq('A-JUN')
Out[169]: Period('2008', 'A-JUN')
```

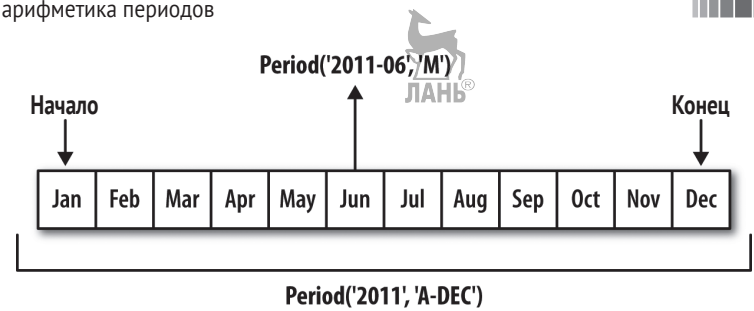


Рис. 11.1. Иллюстрация на тему преобразования частоты периода

Эта семантика сохраняется и в случае преобразования целых объектов PeriodIndex или TimeSeries:

```
In [170]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [172]: ts
Out[172]:
2006   1.607578
2007   0.200381
2008  -0.834068
2009  -0.302988
Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01   1.607578
2007-01   0.200381
2008-01  -0.834068
2009-01  -0.302988
Freq: M, dtype: float64
```

Здесь каждый годичный период заменен месячным, соответствующим первому попадающему в него месяцу. Если бы мы вместо этого захотели получить первый рабочий день каждого года, то должны были бы задать частоту 'B' и указать, что нам нужен конец периода:

```
In [174]: ts.asfreq('B', how='end')
Out[174]:
2006-12-29   1.607578
2007-12-31   0.200381
2008-12-31  -0.834068
2009-12-31  -0.302988
Freq: B, dtype: float64
```

Квартальная частота периода

Квартальные данные стандартно применяются в бухгалтерии, финансах и других областях. Обычно квартальные итоги подводятся относительно *конца финансового года*, каковым считается последний календарный или рабочий день одного из 12 месяцев. Следовательно, период `2012Q4` интерпретируется по-разному в зависимости от того, что понимать под концом финансового года. Библиотека pandas поддерживает все 12 возможных значений квартальной частоты – от `Q-JAN` до `Q-DEC`:

```
In [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p
Out[176]: Period('2012Q4', 'Q-JAN')
```

Если финансовый год заканчивается в январе, то период `2012Q4` охватывает месяцы с ноября по январь, и это легко проверить, преобразовав квартальную частоту в суточную (рис. 11.2).

Рис. 11.2. Различные соглашения о квартальной частоте

```
In [177]: p.asfreq('D', 'start')
Out[177]: Period('2011-11-01', 'D')

In [178]: p.asfreq('D', 'end')
Out[178]: Period('2012-01-31', 'D')
```

Таким образом, арифметические операции с периодами выполняются очень просто; например, чтобы получить временную метку для момента «4 часа пополудни предпоследнего рабочего дня квартала», нужно написать:

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')

In [181]: p4pm.to_timestamp()
Out[181]: <Timestamp: 2012-01-30 16:00:00>
```

Для генерации квартальных диапазонов применяется функция `period_range`. Арифметические операции тоже аналогичны:

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')

In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [184]: ts
Out[184]:
2011Q3   0
2011Q4   1
2012Q1   2
2012Q2   3
2012Q3   4
2012Q4   5
Freq: Q-JAN, dtype: int64

In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [186]: ts.index = new_rng.to_timestamp()

In [187]: ts
Out[187]:
2010-10-28 16:00:00   0
2011-01-28 16:00:00   1
2011-04-28 16:00:00   2
2011-07-28 16:00:00   3
2011-10-28 16:00:00   4
2012-01-30 16:00:00   5
dtype: int64
```

Преобразование временных меток в периоды и обратно

Объекты Series и DataFrame, индексированные временными метками, можно преобразовать в периоды методом to_period:

```
In [188]: rng = pd.date_range('2000-01-01', periods=3, freq='M')

In [189]: ts = pd.Series(np.random.randn(3), index=rng)

In [190]: ts
Out[190]:
2000-01-31   1.663261
2000-02-29  -0.996206
2000-03-31   1.521760
Freq: M, dtype: float64

In [191]: pts = ts.to_period()

In [192]: pts
Out[192]:
2000-01   1.663261
2000-02  -0.996206
2000-03   1.521760
Freq: M, dtype: float64
```

Поскольку периоды всегда ссылаются на непересекающиеся временные промежутки, то временная метка может принадлежать только одному периоду данной частоты. Можно задать любую частоту, а частота нового объекта PeriodIndex по умолчанию выводится из временных меток. Наличие повторяющихся периодов в результате также не приводит ни к каким проблемам:

```
In [193]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)

In [195]: ts2
Out[195]:
2000-01-29   0.244175
2000-01-30   0.423331
2000-01-31  -0.654040
2000-02-01   2.089154
2000-02-02  -0.060220
2000-02-03  -0.167933
Freq: D, dtype: float64

In [196]: ts2.to_period('M')
Out[196]:
2000-01   0.244175
2000-01   0.423331
2000-01  -0.654040
2000-02   2.089154
2000-02  -0.060220
2000-02  -0.167933
Freq: M, dtype: float64
```

Для обратного преобразования во временные метки служит метод to_timestamp:

```
In [197]: pts = ts2.to_period()

In [198]: pts
Out[198]:
2000-01-29   0.244175
2000-01-30   0.423331
2000-01-31  -0.654040
2000-02-01   2.089154
2000-02-02  -0.060220
2000-02-03  -0.167933
Freq: D, dtype: float64

In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29   0.244175
2000-01-30   0.423331
2000-01-31  -0.654040
2000-02-01   2.089154
```

```
2000-02-02  -0.060220
2000-02-03  -0.167933
Freq: D, dtype: float64
```

Создание PeriodIndex из массивов

В наборах данных с фиксированной частотой информация о промежутках времени иногда хранится в нескольких столбцах. Например, в следующем макроэкономическом наборе данных год и квартал находятся в разных столбцах:

```
In [200]: data = pd.read_csv('examples/macrodata.csv')
In [201]: data.head(5)
Out[201]:
        year  quarter   realgdp realcons  realinv  realgovt realdpi    cpi \
0     1959.0      1.0  2710.349   1707.4  286.898   470.045  1886.9  28.98
1     1959.0      2.0  2778.801   1733.7  310.859   481.301  1919.7  29.15
2     1959.0      3.0  2775.488   1751.8  289.226   491.260  1916.4  29.35
3     1959.0      4.0  2785.204   1753.7  299.356   484.052  1931.3  29.37
4     1960.0      1.0  2847.699   1770.5  331.722   462.199  1955.5  29.54
      m1 tbilrate  unemp      pop  infl  realint
0  139.7     2.82    5.8  177.146  0.00     0.00
1  141.7     3.08    5.1  177.830  2.34     0.74
2  140.5     3.82    5.3  178.657  2.74     1.09
3  140.0     4.33    5.6  179.386  0.27     4.06
4  139.6     3.50    5.2  180.007  2.31     1.19

In [202]: data.year
Out[202]:
0    1959.0
1    1959.0
2    1959.0
3    1959.0
4    1960.0
5    1960.0
6    1960.0
7    1960.0
8    1961.0
9    1961.0
      ...
193  2007.0
194  2007.0
195  2007.0
196  2008.0
197  2008.0
198  2008.0
199  2008.0
200  2009.0
201  2009.0
```

```
202  2009.0
Name: year, Length: 203, dtype: float64

In [203]: data.quarter
Out[203]:
0    1.0
1    2.0
2    3.0
3    4.0
4    1.0
5    2.0
6    3.0
7    4.0
8    1.0
9    2.0
     ...
193  2.0
194  3.0
195  4.0
196  1.0
197  2.0
198  3.0
199  4.0
200  1.0
201  2.0
202  3.0
Name: quarter, Length: 203, dtype: float64
```

Передав эти массивы конструктору `PeriodIndex` вместе с частотой, мы сможем объединить их для построения индекса DataFrame:

```
In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....: freq='Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [206]: data.index = index

In [207]: data.infl
Out[207]:
1959Q1   0.00
1959Q2   2.34
1959Q3   2.74
1959Q4   0.27
```

```
1960Q1   2.31
1960Q2   0.14
1960Q3   2.70
1960Q4   1.21
1961Q1  -0.40
1961Q2   1.47
         ...
2007Q2   2.75
2007Q3   3.45
2007Q4   6.38
2008Q1   2.82
2008Q2   8.53
2008Q3  -3.16
2008Q4  -8.79
2009Q1   0.94
2009Q2   3.37
2009Q3   3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6. Передискретизация и преобразование частоты

Под *передискретизацией* понимается процесс изменения частоты временного ряда. Агрегирование с переходом от высокой частоты к низкой называется *понижающей передискретизацией*, а переход от низкой частоты к более высокой – *повышающей передискретизацией*. Не всякая передискретизация попадает в одну из этих категорий; например, преобразование частоты `W-WED` (еженедельно по средам) в `W-FRI` не повышает и не понижает частоту.

Все объекты pandas имеют метод `resample`, отвечающий за любые преобразования частоты. API метода `resample` примерно такой же, как у `groupby`; мы сначала вызываем `resample` для группировки данных, а затем обращаемся к функции агрегирования:

```
In [208]: rng = pd.date_range('2000-01-01', periods=100, freq='D')

In [209]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [210]: ts
Out[210]:
2000-01-01   0.631634
2000-01-02  -1.594313
2000-01-03  -1.519937
2000-01-04   1.108752
2000-01-05   1.255853
2000-01-06  -0.024330
2000-01-07  -2.047939
2000-01-08  -0.272657
2000-01-09  -1.692615
```

```
2000-01-10   1.423830
               ...
2000-03-31  -0.007852
2000-04-01  -1.638806
2000-04-02   1.401227
2000-04-03   1.758539
2000-04-04   0.628932
2000-04-05  -0.423776
2000-04-06   0.789740
2000-04-07   0.937568
2000-04-08  -2.253294
2000-04-09  -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
2000-01-31  -0.165893
2000-02-29   0.078606
2000-03-31   0.223811
2000-04-30  -0.063643
Freq: M, dtype: float64

In [212]: ts.resample('M', kind='period').mean()
Out[212]:
2000-01  -0.165893
2000-02   0.078606
2000-03   0.223811
2000-04  -0.063643
Freq: M, dtype: float64
```

Метод `resample` обладает большой гибкостью и работает быстро, так что применим даже к очень большим временным рядам. Примеры в следующих разделах иллюстрируют его семантику и использование, в табл. 11.5 перечислены некоторые его аргументы.

Таблица 11.5. Аргументы метода `resample`

Аргумент	Описание
`freq`	Строка или объект DateOffset, задающий новую частоту, например `'M'`, `'5min'` или `Second(15)`
`axis`	Ось передискретизации, по умолчанию 0
`fill_method`	Способ интерполяции при повышающей передискретизации, например `'ffill'` или `'bfill'`. По умолчанию интерполяция не производится
`closed`	При понижающей передискретизации определяет, какой конец интервала должен включаться: `'right'` (правый) или `'left'` (левый)
`label`	При понижающей передискретизации определяет, следует ли помечать агрегированный результат меткой правого или левого конца интервала. Например, пятиминутный интервал от 9:30 до 9:35 можно пометить меткой 9:30 или 9:35. По умолчанию `'right'` (т. е. 9:35 в этом примере)

Таблица 11.5 (окончание)

Аргумент	Описание
`loffset`	Поправка для времени меток интервалов, например `'-1s'` / `Second(-1)`, чтобы сдвинуть метки агрегатов на одну секунду назад
`limit`	При прямом или обратном восполнении максимальное количество восполняемых периодов
`kind`	Агрегировать в периоды (`'period'`) или временные метки (`'timestamp'`); по умолчанию определяется видом индекса, связанного с данным временным рядом
`convention`	При передискретизации периодов соглашение (`'start'` или `'end'`) о преобразовании периода низкой частоты в период высокой частоты. По умолчанию `'end'`

Понижающая передискретизация

Агрегирование данных с целью понижения и регуляризации частоты – задача, которая часто встречается при работе с временными рядами. Частота определяет границы интервалов, разбивающих агрегируемые данные на порции. Например, для преобразования к месячному периоду `'M'` или `'BM'` данные нужно разбить на интервалы продолжительностью один месяц. Говорят, что каждый интервал *полуоткрыт*; любая точка может принадлежать только одному интервалу, а их объединение должно покрывать всю протяженность временного ряда. Перед тем как выполнять понижающую передискретизацию данных методом `resample`, нужно решить для себя следующие вопросы:

- какой конец интервала будет включаться;
- помечать ли агрегированный интервал меткой его начала или конца.

Для иллюстрации рассмотрим данные с частотой одна минута:

```
In [213]: rng = pd.date_range('1/1/2000', periods=12, freq='T')

In [214]: ts = pd.Series(np.arange(12), index=rng)

In [215]: ts
Out[215]:
2000-01-01 00:00:00   0
2000-01-01 00:01:00   1
2000-01-01 00:02:00   2
2000-01-01 00:03:00   3
2000-01-01 00:04:00   4
2000-01-01 00:05:00   5
2000-01-01 00:06:00   6
2000-01-01 00:07:00   7
2000-01-01 00:08:00   8
2000-01-01 00:09:00   9
2000-01-01 00:10:00  10
2000-01-01 00:11:00  11
Freq: T, dtype: int64
```

Пусть требуется агрегировать данные в пятиминутные группы, или *столбики*, вычислив сумму по каждой группе:

```
In [216]: ts.resample('5min', how='sum')
Out[216]:
2000-01-01 00:00:00   0
2000-01-01 00:05:00  15
2000-01-01 00:10:00  40
2000-01-01 00:15:00  11
Freq: 5T, dtype: int64
```

Переданная частота определяет границы интервалов с пятиминутным приращением. По умолчанию включается *правый* конец интервала, т. е. значение `00:05` включается в интервал от `00:00` до `00:05`[1]. Если задать параметр `closed='right'`, то будет включаться правый конец интервала:

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
1999-12-31  23:55:00   0
2000-01-01  00:00:00  15
2000-01-01  00:05:00  40
2000-01-01  00:10:00  11
Freq: 5T, dtype: int64
```

Результирующий временной ряд помечен временными метками, соответствующими левым концам интервалов. Параметр `label='right'` позволяет использовать для этой цели метки правых концов:

```
In [218]: ts.resample('5min', closed='right', label='right').sum()
Out[218]:
2000-01-01  00:00:00   0
2000-01-01  00:05:00  15
2000-01-01  00:10:00  40
2000-01-01  00:15:00  11
Freq: 5T, dtype: int64
```

На рис. 11.3 показано, как данные с минутной частотой агрегируются в пятиминутные группы.

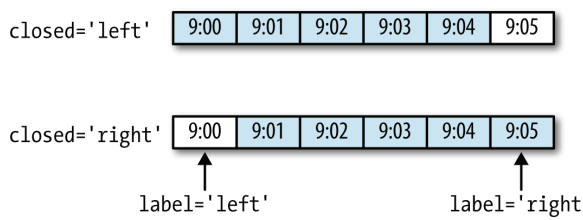


Рис. 11.3. Соглашения о включении конца и о метках интервалов на примере передискретизации с частотой 5 минут

[1] Выбор значений `closed='right'`, `label='right'` по умолчанию некоторым пользователям может показаться странным. На практике выбор более-менее произволен, для одних частот естественнее выглядит `closed='right'`, для других – `closed='left'`. Важно лишь не забывать, как именно сегментированы данные.

Наконец, иногда желательно сдвинуть индекс результата на какую-то величину, скажем, вычесть одну секунду из правого конца, чтобы было понятнее, к какому интервалу относится временная метка. Для этого следует задать строку или смещение даты в параметре `loffset`:

```
In [219]: ts.resample('5min', closed='right',
   .....: label='right', loffset='-1s').sum()
Out[219]:
1999-12-31  23:59:59   0
2000-01-01  00:04:59  15
2000-01-01  00:09:59  40
2000-01-01  00:14:59  11
Freq: 5T, dtype: int64
```

Того же результата можно достичь, вызвав метод `shift` объекта `ts` без параметра `loffset`.

Передискретизация OHLC

В финансовых приложениях очень часто временной ряд агрегируют, вычисляя четыре значения для каждого интервала: первое (открытие – open), последнее (закрытие – close), максимальное (high) и минимальное (low). Задав параметр `how='ohlc'`, мы получим объект DataFrame, в столбцах которого находятся эти четыре агрегата, которые эффективно вычисляются за один проход:

```
In [220]: ts.resample('5min').ohlc()
Out[220]:
                     open  high  low  close
2000-01-01 00:00:00     0     4    0      4
2000-01-01 00:05:00     5     9    5      9
2000-01-01 00:10:00    10    11   10     11
```

Повышающая передискретизация и интерполяция

Для преобразования от низкой частоты к более высокой агрегирование не требуется. Рассмотрим объект DataFrame, содержащий недельные данные:

```
In [221]: frame = pd.DataFrame(np.random.randn(2, 4),
   .....:                      index=pd.date_range('1/1/2000', periods=2,
   .....:                                          freq='W-WED'),
   .....:                   columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [222]: frame
Out[222]:
            Colorado    Texas New York      Ohio
2000-01-05 -0.896431 0.677263 0.036503  0.087102
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Если мы используем с этими данными функцию агрегирования, то на каждую группу получается только одно значение, а отсутствующие значения

приводят к лакунам. Чтобы перейти к более высокой частоте без агрегирования, применяется метод asfreq:

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
            Colorado    Texas New York      Ohio
2000-01-05 -0.896431 0.677263 0.036503  0.087102
2000-01-06       NaN      NaN      NaN       NaN
2000-01-07       NaN      NaN      NaN       NaN
2000-01-08       NaN      NaN      NaN       NaN
2000-01-09       NaN      NaN      NaN       NaN
2000-01-10       NaN      NaN      NaN       NaN
2000-01-11       NaN      NaN      NaN       NaN
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Допустим, мы хотим восполнить значения для дней, отличных от среды. Для этого применимы те же способы восполнения или интерполяции, что в методах fillna и reindex:

```
In [225]: frame.resample('D').ffill()
Out[225]:
            Colorado    Texas New York     Ohio
2000-01-05 -0.896431 0.677263 0.036503  0.087102
2000-01-06 -0.896431 0.677263 0.036503  0.087102
2000-01-07 -0.896431 0.677263 0.036503  0.087102
2000-01-08 -0.896431 0.677263 0.036503  0.087102
2000-01-09 -0.896431 0.677263 0.036503  0.087102
2000-01-10 -0.896431 0.677263 0.036503  0.087102
2000-01-11 -0.896431 0.677263 0.036503  0.087102
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Можно восполнить отсутствующие значения не во всех последующих периодах, а только в заданном числе:

```
In [226]: frame.resample('D').ffill(limit=2)
Out[226]:
            Colorado    Texas New York      Ohio
2000-01-05 -0.896431 0.677263 0.036503  0.087102
2000-01-06 -0.896431 0.677263 0.036503  0.087102
2000-01-07 -0.896431 0.677263 0.036503  0.087102
2000-01-08       NaN      NaN      NaN       NaN
2000-01-09       NaN      NaN      NaN       NaN
2000-01-10       NaN      NaN      NaN       NaN
2000-01-11       NaN      NaN      NaN       NaN
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

Важно отметить, что новый индекс дат может вообще не пересекаться со старым:

```
In [227]: frame.resample('W-THU').ffill()
Out[227]:
           Colorado     Texas New York      Ohio
2000-01-06 -0.896431 0.677263 0.036503  0.087102
2000-01-13 -0.046662 0.927238 0.482284 -0.867130
```

Передискретизация периодов

Передискретизация данных, индексированных периодами, похожа на индексирование временными метками:

```
In [228]: frame = pd.DataFrame(np.random.randn(24, 4),
   .....:                      index=pd.period_range('1-2000', '12-2001',
   .....:                                            freq='M'),
   .....:                      columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [229]: frame[:5]
Out[229]:
         Colorado     Texas  New York      Ohio
2000-01  0.493841 -0.155434  1.397286  1.507055
2000-02 -1.179442  0.443171  1.395676 -0.529658
2000-03  0.787358  0.248845  0.743239  1.267746
2000-04  1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816  0.426419  0.312945 -1.115689

In [230]: annual_frame = frame.resample('A-DEC').mean()

In [231]: annual_frame
Out[231]:
      Colorado     Texas  New York      Ohio
2000  0.556703  0.016631  0.111873 -0.027445
2001  0.046303  0.163344  0.251503 -0.157276
```

Повышающая передискретизация чуть сложнее, потому что необходимо принять решение о том, в какой конец промежутка времени для новой частоты помещать значения до передискретизации, как в случае метода `asfreq`. Аргумент `convention` по умолчанию равен `'end'`, но можно задать и значение `'start'`:

```
# Q-DEC: поквартально, год заканчивается в декабре
In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
        Colorado     Texas  New York      Ohio
2000Q1  0.556703  0.016631  0.111873 -0.027445
2000Q2  0.556703  0.016631  0.111873 -0.027445
2000Q3  0.556703  0.016631  0.111873 -0.027445
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.046303  0.163344  0.251503 -0.157276
2001Q2  0.046303  0.163344  0.251503 -0.157276
2001Q3  0.046303  0.163344  0.251503 -0.157276
2001Q4  0.046303  0.163344  0.251503 -0.157276
```

```
In [233]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[233]:
        Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
```

Поскольку периоды ссылаются на промежутки времени, правила повышающей и понижающей передискретизации более строгие:

- в случае понижающей передискретизации конечная частота должна быть *подпериодом* начальной;
- в случае повышающей передискретизации конечная частота должна быть *надпериодом* начальной.

Если эти правила не выполнены, то будет возбуждено исключение. Это относится главным образом к квартальной, годовой и недельной частоте; например, промежутки времени, определенные частотой Q-MAR, выровнены только с периодами A-MAR, A-JUN, A-SEP и A-DEC:

```
In [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
        Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
2002Q1  0.046303  0.163344  0.251503 -0.157276
2002Q2  0.046303  0.163344  0.251503 -0.157276
2002Q3  0.046303  0.163344  0.251503 -0.157276
```

11.7. Скользящие оконные функции

Важный класс преобразований массива, применяемый для операций с временными рядами, – статистические и иные функции, вычисляемые в скользящем окне или с экспоненциально убывающими весами. Я называю их *скользящими оконными функциями*, хотя сюда относятся также функции, не связанные с окном постоянной ширины, например экспоненциально взвешенное скользящее среднее. Как и во всех статистических функциях, отсутствующие значения автоматически отбрасываются.

Для начала загрузим временной ряд и передискретизируем его на частоту «рабочий день»:

```
In [235]: close_px_all = pd.read_csv('examples/stock_px_2.csv',
   .....: parse_dates=True, index_col=0)
```

```
In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close_px = close_px.resample('B').ffill()
```

Теперь я введу в рассмотрение оператор `rolling`, который ведет себя как `resample` и `groupby`. Его можно применить к объекту Series или DataFrame, передав аргумент `window` (равный количеству периодов; созданный график показан на рис. 11.4):

```
In [238]: close_px.AAPL.plot()

Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f2570cf98>
In [239]: close_px.AAPL.rolling(250).mean().plot()
```

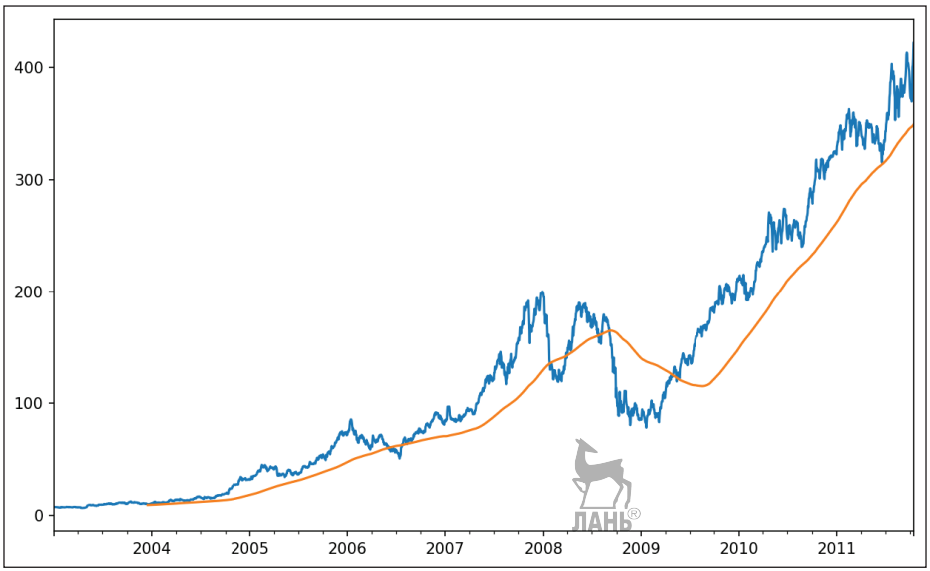


Рис. 11.4. Скользящее среднее котировок акций Apple за 250 дней

Выражение `rolling(250)` ведет себя как `groupby`, но вместо группировки создает объект, который допускает группировку по скользящему окну шириной 250 дней. Таким образом, здесь мы имеем средние котировки акций Apple в скользящем окне шириной 250 дней.

По умолчанию функции создания скользящих окон требуют, чтобы все значения в окне были отличны от NA. Это поведение можно изменить, чтобы учесть возможность отсутствия данных и, в частности, тот факт, что в начале временного ряда количество периодов данных меньше `window` (см. рис. 11.5):

```
In [241]: appl_std250 = close_px.AAPL.rolling(250, min_periods=10).std()

In [242]: appl_std250[5:12]
```

```
Out[242]:
2003-01-09       NaN
2003-01-10       NaN
2003-01-13       NaN
2003-01-14       NaN
2003-01-15  0.077496
2003-01-16  0.074760
2003-01-17  0.112368
Freq: B, Name: AAPL, dtype: float64
```

```
In [243]: appl_std250.plot()
```

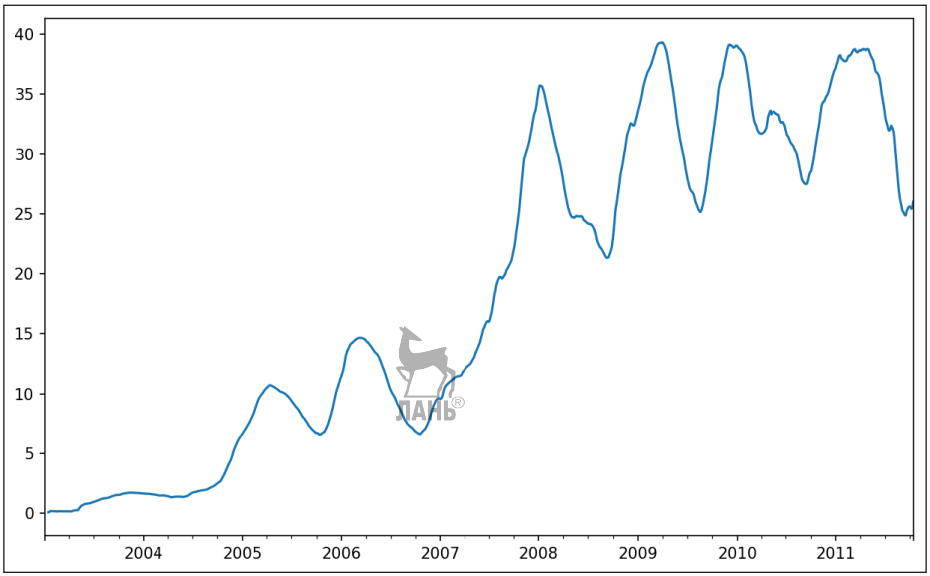


Рис. 11.5. Стандартное отклонение суточного оборота Apple

Чтобы вычислить *среднее с расширяющимся окном*, используйте оператор `expanding` вместо `rolling`. В этом случае начальное окно расположено в начале временного ряда и увеличивается в размере, пока не охватит весь ряд. Среднее с расширяющимся окном для временного ряда `apple_std250` вычисляется следующим образом:

```
In [244]: expanding_mean = appl_std250.expanding().mean()
```

При вызове функции скользящего окна от имени объекта DataFrame преобразование применяется к каждому столбцу (см. рис. 11.6):

```
In [246]: close_px.rolling(60).mean().plot(logy=True)
```

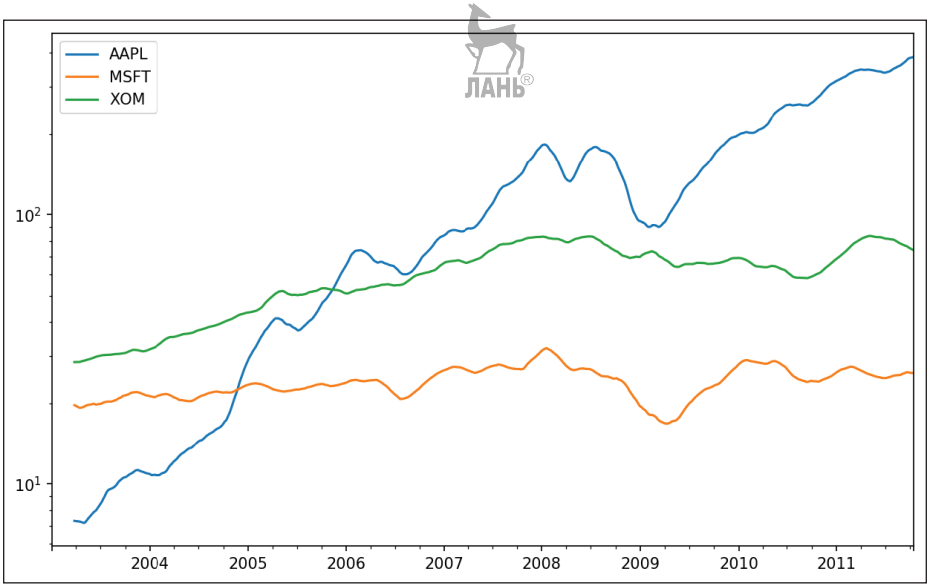


Рис. 11.6. Скользящее среднее котировок акций за 60 дней (по оси y отложен логарифм)

Функция `rolling` принимает также строку, содержащую фиксированное временное смещение, а не количество периодов. Такой вариант может быть полезен для нерегулярных временных рядов. Точно такие же строки передаются `resample`. Например, вот как можно было бы вычислить скользящее среднее за 20 дней:

```
In [247]: close_px.rolling('20D').mean()
Out[247]:
                 AAPL       MSFT        XOM
2003-01-02    7.400000  21.110000  29.220000
2003-01-03    7.425000  21.125000  29.230000
2003-01-06    7.433333  21.256667  29.473333
2003-01-07    7.432500  21.425000  29.342500
2003-01-08    7.402000  21.402000  29.240000
2003-01-09    7.391667  21.490000  29.273333
2003-01-10    7.387143  21.558571  29.238571
2003-01-13    7.378750  21.633750  29.197500
2003-01-14    7.370000  21.717778  29.194444
2003-01-15    7.355000  21.757000  29.152000
...                ...        ...        ...
2011-10-03  398.002143  25.890714  72.413571
2011-10-04  396.802143  25.807857  72.427143
```

```
2011-10-05  395.751429  25.729286  72.422857
2011-10-06  394.099286  25.673571  72.375714
2011-10-07  392.479333  25.712000  72.454667
2011-10-10  389.351429  25.602143  72.527857
2011-10-11  388.505000  25.674286  72.835000
2011-10-12  388.531429  25.810000  73.400714
2011-10-13  388.826429  25.961429  73.905000
2011-10-14  391.038000  26.048667  74.185333
[2292 rows x 3 columns]
```

Экспоненциально взвешенные функции

Вместо использования окна постоянного размера, когда веса всех наблюдений одинаковы, можно задать постоянный *коэффициент затухания*, чтобы повысить вес последних наблюдений. Есть два способа задать коэффициент затухания. Самый популярный – использовать *промежуток* (span), потому что результаты в этом случае получаются сравнимыми с применением простой скользящей оконной функции, для которой размер окна равен промежутку.

Поскольку экспоненциально взвешенная статистика придает больший вес недавним наблюдениям, она быстрее адаптируется к изменениям по сравнению с вариантом с равными весами.

В pandas имеется оператор `ewm`, который работает совместно с `rolling` и `expanding`. В примере ниже скользящее среднее котировок акций Apple за 60 дней сравнивается с экспоненциально взвешенным скользящим средним для `span=60` (рис. 11.7):

```
In [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [251]: ewma60 = aapl_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [254]: plt.legend()
```

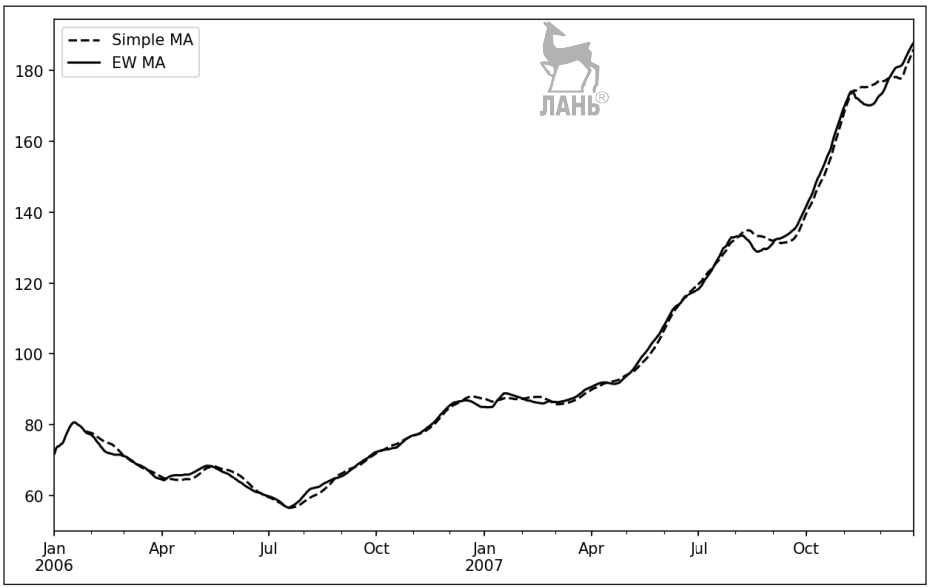


Рис. 11.7. Простое и экспоненциально взвешенное скользящее среднее

Бинарные скользящие оконные функции

Для некоторых статистических операций, в частности корреляции и ковариации, необходимы два временных ряда. Например, финансовых аналитиков часто интересует корреляция цены акции с основным биржевым индексом типа S&P 500. Чтобы найти эту величину, мы сначала вычислим относительные изменения в процентах для всего нашего временного ряда:

```
In [256]: spx_px = close_px_all['SPX']

In [257]: spx_rets = spx_px.pct_change()

In [258]: returns = close_px.pct_change()
```

Если теперь вызвать `rolling`, а за ней функцию агрегирования `corr`, то мы сможем вычислить скользящую корреляцию по `spx_rets` (получающийся график показан на рис. 11.8):

```
In [259]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_rets)

In [260]: corr.plot()
```

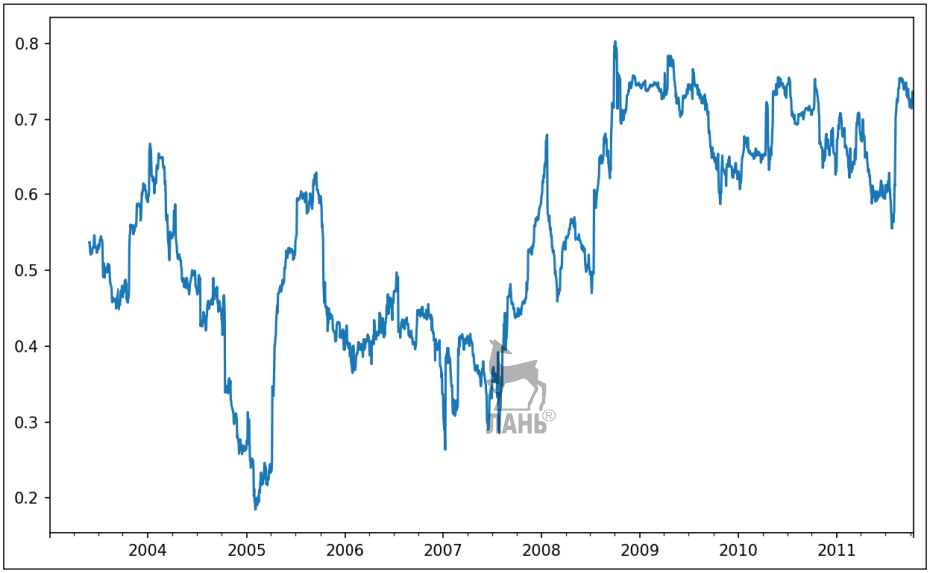


Рис. 11.8. Корреляция доходности AAPL с индексом S&P 500, рассчитанная за шесть месяцев

Пусть требуется вычислить корреляцию индекса S&P 500 сразу с несколькими акциями. Писать каждый раз цикл и создавать новый объект DataFrame, конечно, нетрудно, но уж больно скучно, поэтому если передать функции `rolling_corr` (или ей подобной) объекты TimeSeries и DataFrame, то она вычислит корреляцию TimeSeries (в данном случае `spx_rets`) с каждым столбцом DataFrame. Результат показан на рис. 11.9.

```
In [262]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)

In [263]: corr.plot()
```

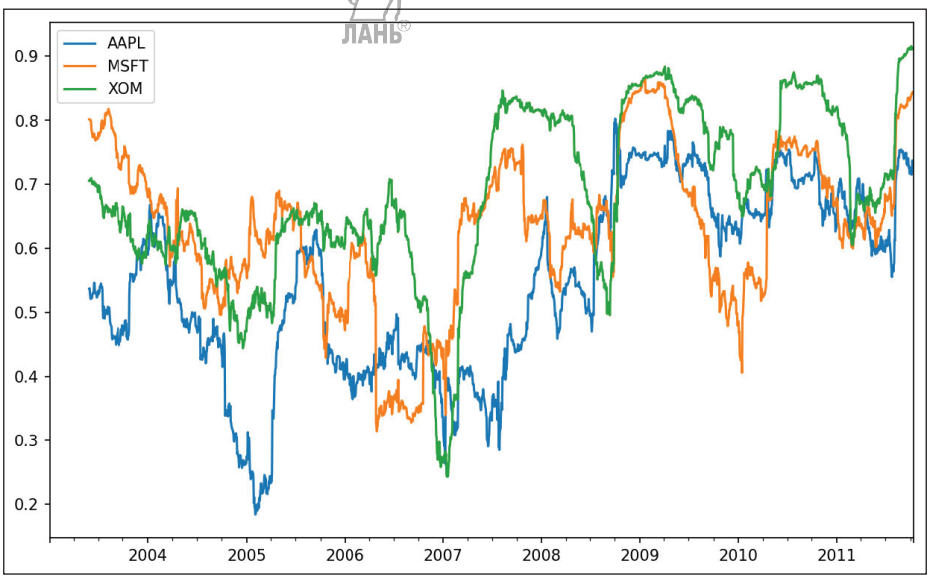


Рис. 11.9. Корреляция доходности нескольких акций с индексом S&P 500, рассчитанная за шесть месяцев

Скользящие оконные функции, определенные пользователем

Метод `apply` скользящего окна `rolling` и других подобных объектов позволяет применить произвольную функцию, принимающую массив, к скользящему окну. Единственное требование заключается в том, что функция должна порождать только одно скалярное значение (производить редукцию) для каждого фрагмента массива. Например, при вычислении выборочных квантилей с помощью `rolling(...).quantile(q)` нам может быть интересен процентильный ранг некоторого значения относительно выборки. Это можно сделать с помощью функции `scipy.stats.percentileofscore` (график показан на рис. 11.10):

```
In [265]: from scipy.stats import percentileofscore

In [266]: score_at_2percent = lambda x: percentileofscore(x, 0.02)

In [267]: result = returns.AAPL.rolling(250).apply(score_at_2percent)

In [268]: result.plot()
```

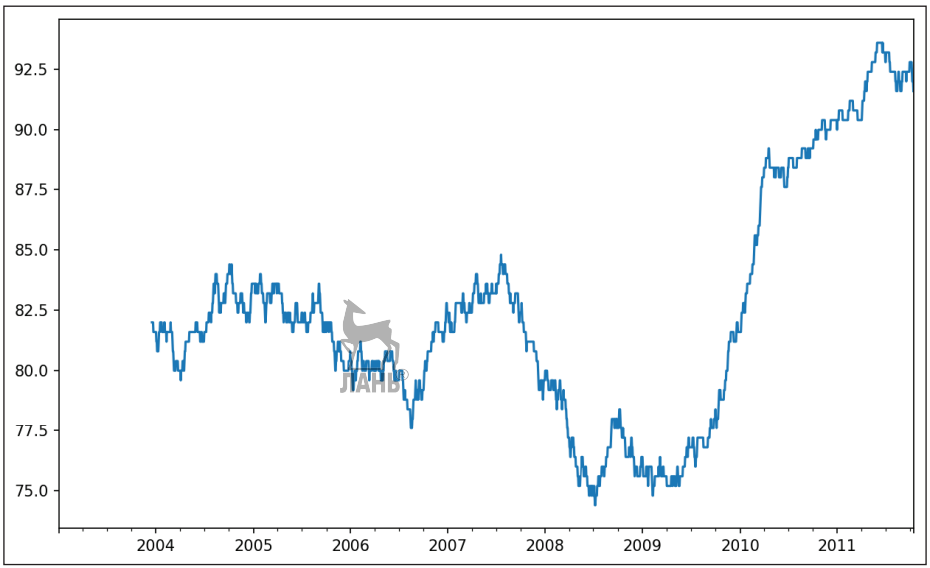


Рис. 11.10. 2%-ный процентильный ранг доходности AAPL, рассчитанный по окну протяженностью 1 год

Установить SciPy можно с помощью conda или pip.

11.8. Заключение

Для анализа временных рядов нужны специальные средства преобразования данных, отличающиеся от рассмотренных в предыдущих главах.

В следующих главах мы перейдем к продвинутым средствам pandas и покажем, как приступить к работе с библиотеками моделирования типа statsmodels и scikit-learn.

Глава 12. Дополнительные сведения о библиотеке NumPy

В предыдущих главах мы познакомились с различными способами применения переформатирования данных, а также со средствами NumPy, pandas и других библиотек. Со временем pandas обогатилась многочисленными инструментами для опытных пользователей. В этой главе мы разберем некоторые продвинутые возможности, освоив которые, вы станете лучше владеть pandas.

12.1. Категориальные данные

В этом разделе вводится тип pandas `Categorical`. Я покажу, как с его помощью повысить производительность и эффективность использования памяти в некоторых операциях. Кроме того, мы познакомимся со способами применения категориальных данных в статистике и машинном обучении.

Для чего это нужно

Часто бывает, что столбец таблицы содержит повторяющиеся значения из небольшого дискретного множества. Мы уже встречались с функциями `unique` и `value_counts`, которые позволяют найти различающиеся значения в массиве и подсчитать их частоты соответственно:

```
In [10]: import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'apple',
   ....: 'apple'] * 2)

In [12]: values
Out[12]:
0    apple
```

```
1   orange
2    apple
3    apple
4    apple
5   orange
6    apple
7    apple
dtype: object
```


```
In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)

In [14]: pd.value_counts(values)
Out[14]:
apple   6
orange  2
dtype: int64
```

Во многих системах работы с данными (организация хранилищ данных, статистические вычисления и т. п.) были разработаны специализированные подходы к представлению данных с повторяющимися значениями с целью повысить эффективность хранения и вычислений. В хранилищах данных рекомендуется использовать *таблицы измерений*, содержащие уникальные значения, а в главной таблице наблюдений хранить целочисленные ключи, ссылающиеся на таблицы измерений:

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)

In [16]: dim = pd.Series(['apple', 'orange'])

In [17]: values
Out[17]:
0   0
1   1
2   0
3   0
4   0
5   1
6   0
7   0
dtype: int64

In [18]: dim
Out[18]:
0    apple
1   orange
dtype: object
```

Для восстановления исходного объекта Series, содержащего строки, можно использовать метод `take`:

```
In [19]: dim.take(values)
Out[19]:
0    apple
1   orange
0    apple
0    apple
0    apple
1   orange
0    apple
0    apple
dtype: object
```

Это представление целыми числами называется *категориальным*, или *словарным*. Массив неповторяющихся значений можно назвать *категориями*, *словарем* или *уровнями* данных. В этой книге мы будем употреблять термины *категориальный* и *категории*. Целые значения, ссылающиеся на категории, будем называть *кодами категорий* или просто *кодами*.

Категориальное представление может дать существенное повышение производительности при выполнении аналитических операций. Кроме того, можно производить преобразования самих категорий, не изменяя коды. Вот два примера преобразований, которые обходятся относительно дешево:

- переименование категорий;
- добавление новой категории без изменения порядка или позиции существующих категорий.

Категориальные типы в pandas

В pandas имеется специальный тип `Categorical` для хранения данных, представленных целочисленными кодами категорий. Рассмотрим приведенный выше пример Series:

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2

In [21]: N = len(fruits)

In [22]: df = pd.DataFrame({'fruit': fruits,
   ....:                    'basket_id': np.arange(N),
   ....:                    'count': np.random.randint(3, 15, size=N),
   ....:                    'weight': np.random.uniform(0, 4, size=N)},
   ....:                   columns=['basket_id', 'fruit', 'count', 'weight'])

In [23]: df
Out[23]:
   basket_id    fruit count    weight
0          0    apple     5  3.858058
1          1   orange     8  2.612708
2          2    apple     4  2.995627
3          3    apple     7  2.614279
4          4    apple    12  2.990859
```

```
5          5   orange     8  3.845227
6          6    apple     5  0.033553
7          7    apple     4  0.425778
```

Здесь `df['fruit']` – массив строковых объектов Python. Его можно преобразовать в категориальное представление следующим образом:

```
In [24]: fruit_cat = df['fruit'].astype('category')

In [25]: fruit_cat
Out[25]:
0    apple
1   orange
2    apple
3    apple
4    apple
5   orange
6    apple
7    apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

Значения `fruit_cat` – это не массив NumPy, а экземпляр класса `pandas.Categorical`:

```
In [26]: c = fruit_cat.values

In [27]: type(c)
Out[27]: pandas.core.categorical.Categorical
```

У объекта `Categorical` имеются атрибуты `categories` и `codes`:

```
In [28]: c.categories
Out[28]: Index(['apple', 'orange'], dtype='object')

In [29]: c.codes
Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

Столбец DataFrame можно привести к категориальному виду, присвоив ему результат преобразования:

```
In [30]: df['fruit'] = df['fruit'].astype('category')

In [31]: df.fruit
Out[31]:
0    apple
1   orange
2    apple
3    apple
4    apple
5   orange
6    apple
7    apple
```

```
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

Можно также создать объект pandas.Categorical непосредственно из последовательностей Python других типов:

```
In [32]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])

In [33]: my_categories
Out[33]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

Если категориальные данные получены из другого источника, то можно воспользоваться альтернативным конструктором from_codes:

```
In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [37]: my_cats_2
Out[37]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

Если явно не оговорено противное, то преобразование к категориальному виду не подразумевает никакого конкретного порядка категорий. То есть порядок элементов массива categories может зависеть от порядка входных данных. При использовании from_codes или любого другого конструктора можно указать, что для категорий существует естественный порядок:

```
In [38]: ordered_cat = pd.Categorical.from_codes(codes, categories,
   ....: ordered=True)

In [39]: ordered_cat
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

Результат [foo < bar < baz] означает, что в отношении порядка 'foo' предшествует 'bar' и т. д. Неупорядоченный категориальный объект можно сделать упорядоченным с помощью метода as_ordered:

```
In [40]: my_cats_2.as_ordered()
Out[40]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

И напоследок отметим, что категориальные данные не обязаны быть строками, хотя во всех примерах выше было именно так. Массив категорий может содержать любые неизменяемые значащие типы.

Вычисления с категориальными значениями

Поведение объектов `Categorical` в pandas, вообще говоря, совпадает с поведением незакодированных объектов (например, массива строк). Некоторые средства pandas, в частности функция `groupby`, с категориальными объектами работают быстрее. Существуют также функции, понимающие флаг `ordered`.

Возьмем какие-нибудь случайные данные и воспользуемся функцией раскладывания по ящикам `pandas.qcut`. Она возвращает объект `pandas.Categorical`; мы уже встречали эту функцию раньше, но прошли мимо деталей работы категориальных объектов:

```
In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000)

In [43]: draws[:5]
Out[43]: array([-0.2047, 0.4789, -0.5194, -0.5557, 1.9658])
```

Разобьем эти данные на квартили и вычислим некоторые статистики:

```
In [44]: bins = pd.qcut(draws, 4)

In [45]: bins
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101], (0.63,
3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (-0.0101, 0.63
], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.0101] < (-0.010
1, 0.63] < (0.63, 3.928]]
```

Точные значения выборочных квартилей не так полезны в отчетах, как имена квартилей. С этой целью можно передать `qcut` аргумент `labels`:

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [47]: bins
Out[47]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

Категориальный объект `bins` с метками не содержит информации о границах ящиков в данных, поэтому мы можем использовать `groupby` для получения сводной статистики:

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
   ....:            .groupby(bins)
```

```
   ....:             .agg(['count', 'min', 'max'])
   ....:             .reset_index())

In [51]: results
Out[51]:
  quartile count       min       max
0       Q1   250 -2.949343 -0.685484
1       Q2   250 -0.683066 -0.010115
2       Q3   250 -0.010032  0.628894
3       Q4   250  0.634238  3.927528
```

Столбец результата 'quartile' сохраняет исходную информацию о категориях из bins, в том числе их порядок:

```
In [52]: results['quartile']
Out[52]:
0 Q1
1 Q2
2 Q3
3 Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

Категориальные объекты повышают производительность

Если вы производите много аналитических операций над некоторым набором данных, то преобразование его в категориальное представление может дать значительный выигрыш в производительности. К тому же категориальный вариант столбца DataFrame часто потребляет заметно меньше памяти. Рассмотрим объект Series, содержащий 10 000 000 элементов, относящихся к небольшому числу различных категорий:

```
In [53]: N = 10000000

In [54]: draws = pd.Series(np.random.randn(N))

In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Теперь преобразуем labels в категориальное представление:

```
In [56]: categories = labels.astype('category')
```

Заметим, что labels занимает гораздо больше памяти, чем categories:

```
In [57]: labels.memory_usage()
Out[57]: 80000080

In [58]: categories.memory_usage()
Out[58]: 10000272
```

Конечно, преобразование в категории не бесплатно, но это одноразовые затраты:

```
In [59]: %time _ = labels.astype('category')
```

```
CPU times: user 490 ms, sys: 240 ms, total: 730 ms
Wall time: 726 ms
```

Операции группировки для категориальных объектов работают намного быстрее, потому что алгоритмы применяются к массиву целочисленных кодов, а не к массиву строк.

Категориальные методы

У объекта Series, содержащего категориальные данные, имеется несколько специальных методов, похожих на методы группы `Series.str` для объектов, содержащих строки. Они также дают удобный доступ к категориям и их кодам. Рассмотрим следующий объект Series:

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0   a
1   b
2   c
3   d
4   a
5   b
6   c
7   d
dtype: category
Categories (4, object): [a, b, c, d]
```

Специальный атрибут `cat` дает доступ ко всем категориальным методам:

```
In [63]: cat_s.cat.codes
Out[63]:
0   0
1   1
2   2
3   3
4   0
5   1
6   2
7   3
dtype: int8
In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```


Допустим, нам известно, что фактический набор категорий для этих данных не ограничивается четырьмя наблюдавшимися значениями. Тогда можно воспользоваться методом `set_categories`, чтобы изменить набор категорий:

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']
In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)
In [67]: cat_s2
Out[67]:
0   a
1   b
2   c
3   d
4   a
5   b
6   c
7   d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

На первый взгляд данные не изменились, но новые категории найдут отражение в операциях, где они используются. Например, `value_counts` учитывает все категории:

```
In [68]: cat_s.value_counts()
Out[68]:
d   2
c   2
b   2
a   2
dtype: int64
```

```
In [69]: cat_s2.value_counts()
Out[69]:
d   2
c   2
b   2
a   2
e   0
dtype: int64
```

В больших наборах данных категориальные объекты часто используются как удобная возможность сэкономить память и повысить производительность. После фильтрации большого объекта DataFrame или Series многие категории могут отсутствовать в данных. Чтобы справиться с этой проблемой, можно использовать метод `remove_unused_categories`, который удаляет ненаблюдаемые категории:

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]
```

```
In [71]: cat_s3
Out[71]:
0   a
1   b
4   a
5   b
```

```
dtype: category
Categories (4, object): [a, b, c, d]

In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0   a
1   b
4   a
5   b
dtype: category
Categories (2, object): [a, b]
```

В табл. 12.1 перечислены имеющиеся категориальные методы.

Таблица 12.1. Категориальные методы класса Series в pandas

Метод	Описание
add_categories	Добавить новые (еще не встречавшиеся) категории в конец списка существующих
as_ordered	Сделать категории упорядоченными
as_unordered	Сделать категории неупорядоченными
remove_categories	Удалить категории, заменив удаленные значения на null
remove_unused_categories	Удалить категории, не встречающиеся в данных
rename_categories	Заменить имена категорий; изменить таким образом число категорий невозможно
reorder_categories	То же, что rename_categories, но позволяет также изменить порядок категорий
set_categories	Заменить категории заданным набором новых категорий; разрешается добавлять и удалять категории

Создание фиктивных переменных для моделирования

При использовании статистических средств или инструментов машинного обучения часто приходится преобразовывать категориальные данные в фиктивные переменные. Этот процесс, который еще называют *унитарным кодированием* (one-hot encoding), сводится к созданию объекта DataFrame, в котором каждой категории соответствует один столбец; столбцы содержат 1 в строках, соответствующих вхождению данной категории, и 0 – в остальных.

Рассмотрим предыдущий пример:

```
In [73]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

Как было сказано в главе 7, функция pandas.get_dummies преобразует эти одномерные категориальные данные в объект DataFrame, содержащий фиктивную переменную:

```
In [74]: pd.get_dummies(cat_s)
Out[74]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
```

```
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

12.2. Дополнительные способы использования GroupBy

Метод `groupby` классов Series и DataFrame подробно обсуждался в главе 10, а здесь мы рассмотрим некоторые дополнительные приемы его использования.

Групповые преобразования и GroupBy с «развертыванием»

В главе 10 мы рассматривали метод `apply` в групповых операциях для выполнения преобразований. Существует также метод `transform`, который похож на `apply`, но налагает дополнительные ограничения на вид применяемой функции:

- она может возвращать скалярное значение, которое укладывается по форме группы;
- она может возвращать объект той же формы, что у входной группы;
- она не должна изменять свои входные данные.

Для иллюстрации рассмотрим простой пример:

```
In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
   ....: 'value': np.arange(12.)})
In [76]: df
Out[76]:
   key value
0    a   0.0
1    b   1.0
2    c   2.0
3    a   3.0
4    b   4.0
5    c   5.0
6    a   6.0
7    b   7.0
8    c   8.0
9    a   9.0
10   b  10.0
11   c  11.0
```

Ниже показаны средние по группам с одинаковым значением ключа `key`:

```
In [77]: g = df.groupby('key').value
```

```
In [78]: g.mean()
Out[78]:
   key
a  4.5
b  5.5
c  6.5
Name: value, dtype: float64
```

Предположим теперь, что мы хотим создать объект Series той же формы, что `df['value']`, но заменить значения средними по группам с одинаковым значением `'key'`. Для выполнения преобразования мы можем передать функцию `lambda x: x.mean()`:


```
In [79]: g.transform(lambda x: x.mean())
Out[79]:
0   4.5
1   5.5
2   6.5
3   4.5
4   5.5
5   6.5
6   4.5
7   5.5
8   6.5
9   4.5
10  5.5
11  6.5
Name: value, dtype: float64
```

Для встроенных функций агрегирования можно передать строковый псевдоним, как в случае метода `agg` объекта GroupBy:

```
In [80]: g.transform('mean')
Out[80]:
0   4.5
1   5.5
2   6.5
3   4.5
4   5.5
5   6.5
6   4.5
7   5.5
8   6.5
9   4.5
10  5.5
11  6.5
Name: value, dtype: float64
```

Как и `apply`, метод `transform` работает с функциями, которые возвращают Series, но результат должен быть такого же размера, как вход. Например, можно умножить каждую группу на 2, воспользовавшись лямбда-функцией:

```
In [81]: g.transform(lambda x: x * 2)
Out[81]:
0    0.0
1    2.0
2    4.0
3    6.0
4    8.0
5   10.0
6   12.0
7   14.0
8   16.0
9   18.0
10  20.0
11  22.0
Name: value, dtype: float64
```


В качестве более сложного примера вычислим ранги каждой группы в порядке убывания:

```
In [82]: g.transform(lambda x: x.rank(ascending=False))
Out[82]:
0    4.0
1    4.0
2    4.0
3    3.0
4    3.0
5    3.0
6    2.0
7    2.0
8    2.0
9    1.0
10   1.0
11   1.0
Name: value, dtype: float64
```


Рассмотрим функцию группового преобразования, составленную из простых операций агрегирования:

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

В этом случае `transform` и `apply` дают одинаковые результаты:

```
In [84]: g.transform(normalize)
Out[84]:
0   -1.161895
1   -1.161895
2   -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
```

```
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64

In [85]: g.apply(normalize)
Out[85]:
0   -1.161895
1   -1.161895
2   -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64
```


Встроенные функции агрегирования типа `'mean'` или `'sum'` часто работают гораздо быстрее общей функции `apply`. Они также обладают «быстрым прошлым» при использовании совместно с `transform`. Это позволяет выполнить так называемую групповую операцию с *развертыванием*:

```
In [86]: g.transform('mean')
Out[86]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
9    4.5
10   5.5
11   6.5
Name: value, dtype: float64
```



```
In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')

In [88]: normalized
Out[88]:
0   -1.161895
```

```
1   -1.161895
2   -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64
```

Хотя групповая операция с развертыванием может включать несколько групповых агрегирований, суммарная выгода от использования векторных операций часто перевешивает этот недостаток.

Групповая передискретизация по времени

Для временных рядов метод `resample` семантически является групповой операцией, основанной на временных интервалах. Рассмотрим небольшую таблицу:

```
In [89]: N = 15

In [90]: times = pd.date_range('2017-05-20 00:00', freq='1min', periods=N)

In [91]: df = pd.DataFrame({'time': times,
   ....:                    'value': np.arange(N)})

In [92]: df
Out[92]:
                  time  value
0  2017-05-20 00:00:00      0
1  2017-05-20 00:01:00      1
2  2017-05-20 00:02:00      2
3  2017-05-20 00:03:00      3
4  2017-05-20 00:04:00      4
5  2017-05-20 00:05:00      5
6  2017-05-20 00:06:00      6
7  2017-05-20 00:07:00      7
8  2017-05-20 00:08:00      8
9  2017-05-20 00:09:00      9
10 2017-05-20 00:10:00     10
11 2017-05-20 00:11:00     11
12 2017-05-20 00:12:00     12
13 2017-05-20 00:13:00     13
14 2017-05-20 00:14:00     14
```

Здесь мы можем проиндексировать по столбцу `'time'`, а затем выполнить передискретизацию:

```
In [93]: df.set_index('time').resample('5min').count()
Out[93]:
                     value
time
2017-05-20 00:00:00      5
2017-05-20 00:05:00      5
2017-05-20 00:10:00      5
```

Предположим, что DataFrame содержит несколько временных рядов, помеченных дополнительным группирующим столбцом key:

```
In [94]: df2 = pd.DataFrame({'time': times.repeat(3),
   ....:                     'key': np.tile(['a', 'b', 'c'], N),
   ....:                     'value': np.arange(N * 3.)})
```

```
In [95]: df2[:7]
Out[95]:
  key                time value
0   a 2017-05-20 00:00:00   0.0
1   b 2017-05-20 00:00:00   1.0
2   c 2017-05-20 00:00:00   2.0
3   a 2017-05-20 00:01:00   3.0
4   b 2017-05-20 00:01:00   4.0
5   c 2017-05-20 00:01:00   5.0
6   a 2017-05-20 00:02:00   6.0
```

Чтобы произвести одинаковую передискретизацию для всех значений 'key', нам понадобится объект pandas.TimeGrouper:

```
In [96]: time_key = pd.TimeGrouper('5min')
```

Затем мы можем проиндексировать по времени, сгруппировать по 'key' и time_key и агрегировать:

```
In [97]: resampled = (df2.set_index('time')
   ....:               .groupby(['key', time_key])
   ....:               .sum())
```

```
In [98]: resampled
Out[98]:
                         value
key time
a   2017-05-20 00:00:00   30.0
    2017-05-20 00:05:00  105.0
    2017-05-20 00:10:00  180.0
b   2017-05-20 00:00:00   35.0
    2017-05-20 00:05:00  110.0
    2017-05-20 00:10:00  185.0
C   2017-05-20 00:00:00   40.0
    2017-05-20 00:05:00  115.0
    2017-05-20 00:10:00  190.0
```

```
In [99]: resampled.reset_index()
Out[99]:
  key                time  value
0   a 2017-05-20 00:00:00   30.0
1   a 2017-05-20 00:05:00  105.0
2   a 2017-05-20 00:10:00  180.0
3   b 2017-05-20 00:00:00   35.0
4   b 2017-05-20 00:05:00  110.0
5   b 2017-05-20 00:10:00  185.0
6   c 2017-05-20 00:00:00   40.0
7   c 2017-05-20 00:05:00  115.0
8   c 2017-05-20 00:10:00  190.0
```

При работе с `TimeGrouper` имеется ограничение: время должно быть индексом объекта Series или DataFrame.

12.3. Сцепление методов

Если к набору данных применяется последовательность преобразований, то создается много временных переменных, которые для анализа совершенно не нужны. Рассмотрим пример:

```
df = load_data()
df2 = df[df['col2'] < 0]
df2['col1_demeaned'] = df2['col1'] - df2['col1'].mean()
result = df2.groupby('key').col1_demeaned.std()
```

Хотя реальные данные здесь не используются, этот пример дает представление о некоторых новых методах. Во-первых, метод `DataFrame.assign` – *функциональная* альтернатива присваиванию столбцов вида `df[k] = v`. Вместо того чтобы модифицировать объект на месте, он возвращает новый DataFrame с заданными модификациями. Таким образом, следующие предложения эквивалентны:

```
# Обычный нефункциональный способ
df2 = df.copy()
df2['k'] = v

# Функциональный способ присваивания
df2 = df.assign(k=v)
```

Присваивание на месте, возможно, работает быстрее, чем `assign`, зато `assign` открывает путь к сцеплению методов:

```
result = (df2.assign(col1_demeaned=df2.col1 - df2.col2.mean())
          .groupby('key')
          .col1_demeaned.std())
```

Я добавил внешние скобки, чтобы удобнее было вставлять разрывы строки.

Применяя сцепление методов, следует помнить, что иногда приходится ссылаться на временные объекты. В примере выше мы не можем сослаться

на результат load_data, пока он не будет присвоен временной переменной df. Чтобы решить эту проблему, assign и многие другие функции pandas принимают похожие на функции аргументы, которые еще называют *вызываемыми объектами*.

Чтобы продемонстрировать вызываемые объекты в действии, рассмотрим приведенный выше фрагмент кода:

```
df = load_data()
df2 = df[df['col2'] < 0]
```


Его можно переписать так:

```
df = (load_data()
      [lambda x: x['col2'] < 0])
```

Здесь результат load_data не присваивается переменной, поэтому функция, переданная оператору [], *связывается* с объектом на стадии сцепления методов.

Мы можем продолжить и записать всю последовательность в виде одного сцепленного выражения:

```
result = (load_data()
          [lambda x: x.col2 < 0]
          .assign(col1_demeaned=lambda x: x.col1 - x.col1.mean())
          .groupby('key')
          .col1_demeaned.std())
```

Какой стиль написания кода предпочесть, дело вкуса, но разбиение выражения на несколько шагов может сделать код более понятным.

Метод pipe

С помощью встроенных в pandas функций и рассмотренного выше сцепления методов уже можно достичь многого, но иногда требуется использовать собственные функции или функции из сторонних библиотек. В этом случае на помощь приходит метод pipe.

Рассмотрим последовательность вызовов функций:

```
a = f(df, arg1=v1)
b = g(a, v2, arg3=v3)
c = h(b, arg4=v4)
```


Если все используемые функции принимают и возвращают объекты Series или DataFrame, то эту последовательность можно переписать с применением метода pipe:

```
result = (df.pipe(f, arg1=v1)
          .pipe(g, v2, arg3=v3)
          .pipe(h, arg4=v4))
```

Конструкции f(df) и df.pipe(f) эквивалентны, но pipe упрощает сцепление вызовов.

Метод pipe может быть полезен в ситуации, когда нужно оформить последовательность операций в виде функции, допускающей повторное использование. Например, рассмотрим вычитание группового среднего из столбца:

```
g = df.groupby(['key1', 'key2'])
df['col1'] = df['col1'] - g.transform('mean')
```

Предположим, что требуется иметь возможность вычитать среднее сразу из нескольких столбцов и легко менять групповые ключи. Кроме того, желательно иметь возможность включать это преобразование в цепочку методов. Ниже приведена одна из возможных реализаций:

```
def group_demean(df, by, cols):
    result = df.copy()
    g = df.groupby(by)
    for c in cols:
        result[c] = df[c] - g[c].transform('mean')
    return result
```

Тогда можно написать:

```
result = (df[df.col1 < 0]
          .pipe(group_demean, ['key1', 'key2'], ['col1']))
```

12.4. Заключение

Pandas, как и многие проекты с открытым исходным кодом, трансформируется и приобретает новые и улучшенные функции. Как и в других местах книги, в этой главе упор сделан на стабильную функциональность, которая вряд ли изменится в ближайшие годы.

Всех, кто хочет обогатить свой опыт работы с pandas, призываю читать документацию и замечания, которые выпускаются вместе с новыми версиями. Мы также приглашаем вас присоединиться к разработке pandas: исправлять ошибки, реализовывать новые возможности и улучшать документацию.

Глава 13. Введение в библиотеки моделирования на Python

В этой книге я стремился в первую очередь заложить фундамент для анализа данных на Python. Поскольку специалисты по анализу данных и науке о данных часто жалуются на непомерно высокие временные затраты на переформатирование и подготовку данных, структура книги отражает важность овладения этими навыками.

Какую библиотеку использовать для разработки моделей, зависит от приложения. Многие статистические задачи можно решить, применяя более простые методы, например регрессию обычным методом наименьших квадратов, тогда как для других задач требуются более развитые методы машинного обучения. По счастью, Python стал одним из избранных языков для реализации аналитических методов, поэтому, прочитав книгу до конца, вы сможете исследовать многочисленные инструменты, имеющиеся в вашем распоряжении.

В этой главе я сделаю обзор некоторых возможностей pandas, которые могут пригодиться, когда вы будете переходить от переформатирования данных к подгонке и оцениванию моделей. Затем дам краткое введение в две популярные библиотеки моделирования: statsmodels (http://www.statsmodels.org/stable/index.html) и scikit-learn (https://scikit-learn.org/). Поскольку оба проекта настолько велики, что заслуживают отдельной книги, я даже не пытаюсь рассмотреть их во всей полноте, а отсылаю к документации и другим книгам по науке о данных, статистике и машинному обучению.

13.1. Интерфейс между pandas и кодом модели

Широко распространенный подход к разработке моделей заключается в том, чтобы с помощью pandas произвести загрузку и очистку данных, а затем воспользоваться библиотекой моделирования для построения самой модели.

Важная часть процесса разработки модели в машинном обучении называется *конструированием признаков* (feature engineering). Речь идет об описании преобразований данных или аналитических операций, извлекающих из исходного набора данных информацию, которая может оказаться полезной в контексте моделирования. Рассмотренные в этой книге средства группировки и агрегирования часто применяются для конструирования признаков.

Хотя детали «правильного» конструирования признаков выходят за рамки этой книги, я все же продемонстрирую некоторые приемы, позволяющие относительно безболезненно переключаться между манипулированием данными средствами pandas и моделированием.

Переходником между pandas и другими аналитическими библиотеками обычно являются массивы NumPy. Для преобразования объекта DataFrame в массив NumPy используется свойство `.values`:

```
In [10]: import pandas as pd

In [11]: import numpy as np

In [12]: data = pd.DataFrame({
   ....:      'x0': [1, 2, 3, 4, 5],
   ....:      'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:      'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0     x1     y
0   1   0.01  -1.5
1   2  -0.01   0.0
2   3   0.25   3.6
3   4  -4.10   1.3
4   5   0.00  -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.values
Out[15]:
array([[ 1. , 0.01, -1.5 ],
[ 2. , -0.01, 0. ],
[ 3. , 0.25, 3.6 ],
[ 4. , -4.1 , 1.3 ],
[ 5. , 0. , -2. ]])
```

Для обратного преобразования в DataFrame можно передать двумерный массив ndarray и факультативно имена столбцов:

```
In [16]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])

In [17]: df2
Out[17]:
   one    two  three
0  1.0   0.01   -1.5
```

```
1  2.0  -0.01    0.0
2  3.0   0.25    3.6
3  4.0  -4.10    1.3
4  5.0   0.00   -2.0
```

Атрибут .values предназначен для использования в случае, когда данные однородны – например, все имеют числовой тип. При наличии неоднородных данных получится массив ndarray объектов Python:

```
In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']

In [20]: df3
Out[20]:
   x0     x1     y  strings
0   1   0.01  -1.5        a
1   2  -0.01   0.0        b
2   3   0.25   3.6        c
3   4  -4.10   1.3        d
4   5   0.00  -2.0        e

In [21]: df3.values
Out[21]:
array([[1,  0.01, -1.5, 'a'],
       [2, -0.01,  0.0, 'b'],
       [3,  0.25,  3.6, 'c'],
       [4, -4.1 ,  1.3, 'd'],
       [5,  0.0 , -2.0, 'e']], dtype=object)
```

Для некоторых моделей нужно использовать лишь подмножество столбцов. Я рекомендую loc-индексирование и values:

```
In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].values
Out[23]:
array([[ 1. ,  0.01],
       [ 2. , -0.01],
       [ 3. ,  0.25],
       [ 4. , -4.1 ],
       [ 5. ,  0. ]])
```

В некоторые библиотеки уже встроена поддержка pandas, и часть работы они делают автоматически: выполняют преобразование в NumPy из DataFrame и присоединяют имена параметров модели к столбцам выходных таблиц или объектов Series. Если это не так, то такое «управление метаданными» ложится на вас.

В главе 12 мы рассматривали тип pandas Categorical и функцию pandas.get_dummies. Пусть в нашем наборе данных имеется нечисловой столбец:

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
   ....:                                   categories=['a', 'b'])
```

```
In [25]: data
Out[25]:
   x0     x1     y category
0   1   0.01  -1.5        a
1   2  -0.01   0.0        b
2   3   0.25   3.6        a
3   4  -4.10   1.3        a
4   5   0.00  -2.0        b
```

Если бы мы хотели заменить столбец `'category'` фиктивными переменными, то создали бы фиктивные переменные, удалили столбец `'category'` и выполнили операцию соединения:

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')

In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)

In [28]: data_with_dummies
Out[28]:
   x0     x1     y category_a category_b
0   1   0.01  -1.5          1          0
1   2  -0.01   0.0          0          1
2   3   0.25   3.6          1          0
3   4  -4.10   1.3          1          0
4   5   0.00  -2.0          0          1
```

При аппроксимации некоторых статистических моделей с помощью фиктивных переменных возникают нюансы. Если приходится работать не только с числовыми столбцами, то, пожалуй, проще и безопаснее использовать библиотеку Patsy (тема следующего раздела).

13.2. Описание моделей с помощью Patsy

Patsy (https://patsy.readthedocs.io/en/latest/) – написанная на Python библиотека для описания статистических моделей (особенно линейных) с применением простого строкового «синтаксиса формул», в основу которого положены формулы из языков статистического программирования R и S (с некоторыми изменениями).

Patsy широко используется в statsmodels для задания линейных моделей, поэтому я расскажу об основных чертах, чтобы вам было проще приступить к работе с ней. Для *формул* в Patsy применяется такой синтаксис:

```
y ~ x0 + x1
```

Конструкция `a + b` – это не сложение `a` и `b`. Имеется в виду, что это *термы* созданной для модели *матрицы плана*. Функция `patsy.dmatrices` принимает строку формулы вместе с набором данных (который может быть представлен в виде объекта DataFrame или словаря массивов) и порождает матрицы плана для линейной модели:

```
In [29]: data = pd.DataFrame({
   ....: 'x0': [1, 2, 3, 4, 5],
   ....: 'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....: 'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0     x1     y
0   1   0.01  -1.5
1   2  -0.01   0.0
2   3   0.25   3.6
3   4  -4.10   1.3
4   5   0.00  -2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

Теперь имеем:

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
   3.6
   1.3
  -2.0
  Terms:
    'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0     x1
          1   1   0.01
          1   2  -0.01
          1   3   0.25
          1   4  -4.10
          1   5   0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

Эти объекты класса Patsy `DesignMatrix` являются массивами NumPy ndarray с дополнительными метаданными:

```
In [35]: np.asarray(y)
Out[35]:
array([[-1.5],
       [ 0. ],
```

```
       [ 3.6],
       [ 1.3],
       [-2. ]])
In [36]: np.asarray(X)
Out[36]:
array([[ 1. , 1. ,  0.01],
       [ 1. , 2. , -0.01],
       [ 1. , 3. ,  0.25],
       [ 1. , 4. , -4.1 ],
       [ 1. , 5. ,  0. ]])
```

Вам, наверное, интересно, откуда взялся свободный член – терм `Intercept`. Это соглашение, принятое для таких линейных моделей, как регрессия обычным методом наименьших квадратов. Свободный член можно подавить, добавив в модель терм `+ 0`:

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0     x1
   1   0.01
   2  -0.01
   3   0.25
   4  -4.10
   5   0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

Объекты Patsy можно передать напрямую в алгоритм типа `numpy.linalg.lstsq`, который выполняет регрессию обычным методом наименьших квадратов:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

Метаданные модели сохраняются в атрибуте `design_info`, так что имена столбцов модели можно связать с найденными коэффициентами для получения объекта Series, например:

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])

In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)

In [41]: coef
Out[41]:
Intercept  0.312910
x0        -0.079106
x1        -0.265464
dtype: float64
```

Преобразование данных в формулах Patsy

В формулы Patsy можно включить код на Python; при вычислении формулы библиотека будет искать использованные функции в объемлющей области видимости:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
  Intercept  x0  np.log(np.abs(x1) + 1)
          1   1                 0.00995
          1   2                 0.00995
          1   3                 0.22314
          1   4                 1.62924
          1   5                 0.00000
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

Из часто используемых преобразований отметим стандартизацию (приведение к распределению со средним 0 и дисперсией 1) и центрирование (вычитание среднего). Для этих целей в Patsy имеются встроенные функции:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)

In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
  Intercept standardize(x0) center(x1)
          1       -1.41421        0.78
          1       -0.70711        0.76
          1        0.00000        1.02
          1        0.70711       -3.33
          1       1.41421        0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

В процессе моделирования мы иногда обучаем модель на одном наборе данных, а затем тестируем на другом. В роли другого набора может выступать *зарезервированная* часть данных или новые данные, полученные позже. Применяя преобразования типа центрирования и стандартизации, следует быть осторожным, когда модель используется для предсказания на новых данных. Говорят, что такие преобразования *обладают состоянием*, потому что при

преобразовании нового набора данных мы должны использовать статистики, в частности среднее и стандартное отклонение, исходного набора.

Функция `patsy.build_design_matrices` умеет применять преобразования к новым *вневыборочным* данным, используя информацию, сохраненную для исходного *внутривыборочного* набора данных:

```
In [46]: new_data = pd.DataFrame({
   ....:     'x0': [6, 7, 8, 9],
   ....:     'x1': [3.1, -0.5, 0, 2.3],
   ....:     'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
  Intercept standardize(x0) center(x1)
          1        2.12132        3.87
          1        2.82843        0.27
          1        3.53553        0.77
          1        4.24264        3.07
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)]
```

Поскольку знак + в формулах Patsy не означает сложения, в случае если мы хотим сложить именованные столбцы из набора данных, необходимо обернуть операцию специальной функцией `I`:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
  Intercept  I(x0 + x1)
          1        1.01
          1        1.99
          1        3.25
          1       -0.10
          1        5.00
  Terms:
    'Intercept' (column 0)
    'I(x0 + x1)' (column 1)
```

В Patsy встроены и другие преобразования, находящиеся в модуле `patsy.builtins`. Дополнительные сведения см. в онлайновой документации.

Для категориальных данных имеется специальный класс преобразований, о котором я расскажу ниже.

Категориальные данные и Patsy

Нечисловые данные можно преобразовать для использования в матрице плана модели многими способами. Полное рассмотрение этой темы выходит за рамки книги и относится скорее к курсу математической статистики.

Когда в формуле Patsy встречаются нечисловые члены, они по умолчанию преобразуются в фиктивные переменные. Если имеется свободный член, то один из уровней будет пропущен, чтобы избежать коллинеарности:

```
In [51]: data = pd.DataFrame({
   ....:      'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
   ....:      'key2': [0, 1, 0, 1, 0, 1, 0, 0],
   ....:      'v1': [1, 2, 3, 4, 5, 6, 7, 8],
   ....:      'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
   ....: })

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

In [53]: X
Out[53]:
DesignMatrix with shape (8, 2)
  Intercept key1[T.b]
          1         0
          1         0
          1         1
          1         1
          1         0
          1         1
          1         0
          1         1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
```

Если исключить из модели свободный член, то столбцы, соответствующие каждому значению категории, будут включены в модельную матрицу плана:

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)

In [55]: X
Out[55]:
DesignMatrix with shape (8, 2)
  key1[a]  key1[b]
        1        0
        1        0
        0        1
        0        1
        1        0
        0        1
        1        0
        0        1
```

```
  Terms:
    'key1' (columns 0:2)
```

Числовые столбцы можно интерпретировать как категориальные с помощью функции C:

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
  Intercept  C(key2)[T.1]
          1             0
          1             1
          1             0
          1             1
          1             0
          1             1
          1             0
          1             0
  Terms:
    'Intercept' (column 0)
    'C(key2)' (column 1)
```

Если в модели несколько категориальных термов, то ситуация осложняется, поскольку можно включать термы взаимодействия вида key1:key2, например в моделях дисперсионного анализа (ANOVA):

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})

In [59]: data
Out[59]:
  key1  key2  v1    v2
0    a  zero   1  -1.0
1    a   one   2   0.0
2    b  zero   3   2.5
3    b   one   4  -0.5
4    a  zero   5   4.0
5    b   one   6  -1.2
6    a  zero   7   0.2
7    b  zero   8  -1.7

In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
  Intercept  key1[T.b]  key2[T.zero]
          1          0             1
          1          0             0
          1          1             1
          1          1             0
```

```
          1          0          1
          1          1          0
          1          0          1
          1          1          1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)
```



```
In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)

In [63]: X
Out[63]:
DesignMatrix with shape (8, 4)
  Intercept  key1[T.b]  key2[T.zero] key1[T.b]:key2[T.zero]
          1          0             1                       0
          1          0             0                       0
          1          1             1                       1
          1          1             0                       0
          1          0             1                       0
          1          1             0                       0
          1          0             1                       0
          1          1             1                       1
  Terms:
   'Intercept' (column 0)
   'key1' (column 1)
   'key2' (column 2)
   'key1:key2' (column 3)
```

Patsy предлагает и другие способы преобразования категориальных данных, в том числе преобразования для термов в определенном порядке. Дополнительные сведения см. в документации.

13.3. Введение в statsmodels

Statsmodels – написанная на Python библиотека для подгонки разнообразных статистических моделей, выполнения статистических тестов, исследования и визуализации данных. В statsmodels представлены в основном «классические» статистические методы на основе частотного подхода, а байесовские методы и модели машинного обучения лучше искать в других библиотеках.

Перечислим несколько видов моделей, имеющихся в statsmodels:

- линейные модели, обобщенные линейные модели и робастные линейные модели;
- линейные модели со смешанными эффектами;
- методы дисперсионного анализа (ANOVA);
- временные ряды и модели в пространстве состояний;
- обобщенные методы моментов.

На следующих страницах мы воспользуемся несколькими базовыми средствами statsmodels и посмотрим, как применять интерфейсы библиотеки моделирования с формулами Patsy объектами pandas DataFrame.

Оценивание линейных моделей

В библиотеке statsmodels имеется несколько видов моделей линейной регрессии – от самых простых (обычный метод наименьших квадратов) до более сложных (метод наименьших квадратов с итерационным повторным взвешиванием).

Линейные модели в statsmodels имеют два основных интерфейса: на основе массивов и на основе формул. Доступ к ним производится путем импорта следующих модулей:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

Чтобы продемонстрировать их использование, сгенерируем линейную модель по случайным данным:

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn(*size)

# Для вопроизводимости результатов
np.random.seed(12345)

N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]

y = np.dot(X, beta) + eps
```

Здесь я записал «истинную» модель с известными параметрами `beta`. В данном случае `dnorm` – вспомогательная функция для генерации нормально распределенных данных с заданными средним и дисперсией. Таким образом, имеем:

```
In [66]: X[:5]
Out[66]:
array([[-0.1295, -1.2128, 0.5042],
       [ 0.3029, -0.4357, -0.2542],
       [-0.3285, -0.0253,  0.1384],
       [-0.3515, -0.7196, -0.2582],
       [ 1.2433, -0.3738, -0.5226]])
In [67]: y[:5]
Out[67]: array([ 0.4279, -0.6735, -0.0909, -0.4895, -0.1289])
```

При подгонке линейной модели обычно присутствует свободный член, как мы видели ранее при изучении Patsy. Функция `sm.add_constant` может добавить столбец свободного члена в существующую матрицу:

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1. , -0.1295, -1.2128,  0.5042],
       [ 1. ,  0.3029, -0.4357, -0.2542],
       [ 1. , -0.3285, -0.0253,  0.1384],
       [ 1. , -0.3515, -0.7196, -0.2582],
       [ 1. ,  1.2433, -0.3738, -0.5226]])
```

Класс `sm.OLS` реализует линейную регрессию обычным методом наименьших квадратов:

```
In [70]: model = sm.OLS(y, X)
```

Метод модели `fit` возвращает объект с результатами регрессии, содержащий оценки параметров модели и диагностическую информацию:

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([ 0.1783, 0.223 , 0.501 ])
```

Метод `summary` объекта `results` печатает подробную диагностическую информацию о модели:

```
In [73]: print(results.summary())
OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.430
Model:                            OLS   Adj. R-squared:                  0.413
Method:                 Least Squares   F-statistic:                     24.42
Date:                Mon, 25 Sep 2017   Prob (F-statistic):           7.44e-12
Time:                        14:06:15   Log-Likelihood:                -34.305
No. Observations:                 100   AIC:                             74.61
Df Residuals:                      97   BIC:                             82.42
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             0.1783      0.053      3.364      0.001       0.073       0.283
x2             0.2230      0.046      4.818      0.000       0.131       0.315
x3             0.5010      0.080      6.237      0.000       0.342       0.660
==============================================================================
Omnibus:                        4.662   Durbin-Watson:                   2.201
Prob(Omnibus):                  0.097   Jarque-Bera (JB):                4.098
```

```
Skew:                          0.481    Prob(JB):                         0.129
Kurtosis:                      3.243    Cond. No.                         1.74
==============================================================================
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Параметрам присвоены обобщенные имена x1, x2 и т. д. Но пусть вместо этого все параметры модели хранятся в объекте DataFrame:

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])

In [75]: data['y'] = y

In [76]: data[:5]
Out[76]:
       col0      col1      col2         y
0 -0.129468 -1.212753  0.504225  0.427863
1  0.302910 -0.435742 -0.254180 -0.673480
2 -0.328522 -0.025302  0.138351 -0.090878
3 -0.351475 -0.719605 -0.258215 -0.489494
4  1.243269 -0.373799 -0.522629 -0.128941
```

Теперь можно использовать формульный API statsmodels и строковые формулы Patsy:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()

In [78]: results.params
Out[78]:
Intercept   0.033559
col0        0.176149
col1        0.224826
col2        0.514808
dtype: float64

In [79]: results.tvalues
Out[79]:
Intercept   0.952188
col0        3.319754
col1        4.850730
col2        6.303971
dtype: float64
```

Заметим, что statsmodels вернула результаты в виде Series, присоединив имена столбцов из DataFrame. Кроме того, при работе с формулами и объектами pandas не нужно использовать add_constant.

Зная оценки параметров, мы можем вычислить для них предсказанные моделью значения для новых вневыборочных данных:

```
In [80]: results.predict(data[:5])
```

```
Out[80]:
0   -0.002327
1   -0.141904
2    0.041226
3   -0.323070
4   -0.100535
dtype: float64
```

В statsmodels существует еще много инструментов для анализа, диагностики и визуализации результатов линейных моделей – изучайте сколько душе угодно. Имеются также другие виды линейных моделей, помимо обычного метода наименьших квадратов.

Оценивание процессов с временными рядами

Еще один класс моделей в statsmodels предназначен для анализа временных рядов. Это в том числе процессы авторегрессии, фильтры Калмана и другие модели в пространстве состояний, а также многомерные авторегрессионные модели.

Смоделируем данные временного ряда с авторегрессионной структурой и шумом:

```
init_x = 4

import random
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

У этих данных структура AR(2) (два *лага*) с параметрами 0.8 и –0.4. Когда аппроксимируется AR-модель, мы не всегда знаем, сколько членов лага включать, поэтому можем аппроксимировать модель с несколько бóльшим числом лагов:

```
In [82]: MAXLAGS = 5

In [83]: model = sm.tsa.AR(values)

In [84]: results = model.fit(MAXLAGS)
```

В оценках параметров в `results` сначала идет свободный член, а затем оценки двух первых лагов:

```
In [85]: results.params
Out[85]: array([-0.0062, 0.7845, -0.4085, -0.0136, 0.015 , 0.0143])
```

В этой книге я не могу углубиться в детали данных моделей и интерпретации результатов, но в документации по statsmodels вас ждет много открытий.

13.4. Введение в scikit-learn

Библиотека scikit-learn – один из самых популярных и пользующихся доверием универсальных инструментов машинного обучения на Python. Она содержит широкий спектр стандартных методов машинного обучения с учителем и без учителя, предназначенных для выбора и оценки модели, загрузки и преобразования данных и сохранения моделей. Эти модели можно использовать для классификации, кластеризации, предсказания и решения других типичных задач.

Существуют прекрасные онлайновые и печатные ресурсы, из которых можно узнать о машинном обучении и применении таких библиотек, как scikit-learn и TensorFlow, к решению реальных задач. В этом разделе я дам лишь краткое представление о стилистических особенностях API scikit-learn.

На момент написания этой книги scikit-learn не была глубоко интегрирована с pandas, хотя в разработке находится несколько сторонних дополнительных пакетов. Впрочем, pandas может быть полезна для предварительной обработки данных перед построением модели.

В качестве примера возьму ставший уже классическим набор данных с конкурса Kaggle (https://www.kaggle.com/c/titanic), содержащий сведения о пассажирах «Титаника», затонувшего в 1912 году. Загрузим обучающий и тестовый наборы с помощью pandas:

```
In [86]: train = pd.read_csv('datasets/titanic/train.csv')

In [87]: test = pd.read_csv('datasets/titanic/test.csv')

In [88]: train[:4]
Out[88]:
   PassengerId Survived Pclass \
0            1        0      3
1            2        1      1
2            3        1      3
3            4        1      1

                                                Name     Sex   Age   SibSp \
0                            Braund, Mr. Owen Harris    male  22.0       1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0       1
2                             Heikkinen, Miss. Laina  female  26.0       0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0       1
   Parch            Ticket      Fare    Cabin   Embarked
0      0         A/5 21171    7.2500      NaN          S
1      0          PC 17599   71.2833      C85          C
2      0  STON/O2. 3101282    7.9250      NaN          S
3      0            113803   53.1000     C123          S
```

Библиотеки типа statsmodels и scikit-learn, вообще говоря, не допускают подачи на вход неполных данных, поэтому мы посмотрим, в каких столбцах есть отсутствующие данные:

```
In [89]: train.isnull().sum()
Out[89]:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64

In [90]: test.isnull().sum()
Out[90]:
PassengerId      0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
dtype: int64
```

В статистике и машинном обучении типичная задача заключается в том, чтобы предсказать, выживет ли пассажир, на основе признаков, содержащихся в данных. Модель обучается на *обучающем* наборе данных, а затем оценивается на непересекающемся с ним *тестовом* наборе данных.

Я хотел бы использовать в качестве предсказательного признака возраст Age, но в этом столбце есть отсутствующие данные. Существует несколько способов *подставить* отсутствующие данные, я выберу самый простой и заменю значения null в обеих таблицах медианными значениями для обучающего набора:

```
In [91]: impute_value = train['Age'].median()

In [92]: train['Age'] = train['Age'].fillna(impute_value)

In [93]: test['Age'] = test['Age'].fillna(impute_value)
```

Теперь мы должны описать модель. Я добавлю столбец IsFemale, вычисляемый на основе столбца 'Sex':

```
In [94]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)

In [95]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

Выберем переменные модели и создадим массивы NumPy:

```
In [96]: predictors = ['Pclass', 'IsFemale', 'Age']

In [97]: X_train = train[predictors].values

In [98]: X_test = test[predictors].values

In [99]: y_train = train['Survived'].values

In [100]: X_train[:5]
Out[100]:
array([[ 3., 0., 22.],
       [ 1., 1., 38.],
       [ 3., 1., 26.],
       [ 1., 1., 35.],
       [ 3., 0., 35.]])

In [101]: y_train[:5]
Out[101]: array([0, 1, 1, 1, 0])
```

Я вовсе не утверждаю, что это хорошая модель или что признаки сконструированы правильно. Мы воспользуемся моделью LogisticRegression из scikit-learn и создадим ее экземпляр:

```
In [102]: from sklearn.linear_model import LogisticRegression
In [103]: model = LogisticRegression()
```

Так же как в statsmodels, мы можем подогнать модель к обучающим данным с помощью метода модели fit:

```
In [104]: model.fit(X_train, y_train)
Out[104]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Теперь можно вычислить предсказания для тестового набора данных методом model.predict:

```
In [105]: y_predict = model.predict(X_test)

In [106]: y_predict[:10]
Out[106]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

Если бы мы знали истинные значения для тестового набора, то могли бы вычислить верность в процентах или еще какой-нибудь показатель ошибки:

```
(y_true == y_predict).mean()
```

На практике обучение модели гораздо сложнее. У многих моделей есть настраиваемые параметры, и существуют такие приемы, как *перекрестный контроль*, позволяющие избежать переобучения модели. Их использование часто повышает предсказательную способность, или робастность, модели на новых данных.

Идея перекрестного контроля состоит в том, чтобы разделить обучающий набор на две части с целью смоделировать предсказание на не предъявлявшихся модели данных. Взяв за основу какую-нибудь оценку верности модели, например среднеквадратическую ошибку, можно произвести подбор параметров модели путем поиска на сетке. Для некоторых моделей, в частности логистической регрессии, существуют классы оценивания, в которые перекрестный контроль уже встроен. Например, класс `LogisticRegressionCV` можно использовать, задав в качестве параметра мелкость сетки, на которой производится поиск регуляризирующего параметра модели `C`:

```
In [107]: from sklearn.linear_model import LogisticRegressionCV

In [108]: model_cv = LogisticRegressionCV(10)

In [109]: model_cv.fit(X_train, y_train)
Out[109]:
LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
           fit_intercept=True, intercept_scaling=1.0, max_iter=100,
           multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
           refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

Чтобы выполнить перекрестный контроль вручную, можно воспользоваться вспомогательной функцией `cross_val_score`, которая отвечает за процесс разделения данных. Например, чтобы подвергнуть нашу модель перекрестному контролю, разбив обучающий набор на четыре непересекающиеся части, нужно написать:

```
In [110]: from sklearn.model_selection import cross_val_score

In [111]: model = LogisticRegression(C=10)

In [112]: scores = cross_val_score(model, X_train, y_train, cv=4)

In [113]: scores
Out[113]: array([ 0.7723, 0.8027, 0.7703, 0.7883])
```

Подразумеваемая по умолчанию метрика оценивания зависит от модели, но можно задать функцию оценивания явно. Обучение с перекрестным контролем занимает больше времени, но часто дает модель лучшего качества.

13.5. Продолжение своего образования

Я лишь очень поверхностно описал некоторые библиотеки моделирования на Python, в то время как существуют многочисленные системы для различных

видов статистической обработки и машинного обучения, либо написанные на Python, либо имеющие интерфейс из Python.

В этой книге речь идет прежде всего о переформатировании данных, но есть много других библиотек, посвященных моделированию и инструментарию науки о данных. Перечислим некоторые из них:

- Andreas Mueller and Sarah Guido «Introduction to Machine Learning with Python» (O'Reilly)[1];
- Jake VanderPlas «Python Data Science Handbook» (O'Reilly)[2];
- Joel Grus «Data Science from Scratch: First Principles with Python» (O'Reilly)[3];
- Sebastian Raschka «Python Machine Learning» (Packt Publishing)[4];
- Aurelien Geron «Hands-On Machine Learning with Scikit-Learn and TensorFlow» (O'Reilly)[5].

Книги конечно, ценное подспорье для изучения предмета, но иногда они устаревают, поскольку программное обеспечение с открытым исходным кодом быстро изменяется. Чтобы всегда быть в курсе последних возможностей и версии API, имеет прямой смысл почитать документацию по различным системам статистики и машинного обучения.

[1] *Мюллер А., Гвидо С.* Введение в машинное обучение с помощью Python. М.: Вильямс, 2017.

[2] *Плас Д. В.* Python для сложных задач. Наука о данных и машинное обучение. СПб.: Питер, 2018.

[3] *Грас Д.* Data Science. Наука о данных с нуля. СПб.: БХВ-Петербург, 2019.

[4] *Рашка С.* Python и машинное обучение. М.: ДМК-Пресс, 2017.

[5] *Жерон О.* Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow. М.: Вильямс, 2018.

Глава 14. Примеры анализа данных

Вот и кончились основные главы книги, и теперь мы рассмотрим несколько реальных наборов данных. Для каждого набора применим описанные в книге приемы, чтобы извлечь смысл из исходных данных. Продемонстрированная техника пригодна и для любых других наборов данных, в том числе ваших собственных.

Все примеры наборов данных имеются в репозитории этой книги на GitHub (http://github.com/wesm/pydata-book).

14.1. 1.usa.gov data from Bitly

В 2011 году служба сокращения URL-адресов bit.ly заключила партнерское соглашение с сайтом правительства США USA.gov (https://www.usa.gov/) о синхронном предоставлении анонимных данных о пользователях, которые сокращают ссылки, заканчивающиеся на .gov или .mil. В 2011 году, помимо синхронной ленты, формировались ежечасные мгновенные снимки, доступные в виде текстовых файлов. В 2017 году эта служба уже закрылась, но мы сохранили файлы данных и приводим их в качестве примеров.

В мгновенном снимке каждая строка представлена в формате JSON (JavaScript Object Notation), широко распространенном в вебе. Например, первая строка файла выглядит примерно так:

```
In [5]: path = 'datasets/bitly_usagov/example.txt'
```

```
In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

Для Python имеется много встроенных и сторонних модулей, позволяющих преобразовать JSON-строку в объект словаря Python. Ниже я воспользовался модулем json; принадлежащая ему функция loads вызывается для каждой строки загруженного мной файла:

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

Получившийся в результате объект records представляет собой список словарей Python:

```
In [18]: records[0]
Out[18]:
{'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like
Gecko) Chrome/17.0.963.78 Safari/535.11',
  'al': u'en-US,en;q=0.8',
  'c': u'US',
  'cy': u'Danvers',
  'g': u'A6qOVH',
  'gr': u'MA',
  'h': u'wfLQtf',
  'hc': 1331822918,
  'hh': u'1.usa.gov',
  'l': u'orofrog',
  'll': [42.576698, -70.954903],
  'nk': 1,
  'r': u'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
  't': 1331923247,
  'tz': u'America/New_York',
  'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Подсчет часовых поясов на чистом Python

Допустим, нас интересуют часовые пояса, чаще всего встречающиеся в наборе данных (поле tz). Решить эту задачу можно разными способами. Во-первых, можно извлечь список часовых поясов, снова воспользовавшись списковым включением:

```
In [12]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

Вот те раз! Оказывается, что не во всех записях есть поле часового пояса. Это легко поправить, добавив проверку if 'tz' in rec в конец спискового включения:

```
In [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

Мы видим, что уже среди первых десяти часовых поясов встречаются неизвестные (пустые). Их можно было бы тоже отфильтровать, но я пока оставлю. Покажу два способа подсчитать количество часовых поясов: трудный (в котором используется только стандартная библиотека Python) и легкий (с помощью pandas). Для подсчета можно завести словарь для хранения счетчиков и обойти весь список часовых поясов:

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

Воспользовавшись более продвинутыми средствами из стандартной библиотеки Python, можно записать то же самое короче:

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # значения будут инициализированы нулями
    for x in sequence:
        counts[x] += 1
    return counts
```

Чтобы можно было повторно воспользоваться этим кодом, я поместил его в функцию. Для применения его к часовым поясам достаточно передать этой функции список time_zones:

```
In [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440
```

Чтобы получить только первые десять часовых поясов со счетчиками, придется поколдовать над словарем:

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

В результате получим:

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
 (74, 'Europe/London'),
 (191, 'America/Denver'),
 (382, 'America/Los_Angeles'),
 (400, 'America/Chicago'),
 (521, ''),
 (1251, 'America/New_York')]
```

Пошарив в стандартной библиотеке Python, можно найти класс `collections.Counter`, который позволяет решить задачу гораздо проще:

```
In [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

Подсчет часовых поясов с помощью pandas

Основной в библиотеке pandas является структура данных *DataFrame*, которую можно представлять себе как таблицу. Создать экземпляр DataFrame из исходного набора записей просто:

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)
```

```
In [27]: frame
Out[27]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_  120 non-null float64
a             3440 non-null object
al            3094 non-null object
c             2919 non-null object
cy            2919 non-null object
g             3440 non-null object
gr            2919 non-null object
h             3440 non-null object
hc            3440 non-null float64
hh            3440 non-null object
kw            93 non-null object
l             3440 non-null object
ll            2919 non-null object
nk            3440 non-null float64
r             3440 non-null object
t             3440 non-null float64
tz            3440 non-null object
u             3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB
```

```
In [28]: frame['tz'][:10]
Out[28]:
0  America/New_York
1    America/Denver
2  America/New_York
3 America/Sao_Paulo
4  America/New_York
5  America/New_York
6     Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

На выходе по запросу frame мы видим *сводное представление*, которое показывается для больших объектов DataFrame. Затем можно воспользоваться методом value_counts объекта Series:

```
In [29]: tz_counts = frame['tz'].value_counts()
```

```
In [30]: tz_counts[:10]
Out[30]:
```

```
America/New_York       1251
                        521
```

```
America/Chicago          400
America/Los_Angeles      382
America/Denver           191
Europe/London             74
Asia/Tokyo                37
Pacific/Honolulu          36
Europe/Madrid             35
America/Sao_Paulo         33
Name: tz, dtype: int64
```

Эти данные можно визуализировать с помощью библиотеки matplotlib. Возможно, придется слегка подправить их, подставив какое-нибудь значение вместо неизвестных и отсутствующих часовых поясов. Заменить отсутствующие значения позволяет функция `fillna`, а пустые строки можно заменить с помощью булева индексированием массива:

```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'

In [33]: tz_counts = clean_tz.value_counts()

In [34]: tz_counts[:10]
Out[34]:
America/New_York       1251
Unknown                 521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Missing                 120
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
```

Теперь можно воспользоваться пакетом seaborn для построения горизонтальной столбчатой диаграммы (результат показан на рис. 14.1):

```
In [36]: import seaborn as sns

In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

Поле `a` содержит информацию о браузере, устройстве или приложении, выполнившем сокращение URL:

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [41]: frame['a'][51][:50]  # длинная строка
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

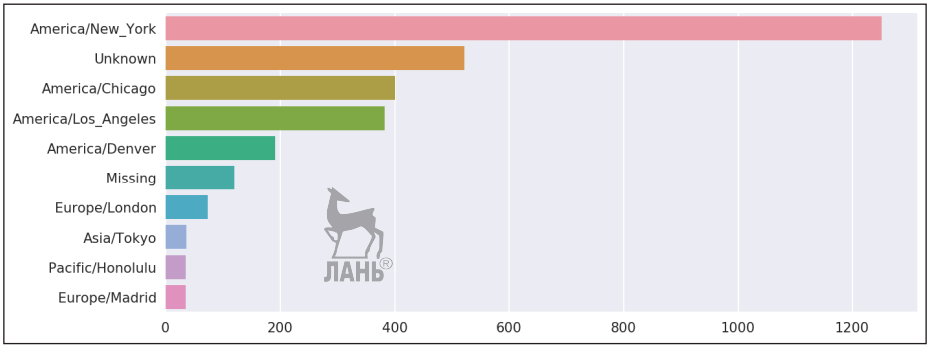


Рис. 14.1. Первые десять часовых поясов из набора данных 1.usa.gov

Выделение всей интересной информации из таких строк «пользовательских агентов» поначалу может показаться пугающей задачей. Одна из возможных стратегий – вырезать из строки первую лексему (грубо описывающую возможности браузера) и представить поведение пользователя в другом разрезе:

```
In [42]: results = Series([x.split()[0] for x in frame.a.dropna()])

In [43]: results[:5]
Out[43]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
Out[44]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
dtype: int64
```

Предположим теперь, что требуется разделить пользователей в первых десяти часовых поясах на работающих в Windows и всех прочих. Упростим задачу, предположив, что пользователь работает в Windows, если строка агента содержит подстроку `'Windows'`. Но строка агента не всегда присутствует, поэтому записи, в которых ее нет, я исключу:

```
In [45]: cframe = frame[frame.a.notnull()]
```

Мы хотим вычислить значение, показывающее, относится строка к пользователю Windows или нет:

```
In [47]: cframe = np.where(cframe['a'].str.contains('Windows'),
    .....:                 'Windows', 'Not Windows')

In [48]: cframe['os'][:5]
Out[48]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: os, dtype: object
```

Затем мы можем сгруппировать данные по часовому поясу и только что сформированному столбцу с типом операционной системы:

```
In [49]: by_tz_os = cframe.groupby(['tz', operating_system])
```

Групповые счетчики по аналогии с рассмотренной выше функцией `value_counts` можно вычислить с помощью функции `size`, а затем преобразовать результат в таблицу с помощью `unstack`:

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [51]: agg_counts[:10]
Out[51]:
a                               Not Windows  Windows
tz
                                      245.0    276.0
Africa/Cairo                            0.0      3.0
Africa/Casablanca                       0.0      1.0
Africa/Ceuta                            0.0      2.0
Africa/Johannesburg                     0.0      1.0
Africa/Lusaka                           0.0      1.0
America/Anchorage                       4.0      1.0
America/Argentina/Buenos_Aires          1.0      0.0
America/Argentina/Cordoba               0.0      1.0
America/Argentina/Mendoza               0.0      1.0
```

Наконец, выберем из полученной таблицы первые десять часовых поясов. Для этого я построю массив косвенных индексов `agg_counts` по счетчикам строк:

```
# Нужен для сортировки в порядке возрастания
In [52]: indexer = agg_counts.sum(1).argsort()

In [53]: indexer[:10]
Out[53]:
tz
                                 24
Africa/Cairo                     20
```

```
Africa/Casablanca                21
Africa/Ceuta                     92
Africa/Johannesburg              87
Africa/Lusaka                    53
America/Anchorage                54
America/Argentina/Buenos_Aires   57
America/Argentina/Cordoba        26
America/Argentina/Mendoza        55
dtype: int64
```

Затем с помощью take расположу строки в порядке, определяемом этим индексом, и оставлю только последние десять (с наибольшими значениями):

```
In [54]: count_subset = agg_counts.take(indexer)[-10:]

In [55]: count_subset
Out[55]:
a                   Not Windows  Windows
tz
America/Sao_Paulo           13.0     20.0
Europe/Madrid               16.0     19.0
Pacific/Honolulu             0.0     36.0
Asia/Tokyo                   2.0     35.0
Europe/London               43.0     31.0
America/Denver             132.0     59.0
America/Los_Angeles        130.0     252.0
America/Chicago            115.0     285.0
                           245.0     276.0
America/New_York           339.0     912.0
```

В pandas имеется вспомогательный метод nlargest, который делает то же самое:

```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York       1251.0
                        521.0
America/Chicago         400.0
America/Los_Angeles     382.0
America/Denver          191.0
Europe/London            74.0
Asia/Tokyo               37.0
Pacific/Honolulu         36.0
Europe/Madrid            35.0
America/Sao_Paulo        33.0
dtype: float64
```

Теперь можно построить столбчатую диаграмму, как и в предыдущем примере. Только на этот раз я сделаю ее штабельной, передав дополнительный аргумент функции seaborn barplot (см. рис. 14.2):

```
# Реорганизовать данные для построения графика
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
                tz           os   total
0  America/Sao_Paulo  Not Windows    13.0
1  America/Sao_Paulo      Windows    20.0
2      Europe/Madrid  Not Windows    16.0
3      Europe/Madrid      Windows    19.0
4   Pacific/Honolulu  Not Windows     0.0
5   Pacific/Honolulu      Windows    36.0
6         Asia/Tokyo  Not Windows     2.0
7         Asia/Tokyo      Windows    35.0
8      Europe/London  Not Windows    43.0
9      Europe/London Windows 31.0
In [62]: sns.barplot(x='total', y='tz', hue='os', data=count_subset)
```

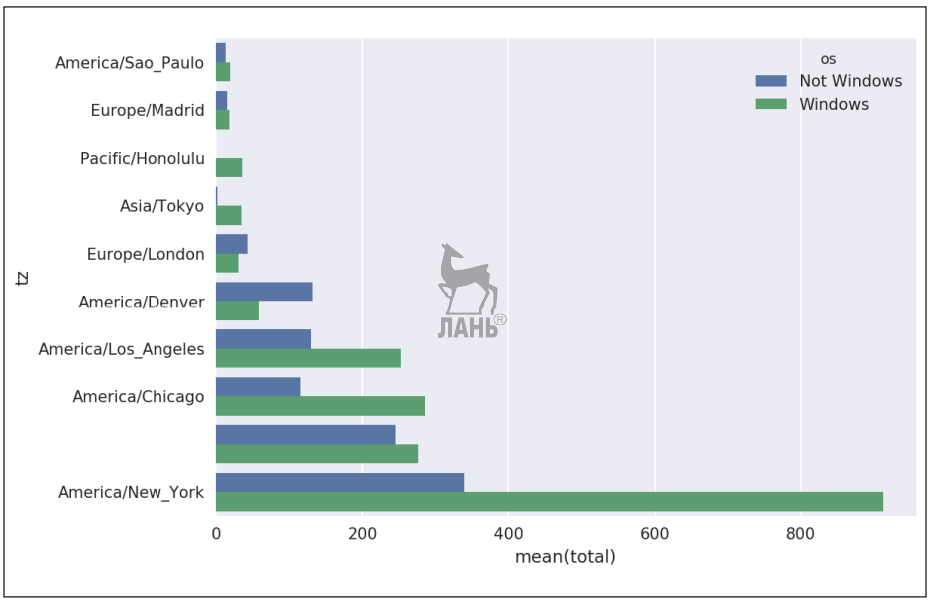


Рис. 14.2. Первые десять часовых поясов с выделением пользователей Windows и прочих

Из этой диаграммы трудно понять, какова процентная доля пользователей Windows в каждой группе, поэтому нормируем процент в группе, так чтобы в сумме получилось 1:

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

Новая диаграмма показана на рис. 14.3:

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os', data=results)
```

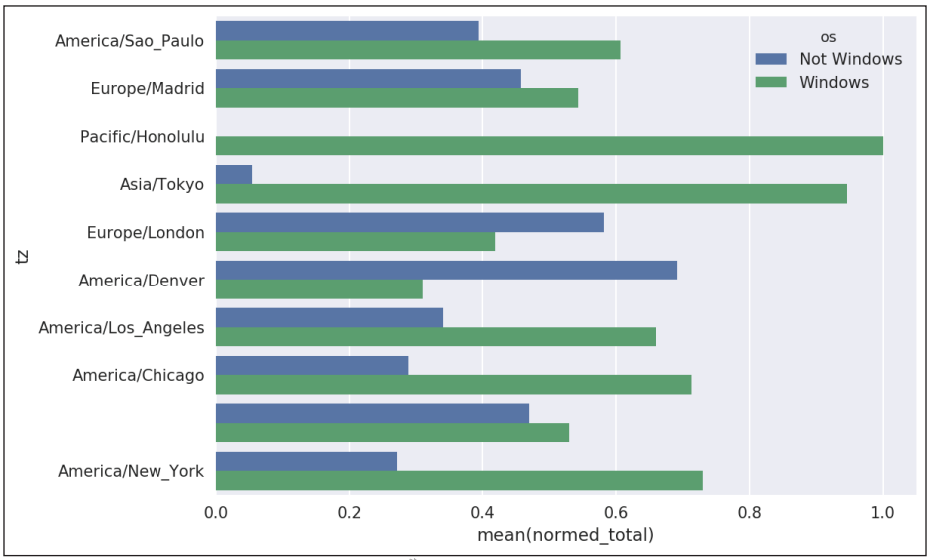


Рис. 14.3. Процентная доля пользователей Windows и прочих в первых десяти часовых поясах

Нормированную сумму можно было бы вычислить эффективнее, воспользовавшись методом `transform` объекта `groupby`:

```
In [66]: g = count_subset.groupby('tz')

In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2. Набор данных MovieLens 1M

Исследовательская группа GroupLens Research (https://grouplens.org/datasets/movielens/) предлагает несколько наборов данных о рейтингах фильмов, проставленных пользователями сайта MovieLens в конце 1990-х – начале 2000-х годов. Наборы содержат рейтинги фильмов, метаданные о фильмах (жанр и год выхода) и демографические данные о пользователях (возраст, почтовый индекс, пол и род занятий). Такие данные часто представляют интерес для разработки систем рекомендования, основанных на алгоритмах

машинного обучения. И хотя в этой книге методы машинного обучения не рассматриваются, я все же покажу, как формировать продольные и поперечные разрезы таких наборов данных с целью привести их к нужному виду.

Набор MovieLens 1M содержит 1 000 000 рейтингов 4000 фильмов, проставленных 6000 пользователей. Данные распределены по трем таблицам: рейтинги, информация о пользователях и информация о фильмах. После распаковки zip-файла каждую таблицу можно загрузить в отдельный объект DataFrame с помощью метода `pandas.read_table`:

```
import pandas as pd

# Уменьшить объем вывода
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('datasets/movielens/users.dat', sep='::', header=None,
                      names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::', header=None,
                        names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::', header=None,
                       names=mnames
```

Проверить, все ли прошло удачно, можно, посмотрев на первые несколько строк каждого DataFrame с помощью встроенного в Python синтаксиса вырезания:

```
In [69]: users[:5]
Out[69]:
   user_id  gender  age  occupation    zip
0        1       F    1          10  48067
1        2       M   56          16  70072
2        3       M   25          15  55117
3        4       M   45           7  02460
4        5       M   25          20  55455

In [70]: ratings[:5]
Out[70]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
3        1      3408       4  978300275
4        1      2355       5  978824291

In [71]: movies[:5]
Out[71]:
  movie_id                        title                         genres
0        1             Toy Story (1995)   Animation|Children's|Comedy
```

```
1        2               Jumanji (1995)      Adventure|Children's|Fantasy
2        3      Grumpier Old Men (1995)                     Comedy|Romance
3        4     Waiting to Exhale (1995)                       Comedy|Drama
4        5 Father of the Bride Part II (1995)                       Comedy
```

```
In [72]: ratings
Out[72]:
           user_id  movie_id  rating  timestamp
0                1      1193       5  978300760
1                1       661       3  978302109
2                1       914       3  978301968
3                1      3408       4  978300275
4                1      2355       5  978824291
...            ...       ...     ...        ...
1000204       6040      1091       1  956716541
1000205       6040      1094       5  956704887
1000206       6040       562       5  956704746
1000207       6040      1096       4  956715648
1000208       6040      1097       4  956715569
[1000209 rows x 4 columns]
```

Отметим, что возраст и род занятий кодируются целыми числами, а расшифровка приведена в прилагаемом к набору данных файлу `README`. Анализ данных, хранящихся в трех таблицах, – непростая задача. Пусть, например, требуется вычислить средние рейтинги для конкретного фильма в разрезе пола и возраста. Как мы увидим, это гораздо легче сделать, если предварительно свести все данные в одну таблицу. Применяя функцию `merge` из библиотеки pandas, мы сначала объединим `ratings` с `users`, а затем результат – с `movies`. Pandas определяет, по каким столбцам объединять (или *соединять*), ориентируясь на совпадение имен:

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)
```

```
In [74]: data
Out[74]:
       user_id  movie_id rating  timestamp gender age occupation    zip \
0            1      1193      5  978300760      F   1         10  48067
1            2      1193      5  978298413      M  56         16  70072
2           12      1193      4  978220179      M  25         12  32793
3           15      1193      4  978199279      M  25          7  22903
4           17      1193      5  978158471      M  50          1  95350
...        ...       ...    ...        ...    ... ...        ...    ...
1000204   5949      2198      5  958846401      M  18         17  47901
1000205   5675      2703      3  976029116      M  35         14  30030
1000206   5780      2845      1  958153068      M  18         17  92886
1000207   5851      3607      5  957756608      F  18         20  55410
1000208   5938      2909      4  957273353      M  25          1  35401
                                    title                        genres
0          One Flew Over the Cuckoo's Nest (1975)                 Drama
```

```
1          One Flew Over the Cuckoo's Nest (1975)                 Drama
2          One Flew Over the Cuckoo's Nest (1975)                 Drama
3          One Flew Over the Cuckoo's Nest (1975)                 Drama
4          One Flew Over the Cuckoo's Nest (1975)                 Drama
...                                          ...                   ...
1000204                        Modulations (1998)          Documentary
1000205                      Broken Vessels (1998)                 Drama
1000206                          White Boys (1999)                 Drama
1000207                 One Little Indian (1973) Comedy|Drama|Western
1000208 Five Wives, Three Secretaries and Me (1998) Documentary
[1000209 rows x 10 columns]

In [75]: data.iloc[0]
Out[75]:
user_id                                       1
movie_id                                   1193
rating                                        5
timestamp                             978300760
gender                                        F
age                                           1
occupation                                   10
zip                                       48067
title       One Flew Over the Cuckoo's Nest (1975)
genres                                    Drama
Name: 0, dtype: object
```

Чтобы получить средние рейтинги каждого фильма по группам зрителей одного пола, воспользуемся методом pivot_table:

```
In [76]: mean_ratings = data.pivot_table('rating', index='title',
   ....:                                 columns='gender', aggfunc='mean')
In [77]: mean_ratings[:5]
Out[77]:
gender                                F         M
title
$1,000,000 Duck (1971)         3.375000  2.761905
'Night Mother (1986)           3.388889  3.352941
'Til There Was You (1997)      2.675676  2.733333
'burbs, The (1989)             2.793478  2.962085
...And Justice for All (1979)   3.828571  3.689024
```

В результате получается еще один объект DataFrame, содержащий средние рейтинги, в котором метками строк («индексом») являются названия фильмов, а метками столбцов – обозначения полов. Сначала я оставлю только фильмы, получившие не менее 250 оценок (число выбрано совершенно произвольно); для этого сгруппирую данные по названию и с помощью метода size() получу объект Series, содержащий размеры групп для каждого наименования:

```
In [78]: ratings_by_title = data.groupby('title').size()

In [79]: ratings_by_title[:10]
Out[79]:
title
$1,000,000 Duck (1971)               37
'Night Mother (1986)                 70
'Til There Was You (1997)            52
'burbs, The (1989)                  303
...And Justice for All (1979)       199
1-900 (1994)                          2
10 Things I Hate About You (1999)   700
101 Dalmatians (1961)               565
101 Dalmatians (1996)               364
12 Angry Men (1957)                 616
dtype: int64

In [80]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [81]: active_titles
Out[81]:
Index(['burbs, The (1989), 10 Things I Hate About You (1999),
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Men (1957)',
       '13th Warrior, The (1999)', '2 Days in the Valley (1996)',
       '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey (1968)',
       '2010 (1984)',
       ...
       'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1998)',
       'Young Frankenstein (1974)', 'Young Guns (1988)',
       'Young Guns II (1990)', 'Young Sherlock Holmes (1985)',
       'Zero Effect (1998)', 'eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

Затем для отбора строк из приведенного выше объекта `mean_ratings` воспользуемся индексом фильмов, получивших не менее 250 оценок:

```
# Выбрать строки по индексу
In [82]: mean_ratings = mean_ratings.loc[active_titles]

In [83]: mean_ratings
Out[83]:
Gender                                F         M
title
'burbs, The (1989)                 2.793478  2.962085
10 Things I Hate About You (1999)  3.646552  3.311966
101 Dalmatians (1961)              3.791444  3.500000
101 Dalmatians (1996)              3.240000  2.911215
12 Angry Men (1957)                4.184397  4.328421
...                                     ...       ...
Young Guns (1988)                  3.371795  3.425620
```

```
Young Guns II (1990)                2.934783  2.904025
Young Sherlock Holmes (1985)        3.514706  3.363344
Zero Effect (1998)                  3.864407  3.723140
eXistenZ (1999)                     3.098592  3.289086
[1216 rows x 2 columns]
```

Чтобы найти фильмы, оказавшиеся на первом месте у женщин, мы можем отсортировать результат по столбцу F в порядке убывания:

```
In [85]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)

In [86]: top_female_ratings[:10]
Out[86]:
gender                                                F         M
Close Shave, A (1995)                          4.644444  4.473795
Wrong Trousers, The (1993)                     4.588235  4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)  4.572650  4.464589
Wallace & Gromit: The Best of Aardman Animation (1996)  4.563107  4.385075
Schindler's List (1993)                        4.562602  4.491415
Shawshank Redemption, The (1994)               4.539075  4.560625
Grand Day Out, A (1992)                        4.537879  4.293255
To Kill a Mockingbird (1962)                   4.536667  4.372611
Creature Comforts (1990)                       4.513889  4.272277
Usual Suspects, The (1995)                     4.513317  4.518248
```

Измерение несогласия в оценках

Допустим, мы хотим найти фильмы, по которым мужчины и женщины сильнее всего разошлись в оценках. Для этого можно добавить столбец mean_ratings, содержащий разность средних, а затем отсортировать по нему:

```
In [87]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Сортировка по столбцу 'diff' дает фильмы с наибольшей разностью оценок, которые больше нравятся женщинам:

```
In [88]: sorted_by_diff = mean_ratings.sort_index(by='diff')

In [89]: sorted_by_diff[:10]
Out[89]:
gender                                        F         M       diff
Dirty Dancing (1987)                   3.790378  2.959596  -0.830782
Jumpin' Jack Flash (1986)              3.254717  2.578358  -0.676359
Grease (1978)                          3.975265  3.367041  -0.608224
Little Women (1994)                    3.870588  3.321739  -0.548849
Steel Magnolias (1989)                 3.901734  3.365957  -0.535777
Anastasia (1997)                       3.800000  3.281609  -0.518391
Rocky Horror Picture Show, The (1975)  3.673016  3.160131  -0.512885
Color Purple, The (1985)               4.158192  3.659341  -0.498851
Age of Innocence, The (1993)           3.827068  3.339506  -0.487561
Free Willy (1993)                      2.921348  2.438776  -0.482573
```

Изменив порядок строк на противоположный и снова отобрав первые 15 строк, мы получим фильмы, которым мужчины поставили высокие, а женщины – низкие оценки:

```
# Изменяем порядок строк на противоположный и отбираем первые 1 строк
In [90]: sorted_by_diff[::-1][:10]
Out[90]:
Gender                                     F         M      diff
Good, The Bad and The Ugly, The (1966)  3.494949  4.221300  0.726351
Kentucky Fried Movie, The (1977)        2.878788  3.555147  0.676359
Dumb & Dumber (1994)                    2.697987  3.336595  0.638608
Longest Day, The (1962)                 3.411765  4.031447  0.619682
Cable Guy, The (1996)                   2.250000  2.863787  0.613787
Evil Dead II (Dead By Dawn) (1987)      3.297297  3.909283  0.611985
Hidden, The (1987)                      3.137931  3.745098  0.607167
Rocky III (1982)                        2.361702  2.943503  0.581801
Caddyshack (1980)                       3.396135  3.969737  0.573602
For a Few Dollars More (1965)           3.409091  3.953795  0.544704
```

А теперь допустим, что нас интересуют фильмы, вызвавшие наибольшее разногласие у зрителей независимо от пола. Разногласие можно изменить с помощью дисперсии или стандартного отклонения оценок:

```
# Стандартное отклонение оценок, сгруппированных по названию
In [91]: rating_std_by_title = data.groupby('title')['rating'].std()

# Оставляем только active_titles
In [92]: rating_std_by_title = rating_std_by_title.ix[active_titles]

# Упорядочиваем Series по значению в порядке убывания
In [93]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[93]:
title
Dumb & Dumber (1994)                     1.321333
Blair Witch Project, The (1999)          1.316368
Natural Born Killers (1994)              1.307198
Tank Girl (1995)                         1.277695
Rocky Horror Picture Show, The (1975)    1.260177
Eyes Wide Shut (1999)                    1.259624
Evita (1996)                             1.253631
Billy Madison (1995)                     1.249970
Fear and Loathing in Las Vegas (1998)    1.246408
Bicentennial Man (1999)                  1.245533
Name: rating, dtype: float64
```

Вы, наверное, обратили внимание, что жанры фильма разделяются вертикальной чертой (|). Чтобы провести анализ по жанрам, прошлось бы проделать дополнительную работу по преобразованию данных в более удобную форму.

14.3. Имена, которые давали детям в США за период с 1880 по 2010 год

Управление социального обеспечения США выложило в Сеть данные о частоте встречаемости детских имен за период с 1880 года по настоящее время. Хэдли Уикхэм (Hadley Wickham), автор нескольких популярных пакетов для R, часто использует этот пример для иллюстрации манипуляций с данными в R.

Чтобы загрузить этот набор, данные придется немного переформатировать, получившийся в результате объект DataFrame выглядит так:

```
In [4]: names.head(10)
Out[4]:
         name  sex births  year
0        Mary    F   7065  1880
1        Anna    F   2604  1880
2        Emma    F   2003  1880
3   Elizabeth    F   1939  1880
4      Minnie    F   1746  1880
5    Margaret    F   1578  1880
6         Ida    F   1472  1880
7       Alice    F   1414  1880
8      Bertha    F   1320  1880
9       Sarah    F   1288  1880
```


С этим набором можно проделать много интересного:

- наглядно представить долю младенцев, получавших данное имя (совпадающее с вашим или какое-нибудь другое) за весь период времени;
- определить относительный ранг имени;
- найти самые популярные в каждом году имена или имена, для которых фиксировалось наибольшее увеличение или уменьшение частоты;
- проанализировать тенденции выбора имен: количество гласных и согласных, длину, общее разнообразие, изменение в написании, первые и последние буквы;
- проанализировать внешние источники тенденций: библейские имена, имена знаменитостей, демографические изменения.

С помощью рассмотренных в этой книге инструментов бóльшая часть этих задач решается без особого труда, и я это кратко продемонстрирую.

На момент написания данной книги управление социального обеспечения США представило данные в виде набора файлов (по одному на каждый год), в которых указано общее число родившихся младенцев для каждой пары пол/имя. Архив этих файлов находится по адресу http://www.ssa.gov/oact/babynames/limits.html.

Если со временем адрес этой страницы поменяется, найти ее, скорее всего, можно будет с помощью поисковой системы. Загрузив и распаковав файл `names.zip`, вы получите каталог, содержащий файлы с именами вида `yob1880.txt`.

С помощью команды UNIX head я могу вывести первые десять строк каждого файла (в Windows можно воспользоваться командой more или открыть файл в текстовом редакторе):

```
In [94]: !head -n 10 names/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

Поскольку поля разделены запятыми, файл можно загрузить в объект DataFrame методом pandas.read_csv:

```
In [95]: import pandas as pd

In [96]: names1880 = pd.read_csv('datasets/babynames/yob1880.txt',
   ....:                         names=['name', 'sex', 'births'])
In [97]: names1880
Out[97]:
          name sex  births
0         Mary   F    7065
1         Anna   F    2604
2         Emma   F    2003
3    Elizabeth   F    1939
4       Minnie   F    1746
...         ... ..     ...
1995    Woodie   M       5
1996    Worthy   M       5
1997    Wright   M       5
1998      York   M       5
1999 Zachariah   M       5
[2000 rows x 3 columns]
```

В эти файлы включены только имена, которыми были названы не менее пяти младенцев в году, поэтому для простоты сумму значений в столбце sex можно считать общим числом родившихся в данном году младенцев:

```
In [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F        90993
M       110493
Name: births, dtype: int64
```

Поскольку в каждом файле находятся данные только за один год, то первое, что нужно сделать, – собрать все данные в единый объект DataFrame и добавить поле year. Это легко сделать методом pandas.concat:

```
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'names/yob%d.txt' % year
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Собрать все данные в один объект DataFrame
names = pd.concat(pieces, ignore_index=True)
```

Обратим внимание на два момента. Во-первых, напомним, что concat по умолчанию объединяет объекты DataFrame построчно. Во-вторых, следует задать параметр ignore_index=True, потому что нам неинтересно сохранять исходные номера строк, прочитанных методом read_csv. Таким образом, мы получили очень большой DataFrame, содержащий данные обо всех именах.

```
In [100]: names
Out[100]:
              name sex  births  year
0             Mary   F    7065  1880
1             Anna   F    2604  1880
2             Emma   F    2003  1880
3        Elizabeth   F    1939  1880
4           Minnie   F    1746  1880
...            ...  ..     ...   ...
1690779    Zymaire   M       5  2010
1690780     Zyonne   M       5  2010
1690781  Zyquarius   M       5  2010
1690782      Zyran   M       5  2010
1690783      Zzyzx   M       5  2010
[1690784 rows x 4 columns]
```

Имея эти данные, мы уже можем приступить к агрегированию на уровне года и пола, используя метод groupby или pivot_table (см. рис. 14.4):

```
In [101]: total_births = names.pivot_table('births', rows='year',

   .....:                                  cols='sex', aggfunc=sum)

In [102]: total_births.tail()
Out[102]:
sex         F        M
```

```
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382
```

```
In [103]: total_births.plot(title='Total births by sex and year')
```

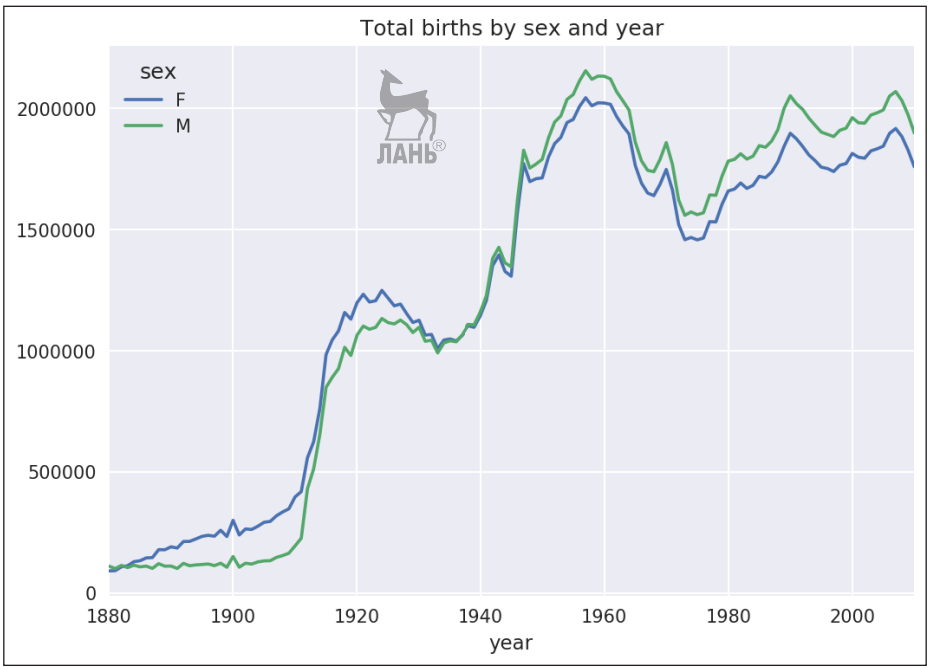


Рис. 14.4. Общее количество родившихся в зависимости от пола и года рождения

Далее вставим столбец `prop`, содержащий долю младенцев, получивших данное имя, относительно общего числа родившихся. Значение `prop`, равное `0.02`, означает, что данное имя получили 2 из 100 младенцев. Затем сгруппируем данные по году и полу и добавим в каждую группу новый столбец:

```
def add_prop(group):
    births = group.births.astype(float)

    group['prop'] = births / births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

Получившийся в результате пополненный набор данных состоит из таких столбцов:

```
In [105]: names
Out[105]:
             name sex  births  year      prop
0            Mary   F    7065  1880  0.077643
1            Anna   F    2604  1880  0.028618
2            Emma   F    2003  1880  0.022013
3       Elizabeth   F    1939  1880  0.021309
4          Minnie   F    1746  1880  0.019188
...           ...  ..     ...   ...       ...
1690779   Zymaire   M       5  2010  0.000003
1690780    Zyonne   M       5  2010  0.000003
1690781  Zyquarius  M       5  2010  0.000003
1690782      Zyran  M       5  2010  0.000003
1690783      Zzyzx  M       5  2010  0.000003
[1690784 rows x 5 columns]
```

При выполнении такой операции группировки часто бывает полезно произвести проверку разумности результата, например удостовериться, что сумма значений в столбце `prop` по всем группам равна 1.

```
In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year sex
1880   F    1.0
       M    1.0
1881   F    1.0
       M    1.0
1882   F    1.0
            ...
2008   M    1.0
2009   F    1.0
       M    1.0
2010   F    1.0
       M    1.0
Name: prop, Length: 262, dtype: float64
```

Далее я извлеку подмножество данных, чтобы упростить последующий анализ: первые 1000 имен для каждой комбинации пола и года. Это еще одна групповая операция:

```
def get_top1000(group):
    return group.sort_index(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# Удалить групповой индекс, он больше не нужен
top1000.reset_index(inplace=True, drop=True)
```

Если вы предпочитаете все делать самостоятельно, то можно поступить и так:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_index(by='births', ascending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

Теперь результирующий набор стал заметно меньше:

```
In [108]: top1000
Out[108]:
             name sex  births  year      prop
0            Mary   F    7065  1880  0.077643
1            Anna   F    2604  1880  0.028618
2            Emma   F    2003  1880  0.022013
3       Elizabeth   F    1939  1880  0.021309
4          Minnie   F    1746  1880  0.019188
...           ...  ..     ...   ...       ...
261872     Camilo   M     194  2010  0.000102
261873     Destin   M     194  2010  0.000102
261874     Jaquan   M     194  2010  0.000102
261875     Jaydan   M     194  2010  0.000102
261876     Maxton   M     193  2010  0.000102
[261877 rows x 5 columns]
```

Это набор, содержащий первые 1000 записей. Его мы и будем использовать для исследования данных в дальнейшем.

Анализ тенденций в выборе имен

Имея полный набор данных и первые 1000 записей, мы можем приступить к анализу различных интересных тенденций. Для начала решим простую задачу: разобьем набор `top1000` на части, относящиеся к мальчикам и девочкам.

```
In [109]: boys = top1000[top1000.sex == 'M']

In [110]: girls = top1000[top1000.sex == 'F']
```

Можно нанести на график простые временные ряды, например количество Джонов и Мэри в каждом году, но для этого потребуется предварительное переформатирование. Сформируем сводную таблицу, в которой представлено общее число родившихся по годам и именам:

```
In [111]: total_births = top1000.pivot_table('births', index='year',
   .....:                                     columns='name',
   .....:                                     aggfunc=sum)
```

Теперь можно нанести на график несколько имен, воспользовавшись методом `plot` объекта DataFrame (результат показан на рис. 14.5):

```
In [112]: total_births.info()
Out[112]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
```

```
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB
```

```
In [113]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]
In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
   .....:             title="Number of births per year")
```

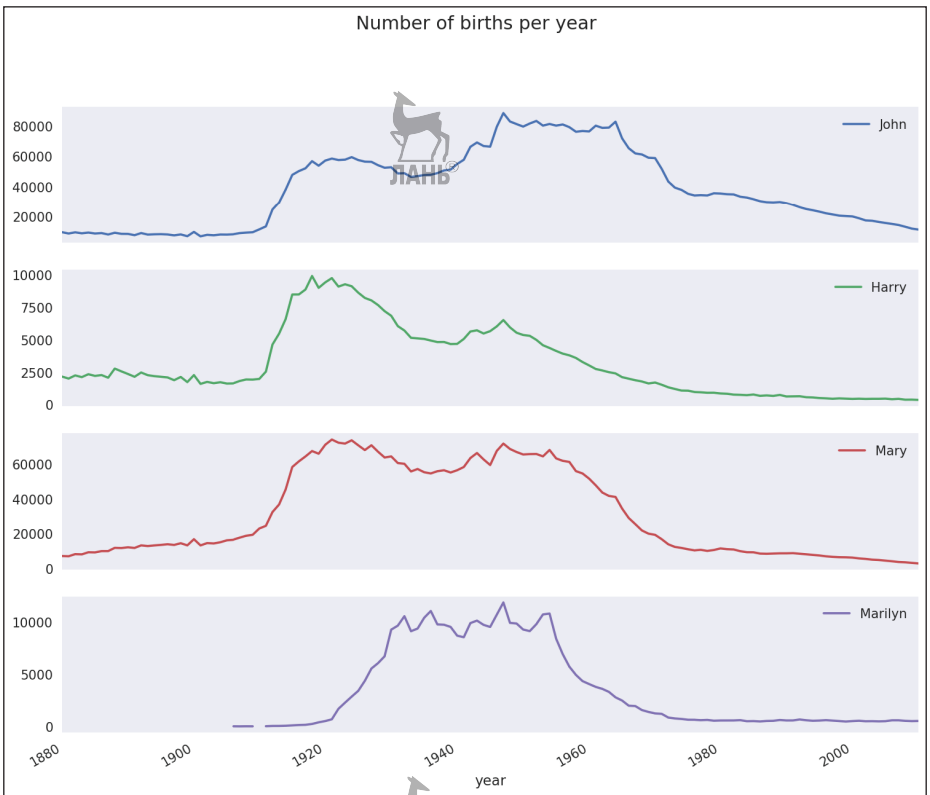


Рис. 14.5. Распределение нескольких имен мальчиков и девочек по годам

Глядя на рисунок, можно сделать вывод, что эти имена в Америке вышли из моды. Но на самом деле картина несколько сложнее, как станет ясно в следующем разделе.

Измерение роста разнообразия имен

Убывание кривых на рисунках выше можно объяснить тем, что меньше родителей стали выбирать такие распространенные имена. Эту гипотезу можно проверить и подтвердить имеющимися данными. Один из возможных показателей – доля родившихся в наборе 1000 самых популярных имен, который я агрегирую по году и полу (результат показан на рис. 14.6):

```
In [390]: table = top1000.pivot_table('prop', rows='year',
   .....:                             cols='sex', aggfunc=sum)

In [391]: table.plot(title='Sum of table1000.prop by year and sex',
   .....:        yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))
```

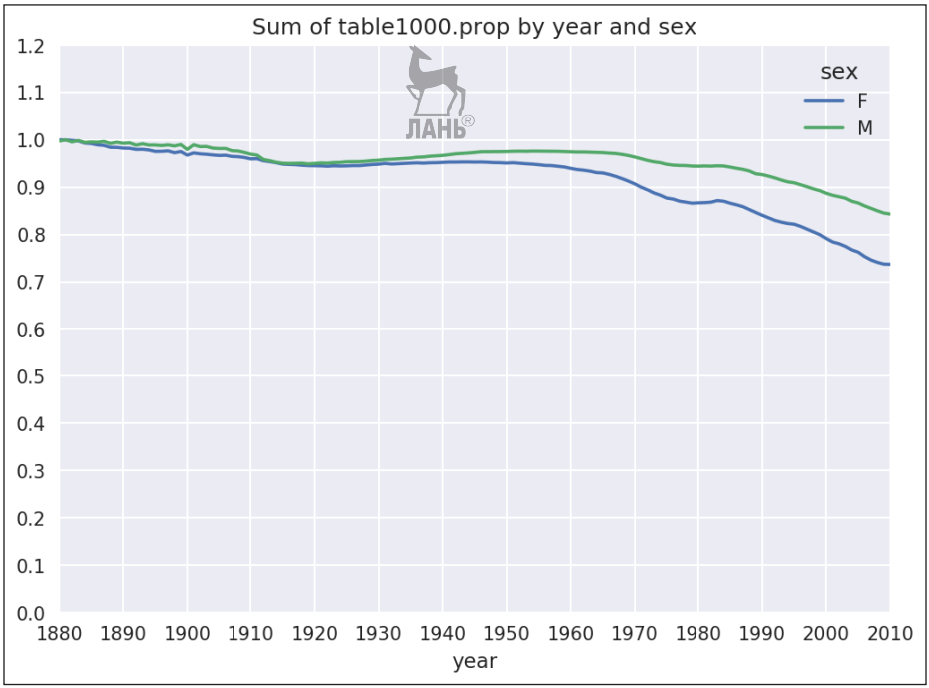


Рис. 14.6. Доля родившихся мальчиков и девочек, представленных в первой тысяче имен

Действительно, похоже, что разнообразие имен растет (доля в первой тысяче падает). Другой интересный показатель – количество различных имен среди первых 50 % родившихся, упорядоченное по популярности в порядке убывания. Вычислить его несколько сложнее. Рассмотрим только имена мальчиков, родившихся в 2010 году:

```
In [118]: df = boys[boys.year == 2010]

In [119]: df
Out[119]:
           name sex  births  year      prop
260877    Jacob   M   21875  2010  0.011523
260878    Ethan   M   17866  2010  0.009411
260879  Michael   M   17133  2010  0.009025
260880   Jayden   M   17030  2010  0.008971
260881  William   M   16870  2010  0.008887
```

```
...         ...  ..     ...   ...       ...
261872   Camilo   M     194  2010  0.000102
261873   Destin   M     194  2010  0.000102
261874   Jaquan   M     194  2010  0.000102
261875   Jaydan   M     194  2010  0.000102
261876   Maxton   M     193  2010  0.000102
[1000 rows x 5 columns]
```

После сортировки prop в порядке убывания мы хотим узнать, сколько популярных имен нужно взять, чтобы достичь 50 %. Можно написать для этого цикл for, но NumPy предлагает более хитроумный векторный подход. Если вычислить накопительные суммы cumsum массива prop, а затем вызвать метод searchsorted, то будет возвращена позиция в массиве накопительных сумм, в которую нужно было бы вставить 0.5, чтобы не нарушить порядок сортировки:

```
In [120]: prop_cumsum = df.sort_values(by='prop', ascending=False).prop.cumsum()

In [121]: prop_cumsum[:10]
Out[121]:
260877   0.011523
260878   0.020934
260879   0.029959
260880   0.038930
260881   0.047817
260882   0.056579
260883   0.065155
260884   0.073414
260885   0.081528
260886   0.089621
Name: prop, dtype: float64

In [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 116
```

Поскольку индексация массивов начинается с нуля, то нужно прибавить к результату 1 – получится 117. Заметим, что в 1900 году этот показатель был гораздо меньше:

```
In [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascending=False).prop.cumsum()

In [125]: in1900.values.searchsorted(0.5) + 1
Out[125]: 25
```

Теперь можно применить данную операцию к каждой комбинации года и пола; произведем группировку по этим полям с помощью метода groupby, а затем с помощью метода apply применим функцию, возвращающую счетчик для каждой группы:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(by='prop', ascending=False)
    return group.prop.cumsum().values.searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

В получившемся объекте DataFrame с именем `diversity` хранится два временных ряда, по одному для каждого пола, индексированные по году. Его можно исследовать в IPython и, как и раньше, нанести на график (рис. 14.7).

```
In [128]: diversity.head()
Out[128]:
sex    F   M
year
1880  38  14
1881  38  14
1882  38  15
1883  39  15
1884  39  16

In [129]: diversity.plot(title="Number of popular names in top 50%")
```

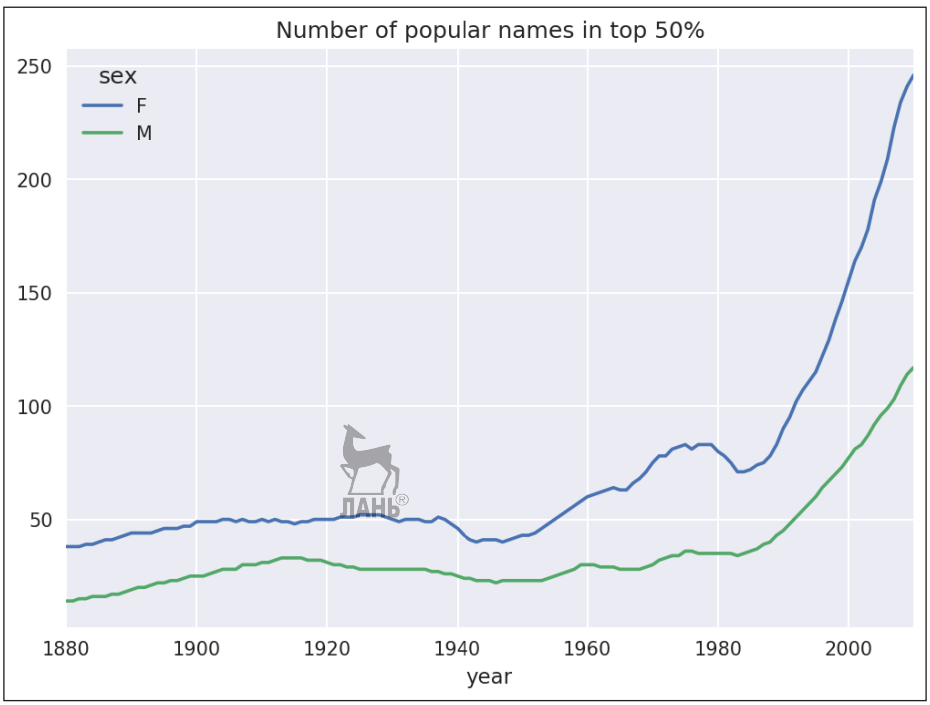


Рис. 14.7. График зависимости разнообразия имен от года

Как видим, девочкам всегда давали более разнообразные имена, чем мальчикам, и со временем эта тенденция проявляется все ярче. Анализ того, что именно является причиной разнообразия (например, растет число вариантов написания одного и того же имени), оставляю читателю.

Революция «последней буквы»

В 2007 году исследовательница детских имен Лаура Уоттенберг (Laura Wattenberg) отметила на своем сайте (http://www.babynamewizard.com), что распределение имен мальчиков по последней букве за последние 100 лет существенно изменилось. Чтобы убедиться в этом, я сначала агрегирую данные полного набора обо всех родившихся по году, полу и последней букве:

```
# извлекаем последнюю букву имени в столбце name
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', index=last_letters,
                          columns=['sex', 'year'], aggfunc=sum)
```

Затем выберу из всего периода три репрезентативных года и напечатаю первые несколько строк:

```
In [131]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [132]: subtable.head()
Out[132]:
sex                F                            M
year            1910      1960      2010     1910      1960      2010
last_letter
a             108376.0  691247.0  670605.0    977.0    5204.0   28438.0
b                  NaN     694.0     450.0    411.0    3912.0   38859.0
c                  5.0      49.0     946.0    482.0   15476.0   23125.0
d               6750.0    3729.0    2607.0  22111.0  262112.0   44398.0
e             133569.0  435013.0  313833.0  28655.0  178823.0  129012.0
```

Далее я пронормирую эту таблицу на общее число родившихся, чтобы вычислить новую таблицу, содержащую долю от общего количества родившихся для каждого пола и каждой последней буквы:

```
In [133]: subtable.sum()
Out[133]:
sex  year
F    1910   396416.0
     1960  2022062.0
     2010  1759010.0
M    1910   194198.0
     1960  2132588.0
     2010  1898382.0
dtype: float64
```

```
In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
Out[135]:
sex           F                            M
year       1910      1960      2010      1910      1960      2010
last_letter
a      0.273390  0.341853  0.381240  0.005031  0.002440  0.014980
b           NaN  0.000343  0.000256  0.002116  0.001834  0.020470
c      0.000013  0.000024  0.000538  0.002482  0.007257  0.012181
d      0.017028  0.001844  0.001482  0.113858  0.122908  0.023387
e      0.336941  0.215133  0.178415  0.147556  0.083853  0.067959
...         ...       ...       ...       ...       ...       ...
v           NaN  0.000060  0.000117  0.000113  0.000037  0.001434
w      0.000020  0.000031  0.001182  0.006329  0.007711  0.016148
x      0.000015  0.000037  0.000727  0.003965  0.001851  0.008614
y      0.110972  0.152569  0.116828  0.077349  0.160987  0.058168
z      0.002439  0.000659  0.000704  0.000170  0.000184  0.001831
[26 rows x 6 columns]
```


Зная доли букв, я теперь могу нарисовать столбчатые диаграммы для каждого пола, разбив их по годам (см. рис. 14.8).

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female',
                      legend=False)
```

Как видим, с 1960-х годов доля имен мальчиков, заканчивающихся буквой *n*, значительно возросла. Снова вернусь к созданной ранее полной таблице, пронормирую ее по году и полу, выберу некое подмножество букв для имен мальчиков и транспонирую, чтобы превратить каждый столбец во временной ряд:

```
In [138]: letter_prop = table / table.sum()

In [139]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T

In [140]: dny_ts.head()
Out[140]:

last_letter         d         n         y
year
1880         0.083055  0.153213  0.075760
1881         0.083247  0.153214  0.077451
1882         0.085340  0.149560  0.077537
1883         0.084066  0.151646  0.079144
1884         0.086120  0.149915  0.080405
```

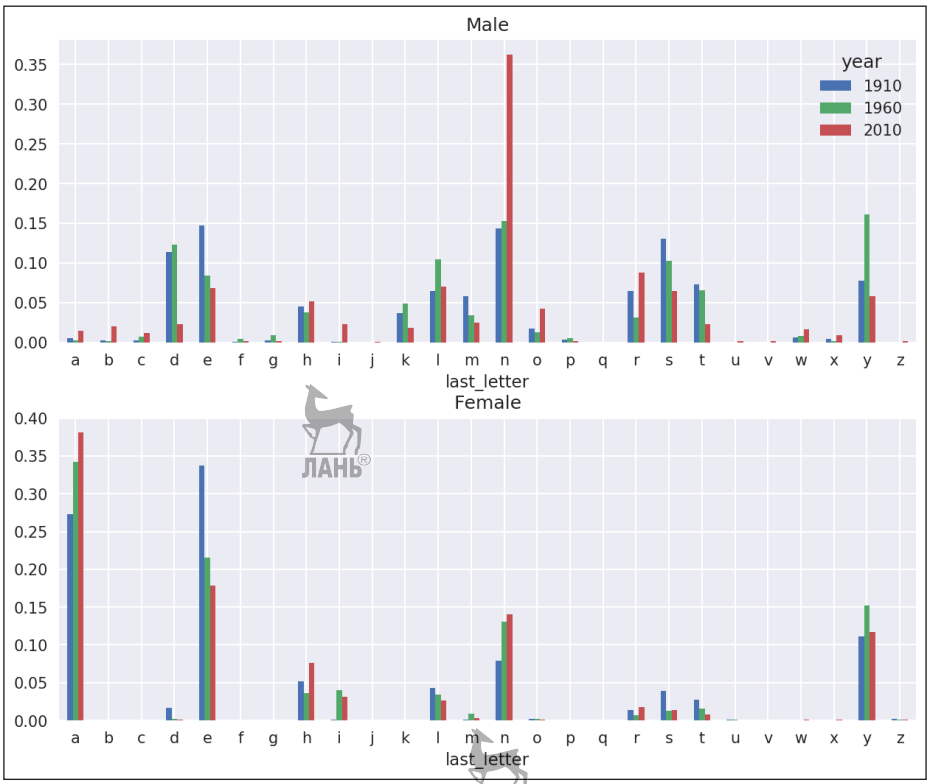


Рис. 14.8. Доли имен мальчиков и девочек, заканчивающихся на каждую букву

Имея этот объект DataFrame, содержащий временные ряды, я могу все тем же методом `plot` построить график изменения тенденций в зависимости от времени (рис. 14.9):

```
In [143]: dny_ts.plot()
```

Мужские имена, ставшие женскими, и наоборот

Еще одно интересное упражнение – изучить имена, которые раньше часто давали мальчикам, а затем «сменили пол». Возьмем, к примеру, имя Lesley или Leslie. По набору `top1000` вычисляю список имен, начинающихся с `'lesl'`:

```
In [144]: all_names = pd.Series(top1000.name.unique())

In [145]: lesley_like = all_names[all_names.str.lower().str.contains('lesl')]

In [146]: lesley_like
Out[146]:
632    Leslie
2294   Lesley
4262   Leslee
```

```
4728   Lesli
6103   Lesly
dtype: object
```

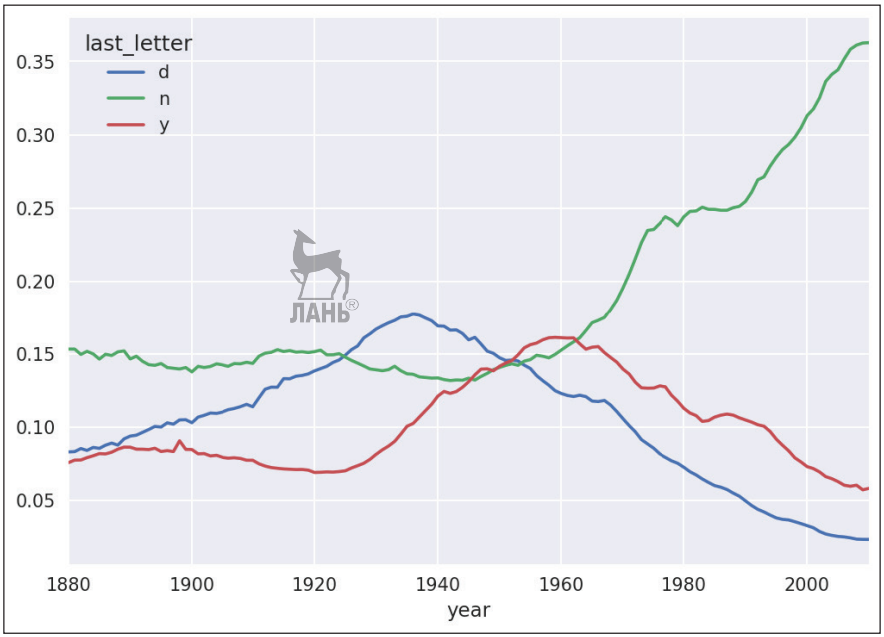


Рис. 14.9. Зависимость доли мальчиков с именами, заканчивающимися на буквы d, n, y, от времени

Далее можно оставить только эти имена и просуммировать количество родившихся, сгруппировав по имени, чтобы найти относительные частоты:

```
In [147]: filtered = top1000[top1000.name.isin(lesley_like)]

In [148]: filtered.groupby('name').births.sum()
Out[148]:
name
Leslee     1082
Lesley    35022
Lesli       929
Leslie   370429
Lesly     10067
Name: births, dtype: int64
```

Затем агрегируем по полу и году и нормируем в пределах каждого года:

```
In [149]: table = filtered.pivot_table('births', rows='year',
   .....:                              columns='sex', aggfunc='sum')

In [150]: table = table.div(table.sum(1), axis=0)
```

```
In [151]: table.tail()
Out[151]:
sex     F   M
year
2006  1.0 NaN
2007  1.0 NaN
2008  1.0 NaN
2009  1.0 NaN
2010  1.0 NaN
```

Наконец, нетрудно построить график распределения по полу в зависимости от времени (рис. 14.10).

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

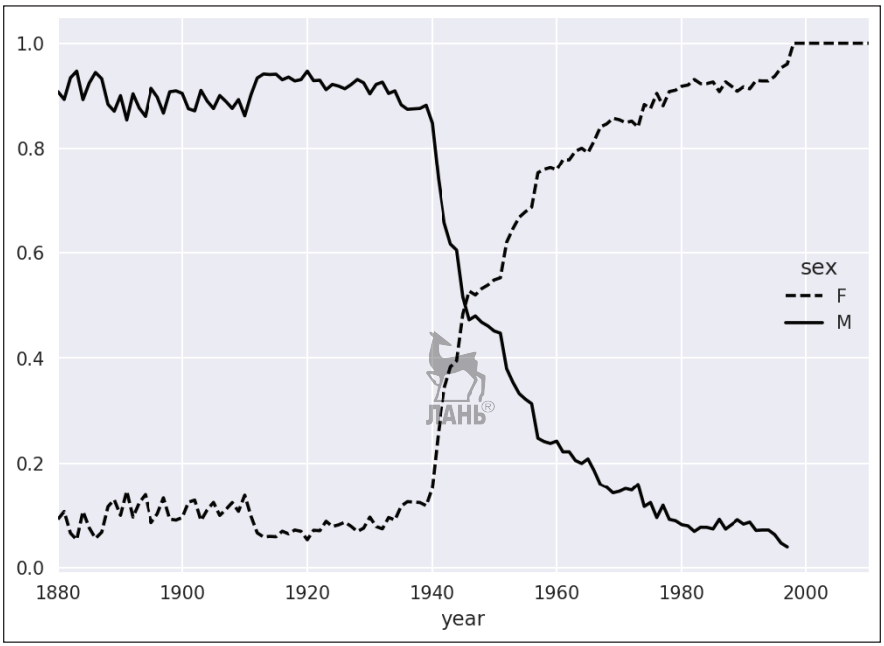


Рис. 14.10. Изменение во времени доли мальчиков и девочек с именами, похожими на Lesley

14.4. База данных о продуктах питания министерства сельского хозяйства США

Министерство сельского хозяйства США публикует данные о пищевой ценности продуктов питания. Программист Эшли Уильямс (Ashley Williams) преобразовал эту базу данных в формат JSON. Записи выглядят следующим образом:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },
    ...
  ],
  "nutrients": [
    {
      "value": 20.8,
      "units": "g",
      "description": "Protein",
      "group": "Composition"
    },
    ...
  ]
}
```

У каждого продукта питания есть ряд идентифицирующих атрибутов и два списка: питательные элементы и размеры порций. Для анализа данные в такой форме подходят плохо, поэтому необходимо их переформатировать.

Скачав архив с указанного адреса и распаковав его, вы можете загрузить его в Python-программу с помощью любой библиотеки для работы с JSON. Я воспользуюсь стандартным модулем Python `json`:

```
In [154]: import json

In [155]: db = json.load(open('datasets/usda_food/database.json'))

In [156]: len(db)
Out[156]: 6636
```

Каждая запись в `db` – словарь, содержащий все данные об одном продукте питания. Поле `'nutrients'` – это список словарей, по одному для каждого питательного элемента:

```
In [157]: db[0].keys()
Out[157]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'group', 'porti
ons', 'nutrients'])

In [158]: db[0]['nutrients'][0]
Out[158]:
{'description': 'Protein',
```

```
 'group': 'Composition',
 'units': 'g',
 'value': 25.18}
```

```
In [159]: nutrients = pd.DataFrame(db[0]['nutrients'])
```

```
In [160]: nutrients[:7]
Out[160]:
                    description       group units    value
0                       Protein  Composition     g    25.18
1             Total lipid (fat)  Composition     g    29.20
2  Carbohydrate, by difference  Composition     g     3.06
3                           Ash        Other     g     3.28
4                        Energy       Energy  kcal   376.00
5                         Water  Composition     g    39.28
6                        Energy       Energy    kJ  1573.00
```

Преобразуя список словарей в DataFrame, можно задать список полей, которые нужно извлекать. Мы ограничимся названием продукта, группой, идентификатором и производителем:

```
In [161]: info_keys = ['description', 'group', 'id', 'manufacturer']
```

```
In [162]: info = DataFrame(db, columns=info_keys)
```

```
In [163]: info[:5]
Out[163]:
                          description                    group    id \
0                     Cheese, caraway  Dairy and Egg Products  1008
1                     Cheese, cheddar  Dairy and Egg Products  1009
2                        Cheese, edam  Dairy and Egg Products  1018
3                        Cheese, feta  Dairy and Egg Products  1019
4  Cheese, mozzarella, part skim milk  Dairy and Egg Products  1028
  manufacturer
0
1
2
3
4
```

```
In [164]: info.info()
Out[164]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description    6636 non-null object
group          6636 non-null object
id             6636 non-null int64
manufacturer   5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

Метод value_counts покажет распределение продуктов питания по группам:

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products    812
Beef Products                        618
Baked Products                       496
Breakfast Cereals                    403
Legumes and Legume Products          365
Fast Foods                           365
Lamb, Veal, and Game Products        345
Sweets                               341
Pork Products                        328
Fruits and Fruit Juices              328
Name: group, dtype: int64
```

Чтобы теперь произвести анализ данных о питательных элементах, проще всего собрать все питательные элементы для всех продуктов в одну большую таблицу. Для этого понадобится несколько шагов. Сначала я преобразую каждый список питательных элементов в объект DataFrame, добавлю столбец id, содержащий идентификатор продукта, и помещу этот DataFrame в список, после чего все объекты можно будет конкатенировать методом concat:

```
nutrients = []

for rec in db:
    fnuts = DataFrame(rec['nutrients'])
    fnuts['id'] = rec['id']
    nutrients.append(fnuts)

nutrients = pd.concat(nutrients, ignore_index=True)
```

Если все пройдет хорошо, то объект nutrients будет выглядеть следующим образом:

```
In [167]: nutrients
Out[167]:
                             description       group units    value     id
0                                Protein  Composition     g   25.180   1008
1                    Total lipid (fat)  Composition     g   29.200   1008
2          Carbohydrate, by difference  Composition     g    3.060   1008
3                                  Ash        Other     g    3.280   1008
4                               Energy       Energy  kcal  376.000   1008
...                                ...          ...   ...      ...    ...
389350            Vitamin B-12, added    Vitamins   mcg    0.000  43546
389351                    Cholesterol        Other    mg    0.000  43546
389352   Fatty acids, total saturated        Other     g    0.072  43546
389353  Fatty acids, total monounsaturated        Other     g    0.028  43546
389354  Fatty acids, total polyunsaturated        Other     g    0.041  43546
[389355 rows x 5 columns]
```

Я заметил, что по какой-то причине в этом DataFrame есть дубликаты, поэтому лучше их удалить:

```
In [168]: nutrients.duplicated().sum()
Out[168]: 14179

In [169]: nutrients = nutrients.drop_duplicates()
```

Поскольку столбцы 'group' и 'description' есть в обоих объектах DataFrame, переименуем их, чтобы было понятно, что есть что:

```
In [170]: col_mapping = {'description' : 'food',
   .....:                'group' : 'fgroup'}

In [171]: info = info.rename(columns=col_mapping, copy=False)

In [172]: info.info()
Out[172]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food          6636 non-null object
fgroup        6636 non-null object
id            6636 non-null int64
manufacturer  5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB

In [173]: col_mapping = {'description' : 'nutrient',
   .....:                'group' : 'nutgroup'}

In [174]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [175]: nutrients
Out[175]:
                                 nutrient     nutgroup units    value     id
0                                 Protein  Composition     g   25.180   1008
1                           Total lipid (fat)  Composition     g   29.200   1008
2                Carbohydrate, by difference  Composition     g    3.060   1008
3                                     Ash        Other     g    3.280   1008
4                                  Energy       Energy  kcal  376.000   1008
...                                   ...          ...   ...      ...    ...
389350                Vitamin B-12, added     Vitamins   mcg    0.000  43546
389351                        Cholesterol        Other    mg    0.000  43546
389352       Fatty acids, total saturated        Other     g    0.072  43546
389353  Fatty acids, total monounsaturated        Other     g    0.028  43546
389354  Fatty acids, total polyunsaturated        Other     g    0.041  43546
[375176 rows x 5 columns]
```

Сделав все это, мы можем слить info с nutrients:

```
In [176]: ndata = pd.merge(nutrients, info, on='id', how='outer')
```

```
In [177]: ndata.info()
Out[177]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns:
nutrient      375176  non-null object
nutgroup      375176  non-null object
units         375176  non-null object
value         375176  non-null float64
id            375176  non-null int64
food          375176  non-null object
fgroup        375176  non-null object
manufacturer  293054  non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB
```

```
In [178]: ndata.iloc[30000]
Out[178]:
nutrient                                      Glycine
nutgroup                                   Amino Acids
units                                                g
value                                             0.04
id                                                6158
food          Soup, tomato bisque, canned, condensed
fgroup                    Soups, Sauces, and Gravies
manufacturer
Name: 30000, dtype: object
```

Теперь можно построить график медианных значений по группе и типу питательного элемента (рис. 14.11):

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)

In [181]: result['Zinc, Zn'].order().plot(kind='barh')
```

Проявив смекалку, вы сможете найти, какой продукт питания наиболее богат каждым питательным элементом:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.loc(x.value.idxmax())
get_minimum = lambda x: x.loc(x.value.idxmin())

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# Немного уменьшить длину названия продукта питания
max_foods.food = max_foods.food.str[:50]
```

Получившийся объект DataFrame слишком велик, для того чтобы приводить его полностью. Ниже представлена только группа питательных элементов 'Amino Acids' (аминокислоты):

```
In [183]: max_foods.ix['Amino Acids']['food']
```

```
Out[183]:
nutrient
Alanine                                  Gelatins, dry powder, unsweetened
Arginine                                      Seeds, sesame flour, low-fat
Aspartic acid                                        Soy protein isolate
Cystine                        Seeds, cottonseed flour, low fat (glandless)
Glutamic acid                                        Soy protein isolate
                                              ...
Serine                      Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Threonine                   Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Tryptophan                   Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine                    Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Valine                      Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Name: food, Length: 19, dtype: object
```

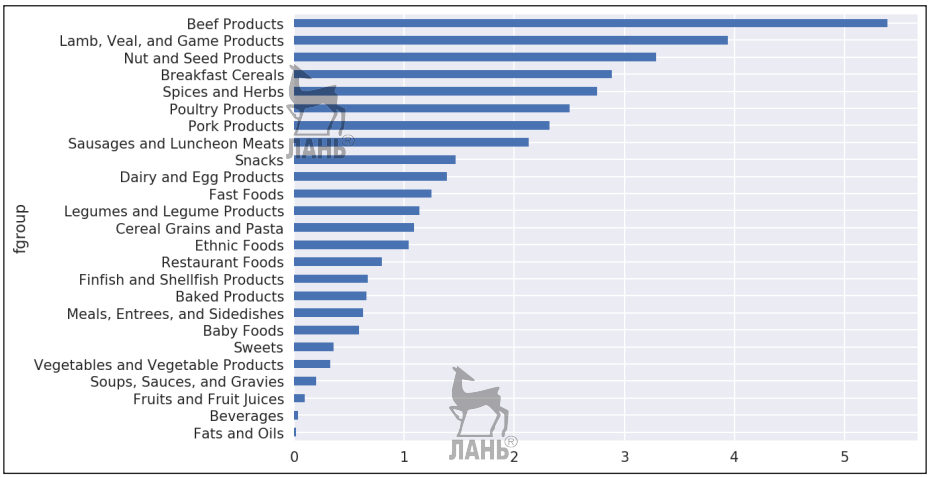


Рис. 14.11. Медианные значения цинка по группе питательных элементов

14.5. База данных федеральной избирательной комиссии

Федеральная избирательная комиссия США публикует данные о пожертвованиях участникам политических кампаний. Указывается имя жертвователя, род занятий, место работы и сумма пожертвования. Интерес представляет набор данных, относящийся к президентским выборам 2012 года. Версия этого набора, загруженная мной в июне 2012 года, представляет собой CSV-файл *P00000001-ALL.csv* размером 150 МБ, который можно загрузить с помощью функции `pandas.read_csv`:

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-ALL.csv')
```

```
In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id            1001731 non-null object
cand_id            1001731 non-null object
cand_nm            1001731 non-null object
contbr_nm          1001731 non-null object
contbr_city        1001712 non-null object
contbr_st          1001727 non-null object
contbr_zip         1001620 non-null object
contbr_employer     988002 non-null object
contbr_occupation   993301 non-null object
contb_receipt_amt  1001731 non-null float64
contb_receipt_dt   1001731 non-null object
receipt_desc         14166 non-null object
memo_cd              92482 non-null object
memo_text            97770 non-null object
form_tp            1001731 non-null object
file_num           1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

Ниже приведен пример записи в объекте DataFrame:

```
In [186]: fec.ix[123456]
Out[186]:
cmte_id                           C00431445
cand_id                           P80003338
cand_nm                       Obama, Barack
contbr_nm                       ELLMAN, IRA
contbr_city                           TEMPE
                                        ...
receipt_desc                            NaN
memo_cd                                 NaN
memo_text                               NaN
form_tp                               SA17A
file_num                             772372
Name: 123456, Length: 16, dtype: object
```

Наверное, вы с ходу сможете придумать множество способов манипуляции этими данными для извлечения полезной статистики о спонсорах и закономерностях жертвования. Далее я покажу различные виды анализа, чтобы проиллюстрировать рассмотренные в книге технические приемы.

Как видите, данные не содержат сведений о принадлежности кандидата к политической партии, а эту информацию было бы полезно добавить. Получить список различных кандидатов можно с помощью функции `unique`:

```
In [187]: unique_cands = fec.cand_nm.unique()
```

```
In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       "Roemer, Charles E. 'Buddy' III" 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, Herman',
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon',
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

Указать партийную принадлежность проще всего с помощью словаря[1]:

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican'
           'Perry, Rick': 'Republican',
           "Roemer, Charles E. 'Buddy' III": 'Republican',
           'Romney, Mitt': 'Republican',
           'Santorum, Rick': 'Republican'}
```

Далее, применяя этот словарь и метод `map` объектов Series, мы можем построить массив политических партий по именам кандидатов:

```
In [191]: fec.cand_nm[123456:123461]
Out[191]:
123456   Obama, Barack
123457   Obama, Barack
123458   Obama, Barack
123459   Obama, Barack
123460   Obama, Barack
Name: cand_nm, dtype=object

In [192]: fec.cand_nm[123456:123461].map(parties)
Out[192]:
123456   Democrat
123457   Democrat
123458   Democrat
123459   Democrat
123460   Democrat
Name: cand_nm, dtype=object
```

[1] Здесь сделано упрощающее предположение о том, что Гэри Джонсон – республиканец, хотя впоследствии он стал кандидатом от Либертарианской партии.

```
# Добавить в виде столбца
In [193]: fec['party'] = fec.cand_nm.map(parties)

In [194]: fec['party'].value_counts()
Out[194]:
Democrat     593746
Republican   407985
Name: party, dtype: int64
```

Теперь два замечания касательно подготовки данных. Во-первых, данные включают как пожертвования, так и возвраты (пожертвования со знаком минус):

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()
Out[195]:
True   991475
False   10256
```

Чтобы упростить анализ, я ограничусь только положительными суммами пожертвований:

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

Поскольку главными кандидатами являются Барак Обама и Митт Ромни, я подготовлю также подмножество, содержащее данные о пожертвованиях на их кампании:

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

Статистика пожертвований по роду занятий и месту работы

Распределение пожертвований по роду занятий – тема, которой посвящено много исследований. Например, юристы (в том числе прокуроры) обычно жертвуют в пользу демократов, а руководители предприятий – в пользу республиканцев. Вы вовсе не обязаны верить мне на слово, можете сами проанализировать данные. Для начала решим простую задачу – получим общую статистику пожертвований по роду занятий:

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED                                   233990
INFORMATION REQUESTED                      35107
ATTORNEY                                   34286
HOMEMAKER                                  29931
PHYSICIAN                                  23432
INFORMATION REQUESTED PER BEST EFFORTS     21138
ENGINEER                                   14334
TEACHER                                    13990
CONSULTANT                                 13273
PROFESSOR                                  12555
```

Видно, что часто различные занятия на самом деле относятся к одной и той же основной профессии с небольшими вариациями. Ниже показан код, который позволяет произвести очистку, отобразив один род занятий на другой. Обратите внимание на трюк с методом dict.get, который позволяет «передавать насквозь» занятия, которым ничего не сопоставлено:

```
occ_mapping = {
  'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
  'INFORMATION REQUESTED' : 'NOT PROVIDED',
  'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
  'C.E.O.': 'CEO'
}

# Если ничего не сопоставлено, вернуть x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

То же самое я проделаю для мест работы:

```
emp_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'SELF' : 'SELF-EMPLOYED',
   'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# Если ничего не сопоставлено, вернуть x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

Теперь можно воспользоваться функцией pivot_table для агрегирования данных по партиям и роду занятий, а затем отфильтровать роды занятий, на которые пришлось пожертвований на общую сумму не менее 2 000 000 долл.:

```
In [201]: by_occupation = fec.pivot_table('contb_receipt_amt',
   ....:                                   rows='contbr_occupation',
   ....:                                   cols='party', aggfunc='sum')

In [202]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [203]: over_2mm
Out[203]:
party                 Democrat         Republican
contbr_occupation
ATTORNEY            11141982.97     7477194.430000
CEO                  2074974.79     4211040.520000
CONSULTANT           2459912.71     2544725.450000
ENGINEER              951525.55     1818373.700000
EXECUTIVE            1355161.05     4138850.090000
...
PRESIDENT            1878509.95     4720923.760000
PROFESSOR            2165071.08      296702.730000
```

```
REAL ESTATE           528902.09     1625902.250000
RETIRED             25305116.38    23561244.489999
SELF-EMPLOYED         672393.40     1640252.540000
[17 rows x 2 columns]
```

Эти данные проще воспринять в виде графика (параметр `'barh'` означает горизонтальную столбчатую диаграмму, см. рис. 14.12):

```
In [205]: over_2mm.plot(kind='barh')
```

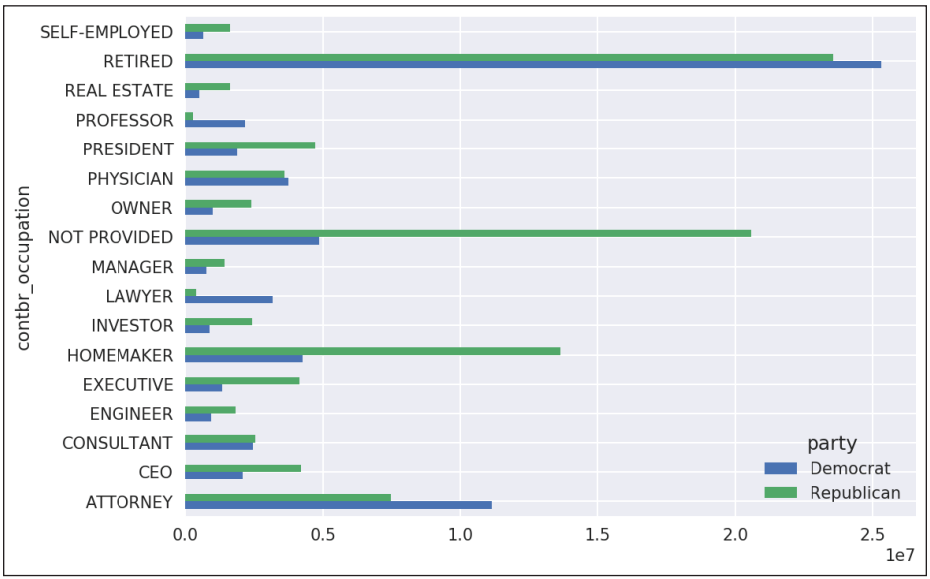


Рис. 14.12. Общая сумма пожертвований по партиям для родов занятий с максимальной суммой пожертвований

Возможно, вам интересны профессии самых щедрых жертвователей или названия компаний, которые больше всех пожертвовали Бараку Обаме или Митту Ромни. Для этого можно сгруппировать данные по имени кандидата, а затем воспользоваться вариантом метода `top`, рассмотренного выше в этой главе:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)['contb_receipt_amt'].sum()
    return totals.order(ascending=False)[-n:]
```

Затем агрегируем по роду занятий и месту работы:

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
Out[208]:
cand_nm         contbr_occupation
```

```
Obama, Barack  RETIRED                   25305116.38
               ATTORNEY                  11141982.97
               NOT PROVIDED               4866973.96
               HOMEMAKER                  4248875.80
               PHYSICIAN                  3735124.94
               LAWYER                     3160478.87
               CONSULTANT                 2459912.71
Romney, Mitt   RETIRED                   11508473.59
               NOT PROVIDED              11396894.84
               HOMEMAKER                  8147446.22
               ATTORNEY                   5364718.82
               PRESIDENT                  2491244.89
               EXECUTIVE                  2300947.03
               C.E.O.                     1968386.11
Name: contb_receipt_amt, Length: 14, dtype: float64

In [209]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[209]:
cand_nm        contbr_employer
Obama, Barack  RETIRED                   22694358.85
               SELF-EMPLOYED             18626807.16
               NOT EMPLOYED               8586308.70
               INFORMATION REQUESTED      5053480.37
               HOMEMAKER                  2605408.54
                                            ...
Romney, Mitt   CREDIT SUISSE              281150.00
               MORGAN STANLEY             267266.00
               GOLDMAN SACH & CO.         238250.00
               BARCLAYS CAPITAL           162750.00
               H.I.G. CAPITAL             139500.00
Name: contb_receipt_amt, Length: 20, dtype: float64
```

Распределение суммы пожертвований по интервалам

Полезный вид анализа данных – дискретизация сумм пожертвований с помощью функции cut:

```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000,
   .....:                  100000, 1000000, 10000000])

In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [212]: labels
Out[212]:
411       (10, 100]
412     (100, 1000]
413     (100, 1000]
414       (10, 100]
415       (10, 100]
          ...
```

```
701381    (10, 100]
701382  (100, 1000]
701383     (1, 10]
701384    (10, 100]
701385  (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64]): [(0, 1] < (1, 10] < (10, 100] < (100, 1000] < (1
000, 10000] <
                                  (10000, 100000] < (100000, 1000000] < (1000000,
10000000]]
```

Затем можно сгруппировать данные для Барака Обамы и Митта Ромни по имени и метке интервала и построить гистограмму сумм пожертвований:

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [214]: grouped.size().unstack(0)
Out[214]:
cand_nm              Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                         493            77
(1, 10]                      40070          3681
(10, 100]                   372280         31853
(100, 1000]                 153991         43357
(1000, 10000]                22284         26186
(10000, 100000]                  2             1
(100000, 1000000]                3           NaN
(1000000, 10000000]              4           NaN
```

Отсюда видно, что Барак Обама получил гораздо больше небольших пожертвований, чем Митт Ромни. Можно также вычислить сумму размеров пожертвований и нормировать распределение по интервалам, чтобы наглядно представить процентную долю пожертвований каждого размера от общего их числа по отдельным кандидатам (рис. 14.13):

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)

In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)

In [218]: normed_sums
Out[218]:
cand_nm                   Obama, Barack      Romney, Mitt
contb_receipt_amt
(0, 1]                         0.805182          0.194818
(1, 10]                        0.918767          0.081233
(10, 100]                      0.910769          0.089231
(100, 1000]                    0.710176          0.289824
(1000, 10000]                  0.447326          0.552674
(10000, 100000]                0.823120          0.176880
(100000, 1000000]              1.000000               NaN
(1000000, 10000000]            1.000000               NaN
```

```
In [219]: normed_sums[:-2].plot(kind='barh')
```

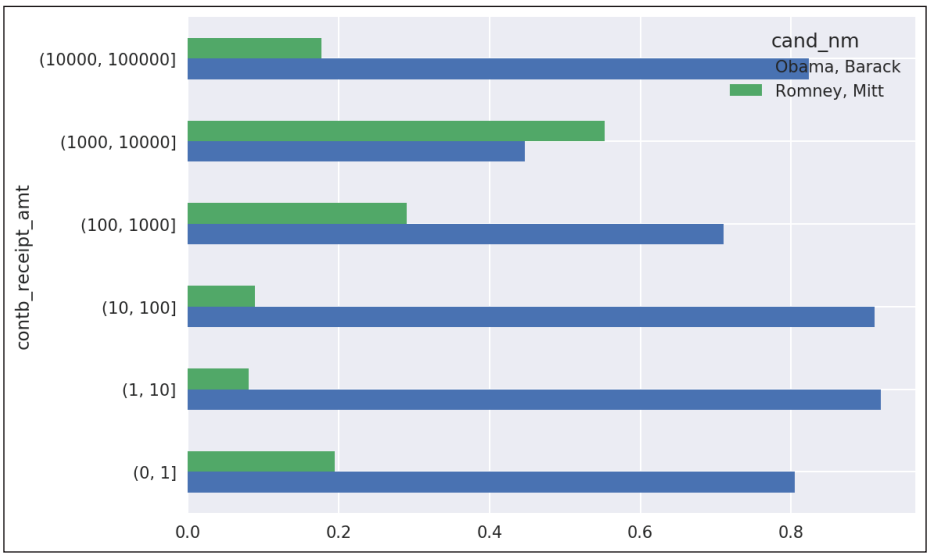


Рис. 14.13. Процентная доля пожертвований каждого размера от общего их числа для обоих кандидатов

Я исключил два самых больших интервала, потому что они соответствуют пожертвованиям юридических лиц. Результат показан на рис. 9.3.

Этот анализ можно уточнить и улучшить во многих направлениях. Например, можно было бы агрегировать пожертвования по имени и почтовому индексу спонсора, чтобы отделить спонсоров, внесших много небольших пожертвований, от тех, кто внес одно или несколько крупных пожертвований. Призываю вас скачать этот набор данных и исследовать его самостоятельно.

Статистика пожертвований по штатам

Агрегирование данных по кандидатам и штатам – вполне рутинная задача:

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [221]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm    Obama, Barack  Romney, Mitt
contbr_st
AK              281840.15      86204.24
AL              543123.48     527303.51
AR              359247.28     105556.00
```

```
AZ           1506476.98     1888436.23
CA          23824984.24    11237636.60
CO           2132429.49     1506714.12
CT           2068291.26     3499475.45
DC           4373538.80     1025137.50
DE            336669.14       82712.00
FL           7318178.58     8338458.81
```

Поделив каждую строку на общую сумму пожертвований, мы получим для каждого кандидата процентную долю от общей суммы, приходящуюся на каждый штат:

```
In [224]: percent = totals.div(totals.sum(1), axis=0)

In [225]: percent[:10]
Out[225]:
cand_nm    Obama, Barack  Romney, Mitt
contbr_st
AK              0.765778      0.234222
AL              0.507390      0.492610
AR              0.772902      0.227098
AZ              0.443745      0.556255
CA              0.679498      0.320502
CO              0.585970      0.414030
CT              0.371476      0.628524
DC              0.810113      0.189887
DE              0.802776      0.197224
FL              0.467417      0.532583
```

14.6. Заключение

На этом основной текст книги заканчивается. Дополнительные материалы, которые могут быть вам полезны, я включил в приложения.

За пять лет, прошедших с момента публикации первого издания этой книги, Python стал популярным и широко распространенным языком для анализа данных. Полученные в процессе чтения навыки программирования останутся актуальными еще долго. Надеюсь, что рассмотренные нами инструменты и библиотеки сослужат вам хорошую службу.

Приложение А. Дополнительные сведения о библиотеке NumPy

В этом приложении мы глубже рассмотрим библиотеку NumPy, предназначенную для вычислений с массивами. Разберемся во внутренних деталях типа ndarray и поговорим о дополнительных операциях с массивами и алгоритмах.

Приложение содержит разнородные темы, поэтому читать его последовательно не обязательно.

А.1. Внутреннее устройство объекта ndarray

Объект ndarray из библиотеки NumPy позволяет интерпретировать блок однородных данных (непрерывный или с шагом, подробнее об этом ниже) как многомерный массив. Мы уже видели, что тип данных, или `dtype`, определяет, как именно интерпретируются данные: как числа с плавающей точкой, целые, булевы или еще как-то.

Своей эффективностью ndarray отчасти обязан тому, что любой объект массива является *шаговым* (strided) представлением блока данных. Может возникнуть вопрос, как удается построить представление массива `arr[::2, ::-1]` без копирования данных. Дело в том, что объект ndarray не просто блок памяти, дополненный знанием о типе в виде dtype; в нем еще хранится информация, позволяющая перемещаться по массиву шагами разного размера. Точнее, в реализации ndarray имеется:

- *указатель на данные*, т. е. на блок полученной от системы памяти;
- *тип данных*, или dtype, описывающий значения элементов массива фиксированного размера;
- кортеж, описывающий *форму* массива;

- кортеж *шагов*, т. е. целых чисел, показывающих, на сколько байтов нужно сместиться, чтобы перейти к следующему элементу по некоторому измерению.

На рис. А.1 схематически показано внутреннее устройство ndarray.

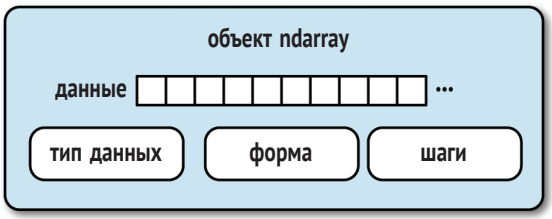


Рис. А.1. Объект ndarray из библиотеки NumPy

Например, у массива 10×5 будет форма `(10, 5)`:

```
In [10]: np.ones((10, 5)).shape
Out[10]: (10, 5)
```

Для типичного массива (организованного в соответствии с принятым в языке C соглашением) 3×4×5 чисел типа `float64` (8-байтовых) кортеж шагов имеет вид `(160, 40, 8)` (знать о шагах полезно, потому что в общем случае чем больше шаг по конкретной оси, тем дороже обходятся вычисления по этой оси):

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[11]: (160, 40, 8)
```

Хотя типичному пользователю NumPy редко приходится интересоваться шагами массива, они играют важнейшую роль в построении представлений массива без копирования. Шаги могут быть даже отрицательными, что позволяет проходить массив в *обратном направлении*, как в случае среза вида `obj[::-1]` или `obj[:, ::-1]`.

Иерархия типов данных в NumPy

Иногда в программе необходимо проверить, что хранится в массиве: целые числа, числа с плавающей точкой, строки или объекты Python. Поскольку существует много типов с плавающей точкой (от `float16` до `float128`), для проверки, присутствует ли dtype в списке типов, приходится писать длинный код. По счастью, определены такие суперклассы, как `np.integer` или `np.floating`, которые можно использовать в сочетании с функцией `np.issubdtype`:

```
In [12]: ints = np.ones(10, dtype=np.uint16)

In [13]: floats = np.ones(10, dtype=np.float32)

In [14]: np.issubdtype(ints.dtype, np.integer)
```

```
Out[14]: True

In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

Вывести все родительские классы данного типа dtype позволяет метод `mro`:

```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```


Поэтому мы также имеем:

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

Большинству пользователей NumPy об этом знать необязательно, но иногда оказывается удобно. На рис. А.2 показан граф наследования dtype[1].

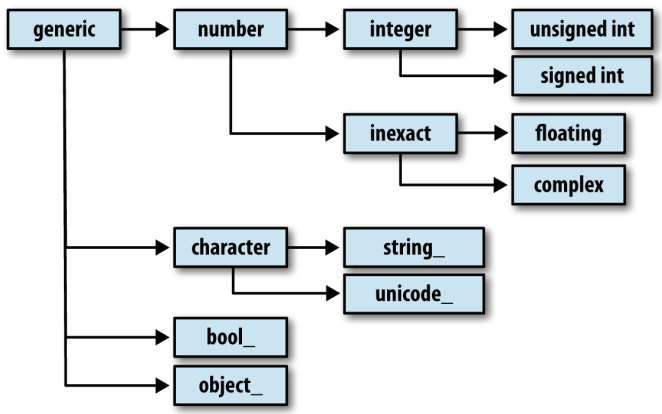


Рис. А.2. Граф наследования dtype

А.2. Дополнительные манипуляции с массивами

Помимо прихотливого индексирования, вырезания и формирования булевых подмножеств, существует много других способов работы с массивами. И хо-

[1] В именах некоторых типов dtype присутствуют знаки подчеркивания. Они нужны, чтобы избежать конфликтов между именами типов NumPy и встроенных типов Python.

тя большую часть сложных задач, решаемых в ходе анализа данных, берут на себя высокоуровневые функции из библиотеки pandas, иногда возникает необходимость написать алгоритм обработки данных, которого нет в имеющихся библиотеках.

Изменение формы массива

Во многих случаях изменить форму массива можно без копирования данных. Для этого следует передать кортеж с описанием новой формы методу экземпляра массива `reshape`. Например, предположим, что имеется одномерный массив, который мы хотели бы преобразовать в матрицу (результат показан на рис. А.3):

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
Out[20]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Рис. А.3. Изменение формы с преобразованием в двумерный массив, организованный как в C (по строкам) и как в Fortran (по столбцам)

Форму многомерного массива также можно изменить:

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Одно из переданных в описателе формы измерений может быть равно –1, его значение будет выведено из данных:

```
In [22]: arr = np.arange(15)

In [23]: arr.reshape((5, -1))
Out[23]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Поскольку атрибут shape массива является кортежем, его также можно передать методу reshape:

```
In [24]: other_arr = np.ones((3, 5))

In [25]: other_arr.shape
Out[25]: (3, 5)

In [26]: arr.reshape(other_arr.shape)
Out[26]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

Обратная операция – переход от многомерного к одномерному массиву – называется *линеаризацией*:

```
In [27]: arr = np.arange(15).reshape((5, 3))

In [28]: arr
Out[28]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [29]: arr.ravel()
Out[29]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Метод ravel не создает копию данных, если без этого можно обойтись (подробнее об этом ниже). Метод flatten ведет себя как ravel, но всегда возвращает копию данных:

```
In [30]: arr.flatten()
Out[30]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Данные можно линеаризовать в разном порядке. Начинающим пользователям NumPy эта тема может показаться довольно сложной, поэтому ей посвящен целиком следующий подраздел.

Упорядочение элементов массива в C и в Fortran

Библиотека NumPy предлагает большую гибкость в определении порядка размещения данных в памяти. По умолчанию массивы NumPy размещаются *по строкам*. Это означает, что при размещении двумерного массива в памяти соседние элементы строки находятся в соседних ячейках памяти. Альтернативой является размещение *по столбцам*, тогда в соседних ячейках находятся соседние элементы столбца.

По историческим причинам порядок размещения по строкам называется порядком C, а по столбцам – порядком Fortran. В языке FORTRAN 77 матрицы размещаются по столбцам.

Функции типа `reshape` и `ravel` принимают аргумент `order`, показывающий, в каком порядке размещать данные в массиве. Обычно задают значение `'C'` или `'F'` (о менее употребительных значениях `'A'` и `'K'` можно прочитать в документации по NumPy, а их действие показано на рис. A.3).

```
In [31]: arr = np.arange(12).reshape((3, 4))

In [32]: arr
Out[32]:
array([[ 0, 1,  2,  3],
       [ 4, 5,  6,  7],
       [ 8, 9, 10, 11]])

In [33]: arr.ravel()
Out[33]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

In [34]: arr.ravel('F')
Out[34]: array([ 0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])
```

Изменение формы массива, имеющего больше двух измерений, – головоломное упражнение (см. рис. A.3). Основное различие между порядком C и Fortran состоит в том, в каком порядке перебираются измерения:

- *порядок по строкам* (*C*): старшие измерения обходятся *раньше* (т. е. сначала обойти ось 1, а потом переходить к оси 0);
- *порядок по столбцам* (*Fortran*): старшие измерения обходятся *позже* (т. е. сначала обойти ось 0, а потом переходить к оси 1).

Конкатенация и разбиение массива

Метод `numpy.concatenate` принимает произвольную последовательность (кортеж, список и т. п.) массивов и соединяет их вместе в порядке, определяемом указанной осью.

```
In [35]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [36]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [37]: np.concatenate([arr1, arr2], axis=0)
```

```
Out[37]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [38]: np.concatenate([arr1, arr2], axis=1)
Out[38]:
array([[ 1, 2, 3,  7,  8,  9],
       [ 4, 5, 6, 10, 11, 12]])
```


Есть несколько вспомогательных функций, например vstack и hstack, для выполнения типичных операций конкатенации. Приведенные выше операции можно было бы записать и так:

```
In [39]: np.vstack((arr1, arr2))
Out[39]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [40]: np.hstack((arr1, arr2))
Out[40]:
array([[ 1, 2, 3,  7,  8,  9],
       [ 4, 5, 6, 10, 11, 12]])
```

С другой стороны, функция split разбивает массив на несколько частей вдоль указанной оси:

```
In [41]: arr = np.random.randn(5, 2)

In [42]: arr
Out[42]:
array([[-0.2047,  0.4789],
       [-0.5194, -0.5557],
       [ 1.9658,  1.3934],
       [ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])

In [43]: first, second, third = np.split(arr, [1, 3])

In [44]: first
Out[44]: array([[-0.2047, 0.4789]])

In [45]: second
Out[45]:
array([[-0.5194, -0.5557],
       [ 1.9658,  1.3934]])

In [46]: third
Out[46]:
array([[ 0.0929, 0.2817],
       [ 0.769 , 1.2464]])
```

Значение [1, 3], переданное np.split, содержит индексы, по которым массив нужно разбить на части.

Перечень всех функций, относящихся к конкатенации и разбиению, приведен в табл. А.1, хотя некоторые из них – лишь надстройки над очень общей функцией concatenate.

Таблица А.1. Функции конкатенации массива

Функция	Описание
concatenate	Самая общая функция – конкатенирует коллекцию массивов вдоль указанной оси
vstack, row_stack	Составляет массивы по строкам (вдоль оси 0)
hstack	Составляет массивы по столбцам (вдоль оси 1)
column_stack	Аналогична hstack, но сначала преобразует одномерные массивы в двумерные векторы по столбцам
dstack	Составляет массивы в глубину (вдоль оси 2)
split	Разбивает массив в указанных позициях вдоль указанной оси
hsplit / vsplit / dsplit	Вспомогательные функции для разбиения по оси 0, 1 и 2 соответственно

Вспомогательные объекты: r_ и c_

В пространстве имен NumPy есть два специальных объекта: r_ и c_, благодаря которым составление массивов можно записать более кратко:

```
In [47]: arr = np.arange(6)

In [48]: arr1 = arr.reshape((3, 2))

In [49]: arr2 = np.random.randn(3, 2)

In [50]: np.r_[arr1, arr2]
Out[50]:
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
       [ 4.    ,  5.    ],
       [ 1.0072, -1.2962],
       [ 0.275 ,  0.2289],
       [ 1.3529,  0.8864]])

In [51]: np.c_[np.r_[arr1, arr2], arr]
Out[51]:
array([[ 0.    ,  1.    , 0. ],
       [ 2.    ,  3.    , 1. ],
       [ 4.    ,  5.    , 2. ],
       [ 1.0072, -1.2962, 3. ],
       [ 0.275 ,  0.2289, 4. ],
       [ 1.3529,  0.8864, 5. ]])
```

С их помощью можно также преобразовывать срезы в массивы:

```
In [52]: np.c_[1:6, -10:-5]
```

```
Out[52]:
array([[ 1, -10],
       [ 2,  -9],
       [ 3,  -8],
       [ 4,  -7],
       [ 5,  -6]])
```

О том, что еще могут делать объекты `c_` и `r_`, читайте в строке документации.

Повторение элементов: функции tile и repeat

Два полезных инструмента повторения, или репликации, массивов для порождения массивов большего размера – функции `repeat` и `tile`. Функция `repeat` повторяет каждый элемент массива несколько раз и создает больший массив:

```
In [53]: arr = np.arange(3)

In [54]: arr
Out[54]: array([0, 1, 2])

In [55]: arr.repeat(3)
Out[55]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

Необходимость повторять массивы при работе с NumPy возникает реже, чем в других популярных средах программирования, например в MATLAB. Основная причина заключается в том, что *укладывание* (тема следующего раздела) решает эту задачу лучше.

По умолчанию если передать целое число, то каждый элемент повторяется столько раз. Если же передать массив целых чисел, то разные элементы могут быть повторены разное число раз:

```
In [56]: arr.repeat([2, 3, 4])
Out[56]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Элементы многомерных массивов повторяются вдоль указанной оси:

```
In [57]: arr = np.random.randn(2, 2)

In [58]: arr
Out[58]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [59]: arr.repeat(2, axis=0)
Out[59]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])
```

Отметим, что если ось не указана, то массив сначала линеаризуется, а это, скорее всего, не то, что вы хотели. Чтобы повторить разные срезы многомерного массива различное число раз, можно передать массив целых чисел:

```
In [60]: arr.repeat([2, 3], axis=0)
Out[60]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])

In [61]: arr.repeat([2, 3], axis=1)
Out[61]:
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
       [ 1.669 ,  1.669 , -0.4386, -0.4386, -0.4386]])
```

Функция `tile` (замостить) – с другой стороны, просто сокращенный способ составления копий массива вдоль оси. Это можно наглядно представлять себе как «укладывание плиток»:

```
In [62]: arr
Out[62]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [63]: np.tile(arr, 2)
Out[63]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

Второй аргумент – количество плиток; если это скаляр, то мощение производится по строкам, а не по столбцам. Но второй аргумент `tile` может быть кортежем, описывающим порядок мощения:

```
In [64]: arr
Out[64]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [65]: np.tile(arr, (2, 1))
Out[65]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [66]: np.tile(arr, (3, 2))
Out[66]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
```

```
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

Эквиваленты прихотливого индексирования: функции take и put

В главе 4 описывался способ получить и установить подмножество массива с помощью *прихотливого* индексирования массивами целых чисел:

```
In [67]: arr = np.arange(10) * 100

In [68]: inds = [7, 1, 2, 6]

In [69]: arr[inds]
Out[69]: array([700, 100, 200, 600])
```

Существуют и другие методы ndarray, полезные в частном случае, когда выборка производится только по одной оси:

```
In [70]: arr.take(inds)
Out[70]: array([700, 100, 200, 600])

In [71]: arr.put(inds, 42)

In [72]: arr
Out[72]: array([ 0, 42, 42, 300, 400, 500, 42, 42, 800, 900])

In [73]: arr.put(inds, [40, 41, 42, 43])

In [74]: arr
Out[74]: array([ 0, 41, 42, 300, 400, 500, 43, 40, 800, 900])
```

Чтобы использовать функцию take для других осей, нужно передать именованный параметр axis:

```
In [75]: inds = [2, 0, 2, 1]

In [76]: arr = np.random.randn(2, 4)

In [77]: arr
Out[77]:
array([[-0.5397, 0.477 , 3.2489, -1.0212],
       [-0.5771, 0.1241, 0.3026,  0.5238]])

In [78]: arr.take(inds, axis=1)
Out[78]:
array([[ 3.2489, -0.5397, 3.2489, 0.477 ],
       [ 0.3026, -0.5771, 0.3026, 0.1241]])
```

Функция put не принимает аргумент axis, а обращается по индексу к линеаризованной версии массива (одномерному массиву, построенному в предположении, что исходный массив организован, как в C). Следовательно, если с помощью массива индексов требуется установить элементы на других осях, то придется воспользоваться прихотливым индексированием.

A.3. Укладывание

Словом «укладывание» (broadcasting) описывается способ выполнения арифметических операций над массивами разной формы. Это очень мощный механизм, но даже опытные пользователи иногда испытывают затруднения с его пониманием. Простейший пример укладывания – комбинирование скалярного значения с массивом:

```
In [79]: arr = np.arange(5)

In [80]: arr
Out[80]: array([0, 1, 2, 3, 4])

In [81]: arr * 4
Out[81]: array([ 0, 4, 8, 12, 16])
```

Здесь мы говорим, что скалярное значение 4 уложено на все остальные элементы в результате операции умножения.

Другой пример: мы можем сделать среднее по столбцам массива равным нулю, вычтя из каждого столбца столбец, содержащий средние значения. И сделать это очень просто:

```
In [82]: arr = np.random.randn(4, 3)

In [83]: arr.mean(0)
Out[83]: array([-0.3928, -0.3824, -0.8768])

In [84]: demeaned = arr - arr.mean(0)

In [85]: demeaned
Out[85]:
array([[ 0.3937,  1.7263,  0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 ,  0.9426, -0.3891],
       [ 0.5126, -0.6811,  1.2097]])

In [86]: demeaned.mean(0)
Out[86]: array([-0., 0., -0.])
```

Эта операция показана на рис. A.4. Для приведения к нулю средних по строкам с помощью укладывания требуется проявить осторожность. К счастью, укладывание значений меньшей размерности вдоль любого измерения массива (например, вычитание средних по строкам из каждого столбца двумерного массива) возможно при соблюдении следующего правила:

Правило укладывания

Два массива совместимы по укладыванию, если для обоих *последних измерений* (т. е. отсчитываемых с конца) длины осей совпадают или хотя бы одна длина равна 1. Тогда укладывание производится по отсутствующим измерениям или по измерениям длины 1.

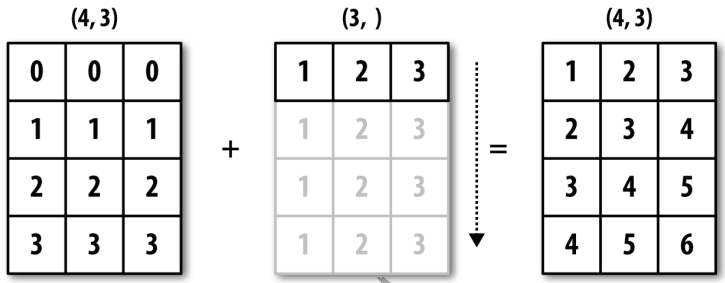


Рис. А.4. Укладывание одномерного массива по оси 0

Даже я, опытный пользователь NumPy, иногда вынужден рисовать картинки, чтобы понять, как будет применяться правило укладывания. Вернемся к последнему примеру и предположим, что мы хотим вычесть среднее значение из каждой строки, а не из каждого столбца. Поскольку длина массива `arr.mean(0)` равна 3, он совместим по укладыванию вдоль оси 0, так как по последнему измерению длины осей (три) совпадают. Согласно правилу, чтобы произвести вычитание по оси 1 (т. е. вычесть среднее по строкам из каждой строки), меньший массив должен иметь форму `(4, 1)`:

```
In [87]: arr
Out[87]:
array([[ 0.0009,  1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])

In [88]: row_means = arr.mean(1)

In [89]: row_means.shape
Out[89]: (4,)

In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [91]: demeaned = arr - row_means.reshape((4, 1))
```

```
In [92]: demeaned.mean(1)
Out[92]: array([ 0., -0., 0., 0.])
```

Эта операция проиллюстрирована рис. A.5:

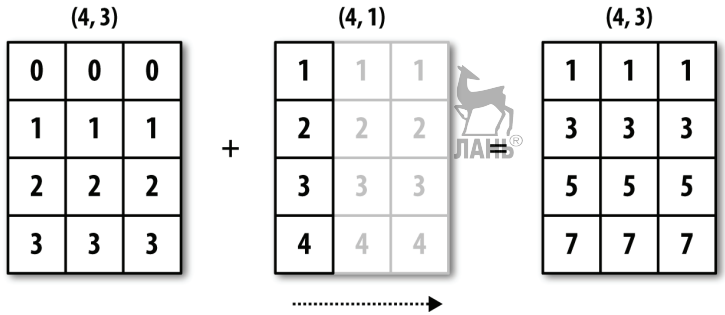


Рис. A.5. Укладывание двумерного массива по оси 1

На рис. A.6 приведена еще одна иллюстрация, где мы вычитаем двумерный массив из трехмерного по оси 0.

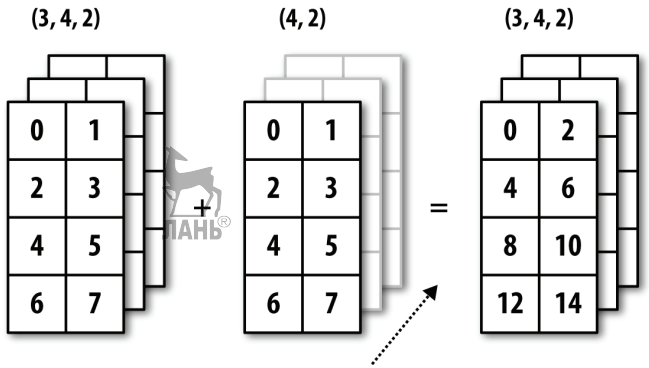


Рис. A.6. Укладывание трехмерного массива по оси 0

Укладывание по другим осям

Укладывание многомерных массивов может показаться еще более головоломной задачей, но на самом деле нужно только соблюдать правило. Если оно не соблюдено, то будет выдана ошибка вида:

```
In [93]: arr - arr.mean(1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4)
```

Очень часто возникает необходимость выполнить арифметическую операцию с массивом меньшей размерности по оси, отличной от 0. Согласно правилу укладывания, длина «размерности укладывания» в меньшем массиве должна быть равна 1. В примере вычитания среднего это означало, что массив средних по строкам должен иметь форму `(4, 1)`, а не `(4,)`:

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[-0.2095,  1.1334, -0.9239],
       [ 0.8562, -0.6828, -0.1734],
       [-0.3386,  1.0823, -0.7438],
       [ 0.3234, -0.8599,  0.5365]])
```

В трехмерном случае укладывание по любому из трех измерений сводится к изменению формы данных для обеспечения совместимости массивов. На рис. A.7 наглядно показано, каковы должны быть формы для укладывания по любой оси трехмерного массива.

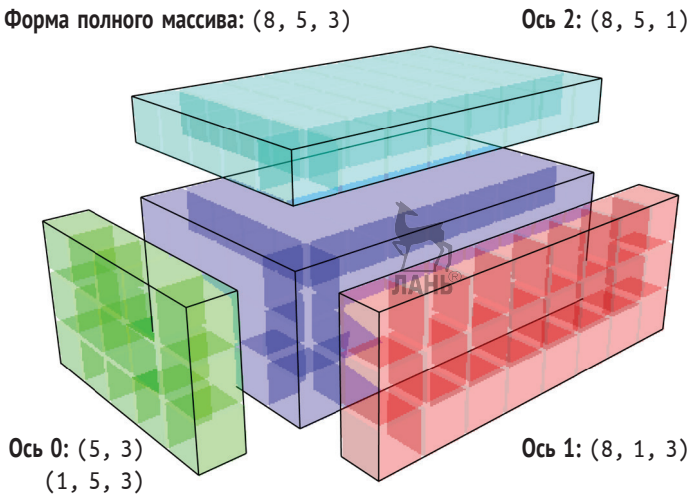


Рис. A.7. Совместимые формы двумерного массива для укладывания в трехмерный массив

Поэтому часто приходится добавлять новую ось длины 1 специально для укладывания, особенно в обобщенных алгоритмах. Один из вариантов – использование `reshape`, но для вставки оси нужно построить кортеж, описывающий новую форму. Это утомительное занятие. Поэтому в NumPy имеется специальный синтаксис для вставки новых осей путем доступа по индексу. Чтобы вставить новую ось, мы воспользуемся специальным атрибутом `np.newaxis` и «полными» срезами:

```
In [95]: arr = np.zeros((4, 4))

In [96]: arr_3d = arr[:, np.newaxis, :]
```

```
In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis]
Out[99]:
array([[-2.3594],
       [-0.1995],
       [-1.542 ]])

In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[-2.3594, -0.1995, -1.542 ]])
```

Таким образом, если имеется трехмерный массив и требуется привести его к нулевому среднему, скажем, по оси 2, то нужно написать:

```
In [101]: arr = np.random.randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [104]: depth_means.shape
Out[104]: (3, 4)

In [105]: demeaned = arr - depth_means[:, :, np.newaxis]

In [106]: demeaned.mean(2)
Out[106]:
array([[ 0., 0., -0., -0.],
       [ 0., 0., -0., 0.],
       [ 0., 0., -0., -0.]])
```

Возможно, вас интересует, нет ли способа обобщить вычитание среднего вдоль оси, не жертвуя производительностью. Есть, но придется попотеть с индексированием:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # Это обобщает операции вида [:, :, np.newaxis] на N измерений
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

Установка элементов массива с помощью укладывания

То же правило укладывания, что управляет арифметическими операциями, применимо и к установке значений элементов с помощью доступа по индексу. В простейшем случае это выглядит так:

```
In [107]: arr = np.zeros((4, 3))

In [108]: arr[:] = 5

In [109]: arr
Out[109]:
array([[ 5., 5., 5.],
       [ 5., 5., 5.],
       [ 5., 5., 5.],
       [ 5., 5., 5.]])
```

Однако если имеется одномерный массив значений, который требуется записать в столбцы массива, то можно сделать и это – при условии совместимости формы:

```
In [110]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [111]: arr[:] = col[:, np.newaxis]

In [112]: arr
Out[112]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [113]: arr[:2] = [[-1.37], [0.509]]

In [114]: arr
Out[114]:
array([[-1.37 , -1.37 , -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6 ,   1.6  ,  1.6  ]])
```

A.4. Дополнительные способы использования универсальных функций

Многие пользователи NumPy используют универсальные функции только ради быстрого выполнения поэлементных операций, однако у них есть и другие возможности, которые иногда позволят кратко записать код без циклов.

Методы экземпляра u-функций

Любая бинарная u-функция в NumPy имеет специальные методы для выполнения некоторых видов векторных операций. Все они перечислены в табл. А.2, но я приведу и несколько конкретных примеров для иллюстрации.

Метод `reduce` принимает массив и агрегирует его, возможно, вдоль указанной оси, выполняя последовательность бинарных операций. Вот, например, как можно с помощью `np.add.reduce` просуммировать элементы массива:

```
In [115]: arr = np.arange(10)

In [116]: np.add.reduce(arr)
Out[116]: 45

In [117]: arr.sum()
Out[117]: 45
```

Начальное значение (для add оно равно 0) зависит от u-функции. Если задана ось, то редукция производится вдоль этой оси. Это позволяет давать краткие ответы на некоторые вопросы. В качестве не столь тривиального примера воспользуемся методом np.logical_and, чтобы проверить, отсортированы ли значения в каждой строке массива:

```
In [118]: np.random.seed(12346)     # для воспроизводимости

In [119]: arr = np.random.randn(5, 5)

In [120]: arr[::2].sort(1) # отсортировать несколько строк

In [121]: arr[:, :-1] < arr[:, 1:]
Out[121]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

In [122]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[122]: array([ True, False,  True, False,  True], dtype=bool)
```

Отметим, что logical_and.reduce эквивалентно методу all.

Метод accumulate соотносится с reduce, как cumsum с sum. Он порождает массив того же размера, содержащий промежуточные «аккумулированные» значения:

```
In [123]: arr = np.arange(15).reshape((3, 5))

In [124]: np.add.accumulate(arr, axis=1)
Out[124]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

Метод outer вычисляет прямое произведение двух массивов:

```
In [125]: arr = np.arange(3).repeat([1, 2, 2])

In [126]: arr
Out[126]: array([0, 1, 1, 2, 2])
```

```
In [127]: np.multiply.outer(arr, np.arange(5))
Out[127]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

Размерность массива, возвращенного `outer`, равна сумме размерностей его параметров:

```
In [128]: x, y = np.random.randn(3, 4), np.random.randn(5)

In [129]: result = np.subtract.outer(x, y)

In [130]: result.shape
Out[130]: (3, 4, 5)
```

Последний метод, `reduceat`, выполняет локальную редукцию, т. е. по существу операцию `groupby`, в которой агрегируется сразу несколько срезов массива. Он принимает последовательность «границ интервалов», описывающую, как разбивать и агрегировать значения:

```
In [131]: arr = np.arange(10)

In [132]: np.add.reduceat(arr, [0, 5, 8])
Out[132]: array([10, 18, 17])
```

На выходе получаются результаты редукции (в данном случае суммирования) по срезам `arr[0:5]`, `arr[5:8]` и `arr[8:]`. Как и другие методы, `reduceat` принимает необязательный аргумент `axis`:

```
In [133]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [134]: arr
Out[134]:
array([[ 0, 0, 0, 0,  0],
       [ 0, 1, 2, 3,  4],
       [ 0, 2, 4, 6,  8],
       [ 0, 3, 6, 9, 12]])

In [135]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[135]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

Неполный перечень методов u-функций приведен в табл. А.2.

Таблица А.2. Методы u-функций

Метод	Описание
`reduce(x)`	Агрегирует значения путем последовательного применения операции
`accumulate(x)`	Агрегирует значения, сохраняя все промежуточные агрегаты
`reduceat(x, bins)`	Локальная редукция, или «group by». Редуцирует соседние срезы данных и порождает массив агрегатов
`outer(x, y)`	Применяет операцию ко всем парам элементов x и y. Результирующий массив имеет форму `x.shape + y.shape`

Написание новых u-функций на Python

Для создания собственных u-функций для NumPy существует несколько механизмов. Самый общий – использовать C API NumPy, но он выходит за рамки книги. В этом разделе будем рассматривать u-функции на чистом Python.

Метод `numpy.frompyfunc` принимает функцию Python и спецификацию количества входов и выходов. Например, простую функцию, выполняющую поэлементное сложение, можно было бы описать так:

```
In [136]: def add_elements(x, y):
   .....:     return x + y

In [137]: add_them = np.frompyfunc(add_elements, 2, 1)

In [137]: add_them(np.arange(8), np.arange(8))
Out[137]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Функции, созданные методом `frompyfunc`, всегда возвращают массивы объектов Python, что не очень удобно. По счастью, есть альтернативный, хотя и не столь функционально богатый метод `numpy.vectorize`, который умеет лучше выводить типы:

```
In [139]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

Оба метода позволяют создавать аналоги u-функций, которые, правда, работают очень медленно, потому что должны вызывать функцию Python для вычисления каждого элемента, а это далеко не так эффективно, как циклы в написанных на C универсальных функциях NumPy:

```
In [141]: arr = np.random.randn(10000)

In [142]: %timeit add_them(arr, arr)
4.12 ms +- 182 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [143]: %timeit np.add(arr, arr)
6.89 us +- 504 ns per loop (mean +- std. dev. of 7 runs, 100000 loops each)
```

Ниже в этой главе мы покажем, как создавать быстрые u-функции на Python с помощью проекта Numba (http://numba.pydata.org/).

A.5. Структурные массивы

Вы, наверное, обратили внимание, что все рассмотренные до сих пор примеры ndarray были контейнерами *однородных* данных, т. е. блоками памяти, в которых каждый элемент занимает одно и то же количество байтов, определяемое типом данных dtype. Создается впечатление, что представить в виде массива неоднородные данные, как в таблице, невозможно. *Структурный* массив – это объект ndarray, в котором каждый элемент можно рассматривать как аналог *структуры* (struct) в языке C (отсюда и название «структурный») или строки в таблице SQL, содержащий несколько именованных полей:

```
In [144]: dtype = [('x', np.float64), ('y', np.int32)]

In [145]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)

In [146]: sarr
Out[146]:
array([(1.5, 6), (3.1416, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

Существует несколько способов задать структурный dtype (см. документацию по NumPy в Сети). Наиболее распространенный – с помощью списка кортежей вида `(field_name, field_data_type)`. Теперь элементами массива являются кортежеподобные объекты, к элементам которых можно обращаться как к словарю:

```
In [147]: sarr[0]
Out[147]: (1.5, 6)

In [148]: sarr[0]['y']
Out[148]: 6
```

Имена полей хранятся в атрибуте `dtype.names`. При доступе к полю структурного массива возвращается шаговое представление данных, т. е. копирования не происходит:

```
In [149]: sarr['x']
Out[149]: array([ 1.5 , 3.1416])
```

Вложенные типы данных и многомерные поля

При описании структурного dtype можно факультативно передать форму (в виде целого числа или кортежа):

```
In [150]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [151]: arr = np.zeros(4, dtype=dtype)

In [152]: arr
Out[152]:
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

В данном случае поле x в каждой записи ссылается на массив длиной 3:

```
In [153]: arr[0]['x']
Out[153]: array([0, 0, 0])
```

При этом результатом операции arr['x'] является двумерный массив, а не одномерный, как в предыдущих примерах:

```
In [154]: arr['x']
Out[154]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Это позволяет представлять более сложные вложенные структуры в виде одного блока памяти в массиве. Но если dtype может быть произвольно сложным, то почему бы и не вложенным? Вот простой пример:

```
In [155]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]

In [156]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

In [157]: data['x']
Out[157]:
array([(1.0, 2.0), (3.0, 4.0)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [158]: data['y']
Out[158]: array([5, 6], dtype=int32)

In [159]: data['x']['a']
Out[159]: array([ 1., 3.])
```

Объект DataFrame из библиотеки pandas не поддерживает этот механизм напрямую, хотя иерархическое индексирование в чем-то похоже.

Зачем нужны структурные массивы?

По сравнению, скажем, с объектом DataFrame из pandas структурные массивы NumPy – средство относительно низкого уровня. Они позволяют интерпретировать блок памяти как табличную структуру с вложенными столбцами произвольной сложности. Поскольку каждый элемент представлен в памяти фиксированным количеством байтов, структурный массив дает очень быстрый и эффективный способ записи данных на диск и чтения с диска (в том числе в файлы, спроецированные на память, о чем речь пойдет ниже), передачи по Сети и прочих операций такого рода.

Еще одно распространенное применение структурных массивов связано со стандартным способом сериализации данных в C и C++, часто встречающимся в унаследованных системах; данные выводятся в файл в виде потока байтов с фиксированной длиной записи. Коль скоро известен формат файла

(размер каждой записи, порядок байтов и тип данных каждого элемента), данные можно прочитать в память методом np.fromfile. Подобные специализированные применения выходят за рамки этой книги, но знать об их существовании полезно.

A.6. Еще о сортировке

Как и у встроенных списков Python, метод sort объекта производит сортировку *на месте*, т. е. массив переупорядочивается без порождения нового массива:

```
In [160]: arr = np.random.randn(6)

In [161]: arr.sort()

In [162]: arr
Out[162]: array([-1.082 , 0.3759, 0.8014, 1.1397, 1.2888, 1.8413])
```

Сортируя на месте, не забывайте, что если сортируемый массив – представление другого массива ndarray, то модифицируется исходный массив:

```
In [163]: arr = np.random.randn(3, 5)

In [164]: arr
Out[164]:
array([[-0.3318, -1.4711, 0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436, 2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424, 0.8548,  1.038 , -0.329 ]])

In [165]: arr[:, 0].sort() # отсортировать значения в первом столбце на месте

In [166]: arr
Out[166]:
array([[-1.0111, -1.4711, 0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436, 2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424, 0.8548,  1.038 , -0.329 ]])
```

С другой стороны, функция numpy.sort создает отсортированную копию массива, принимая те же самые аргументы (в частности, kind, о котором будет сказано ниже), что и метод ndarray.sort:

```
In [167]: arr = np.random.randn(5)

In [167]: arr
Out[167]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])

In [168]: np.sort(arr)
Out[168]: array([-2.0051, -1.1181, -1.0614, -0.2415, 0.7379])

In [169]: arr
Out[169]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])
```


Все методы сортировки принимают аргумент axis, что позволяет независимо сортировать участки массива вдоль указанной оси:

```
In [171]: arr = np.random.randn(3, 5)

In [172]: arr
Out[172]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])

In [173]: arr.sort(axis=1)

In [174]: arr
Out[174]:
array([[-0.2682, -0.1872,  0.5955, 0.9111, 1.3389],
       [-0.5168, -0.3215, -0.1989, 1.0054, 1.1925],
       [-1.7638, -0.2222, -0.2171, 0.3969, 0.6071]])
```

Вероятно, вы обратили внимание, что ни у одного метода нет параметра, который задавал бы сортировку в порядке убывания. Но это не составляет проблемы, потому вырезание массива порождает представления, для чего не требуется копирование или еще какие-то вычисления. Многие пользователи Python знают, что если `values` – список, то `values[::-1]` возвращает его в обратном порядке. То же справедливо и для объектов ndarray:

```
In [175]: arr[:, ::-1]
Out[175]:
array([[ 1.3389, 0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925, 1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071, 0.3969, -0.2171, -0.2222, -1.7638]])
```

Косвенная сортировка: методы argsort и lexsort

В ходе анализа данных очень часто возникает необходимость переупорядочить набор данных по одному или нескольким ключам. Например, отсортировать таблицу, содержащую данные о студентах, сначала по фамилии, а потом по имени. Это пример *косвенной* сортировки, и если вы читали главы, относящиеся к библиотеке pandas, то видели много других примеров более высокого уровня. Имея один или несколько ключей (массив или несколько массивов значений), мы хотим получить массив целочисленных *индексов* (буду называть их просто *индексаторами*), который говорит, как переупорядочить данные в нужном порядке сортировки. Для этого существуют два основных метода: `argsort` и `numpy.lexsort`. Вот пример:

```
In [176]: values = np.array([5, 0, 1, 3, 2])

In [177]: indexer = values.argsort()

In [178]: indexer
Out[178]: array([1, 2, 4, 3, 0])

In [179]: values[indexer]
Out[179]: array([0, 1, 2, 3, 5])
```

А в следующем, более сложном примере двумерный массив переупорядочивается по первой строке:

```
In [180]: arr = np.random.randn(3, 5)

In [181]: arr[0] = values

In [182]: arr
Out[182]:
array([[ 5.    ,  0.    , 1.    ,  3.    , 2.    ],
       [-0.3636, -0.1378, 2.1777, -0.4728, 0.8356],
       [-0.2089,  0.2316, 0.728 , -1.3918, 1.9956]])

In [183]: arr[:, arr[0].argsort()]
Out[183]:
array([[ 0.    , 1.    , 2.    ,  3.    ,  5.    ],
       [-0.1378, 2.1777, 0.8356, -0.4728, -0.3636],
       [ 0.2316, 0.728 , 1.9956, -1.3918, -0.2089]])
```

Метод `lexsort` аналогичен `argsort`, но выполняет косвенную *лексикографическую* сортировку по нескольким массивам ключей. Пусть требуется отсортировать данные, идентифицируемые именем и фамилией:

```
In [184]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])

In [185]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])

In [186]: sorter = np.lexsort((first_name, last_name))

In [187]: sorter
Out[187]: array([1, 2, 3, 0, 4])

In [188]: zip(last_name[sorter], first_name[sorter])
Out[188]: <zip at 0x7fa203eda1c8>
```

При первом использовании метод `lexsort` может вызвать недоумение, потому что первым для сортировки используется ключ, указанный в *последнем* массиве. Как видите, ключ `last_name` использовался раньше, чем `first_name`.

В библиотеке pandas методы `sort_index` объектов Series и DataFrame, а также метод `order` объекта Series реализованы с помощью вариантов этих функций (в которые добавлен учет отсутствующих значений).

Альтернативные алгоритмы сортировки

Устойчивый алгоритм сортировки сохраняет относительные позиции равных элементов. Это особенно важно при косвенной сортировке, когда относительный порядок имеет значение:

```
In [189]: values = np.array(['2:first', '2:second', '1:first', '1:second', '1:third'])

In [190]: key = np.array([2, 2, 1, 1, 1])

In [191]: indexer = key.argsort(kind='mergesort')
```

```
In [192]: indexer
Out[192]: array([2, 3, 4, 0, 1])

In [193]: values.take(indexer)
Out[193]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<U8')
```

Единственный имеющийся устойчивый алгоритм сортировки с гарантированным временем работы O(n log n) – *mergesort*, но его производительность в среднем хуже, чем у алгоритма *quicksort*. В табл. А.3 перечислены имеющиеся алгоритмы, их сравнительное быстродействие и гарантированная производительность. Большинству пользователей эта информация не особенно интересна, но знать о ее существовании стоит.

Таблица А.3. Алгоритмы сортировки массива

Алгоритм	Быстродействие	Устойчивый	Рабочая память	В худшем случае
`'quicksort'`	1	Нет	0	$O(n^2)$
`'mergesort'`	2	Да	n / 2	O(n log n)
`'heapsort'`	3	Нет	0	O(n log n)

Частичная сортировка массивов

Одна из целей сортировки – найти наибольший или наименьший элемент массива. В NumPy имеются оптимизированные методы, `numpy.partition` и `np.argpartition`, для разделения массива по *k*-му наименьшему элементу:

```
In [194]: np.random.seed(12345)

In [195]: arr = np.random.randn(20)

In [196]: arr
Out[196]:
array([-0.2047, 0.4789, -0.5194, -0.5557,  1.9658,  1.3934, 0.0929,
        0.2817, 0.769 ,  1.2464,  1.0072, -1.2962,  0.275 , 0.2289,
        1.3529, 0.8864, -2.0016, -0.3718,  1.669 , -0.4386])

In [197]: np.partition(arr, 3)
Out[197]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

После вызова `partition(arr, 3)` первые три элемента результата – это три наименьших значения в произвольном порядке. Метод `numpy.argpartition`, похожий на `numpy.argsort`, возвращает индексы элементов, определяющие эквивалентный порядок:

```
In [198]: indices = np.argpartition(arr, 3)

In [199]: indices
```

```
Out[199]:
array([16, 11, 3, 2, 17, 19, 0, 7, 8, 1, 10, 6, 12, 13, 14, 15, 5,
       4, 18, 9])

In [200]: arr.take(indices)
Out[200]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
       0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
       1.3529,  0.8864,  1.3934,  1.9658,  1.669 , -1.2464])
```


Метод numpy.searchsorted: поиск элементов в отсортированном массиве

Метод массива `searchsorted` производит двоичный поиск в отсортированном массиве и возвращает место, в которое нужно было бы вставить значение, чтобы массив оставался отсортированным:

```
In [201]: arr = np.array([0, 1, 7, 12, 15])

In [202]: arr.searchsorted(9)
Out[202]: 3
```

Можно передать также массив значений и получить в ответ массив индексов:

```
In [203]: arr.searchsorted([0, 8, 11, 16])
Out[203]: array([0, 3, 3, 5])
```

Вы, наверное, заметили, что `searchsorted` вернул индекс `0` для элемента `0`. Это объясняется тем, что по умолчанию возвращается индекс самого левого из группы элементов с одинаковыми значениями:

```
In [204]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [205]: arr.searchsorted([0, 1])
Out[205]: array([0, 3])

In [206]: arr.searchsorted([0, 1], side='right')
Out[206]: array([3, 7])
```

Чтобы проиллюстрировать еще одно применение метода `searchsorted`, предположим, что имеется массив значений между 0 и 10 000 и отдельный массив «границ интервалов», который мы хотим использовать для распределения данных по интервалам:


```
In [207]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [208]: bins = np.array([0, 100, 1000, 5000, 10000])

In [209]: data
Out[209]:
array([ 9940., 6768., 7908., 1709.,  268., 8003., 9037.,  246.,
        4917., 5262., 5963.,  519., 8950., 7282., 8183., 5002.,
```

```
        8101. , 959., 2189., 2587., 4681., 4593., 7095., 1780.,
        5314., 1677., 7688., 9281., 6094., 1501., 4896., 3773.,
        8486., 9110., 3838., 3154., 5683., 1878., 1258., 6875.,
        7996., 5735., 9732., 6340., 8884., 4954., 3516., 7142.,
        5039., 2256.])
```

Чтобы теперь для каждой точки узнать, какому интервалу она принадлежит (считая, что 1 означает интервал `[0, 100)`), мы можем воспользоваться методом `searchsorted`:

```
In [210]: labels = bins.searchsorted(data)

In [211]: labels
Out[211]:
array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4,
       3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4,
       3, 4, 2, 4])
```

В сочетании с методом `groupby` из библиотеки pandas этого достаточно, чтобы распределить данные по интервалам:

```
In [212]: Series(data).groupby(labels).mean()
Out[212]:
2    498.000000
3   3064.277778
4   7389.035714
dtype: float64
```

A.7. Написание быстрых функций для NumPy с помощью Numba

Numba (http://numba.pydata.org/) – проект с открытым исходным кодом, предназначенный для создания быстрых функций для работы с данными NumPy и похожими на них с использованием CPU, GPU и другого оборудования. Для трансляции написанного на Python кода в машинные команды применяется проект LLVM (http://llvm.org/).

Чтобы составить представление о Numba, рассмотрим функцию на чистом Python, которая вычисляет выражение `(x - y).mean()` в цикле `for`:

```
import numpy as np

def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

Эта функция работает очень медленно:

```
In [209]: x = np.random.randn(10000000)
In [210]: y = np.random.randn(10000000)
In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop
In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

Версия, встроенная в NumPy, быстрее более чем в 100 раз. Мы можем преобразовать написанную нами функцию в откомпилированную функцию Numba, воспользовавшись функцией `numba.jit`:

```
In [213]: import numba as nb
In [214]: numba_mean_distance = nb.jit(mean_distance)
```

Можно было оформить это и в виде декоратора:

```
@nb.jit
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

Получившаяся функция даже быстрее векторной версии в NumPy:

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba не умеет компилировать произвольный код на Python, но поддерживает обширное подмножество Python, наиболее полезное при реализации численных алгоритмов.

Numba – серьезная библиотека, поддерживающая различные виды оборудования, режимы компиляции и пользовательские расширения. Она способна откомпилировать значительное подмножество Python API библиотеки NumPy, не прибегая к явным циклам `for`. Кроме того, Numba умеет распознавать конструкции, допускающие встраивание на машинном коде, а если не знает, как откомпилировать код функции, то подставляет вызовы CPython API. У функции Numba `jit` имеется факультативный аргумент `nopython=True`, который разрешает использовать только такой код на Python, который можно транслировать на LLVM, не прибегая к вызовам Python C API. Вызов `jit(nopython=True)` имеет короткий псевдоним `numba.njit`.

Предыдущий пример можно было бы записать и так:

```
from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

Призываю вас ознакомиться с онлайновой документацией по Numba на сайте http://numba.pydata.org/. В следующем разделе приведен пример создания пользовательской u-функции для NumPy.

Создание пользовательских объектов numpy.ufunc с помощью Numba

Функция `numba.vectorize` создает откомпилированные u-функции NumPy, которые ведут себя так же, как встроенные. Рассмотрим реализацию `numpy.add` на Python:

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

Имеем:

```
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([ 0., 2., 4., 6., 8., 10., 12., 14., 16., 18.])

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([ 0., 1., 3., 6., 10., 15., 21., 28., 36., 45.])
```

A.8. Дополнительные сведения о вводе-выводе массивов

В главе 4 мы познакомились с методами `np.save` и `np.load` для хранения массивов в двоичном формате на диске. Но есть и целый ряд дополнительных возможностей на случай, когда нужно что-то более сложное. В частности, файлы, спроецированные на память, позволяют работать с наборами данных, не умещающимися в оперативной памяти.

Файлы, спроецированные на память

Проецирование файла на память – метод, позволяющий рассматривать потенциально очень большой набор данных на диске как массив в памяти. В NumPy объект `memmap` реализован по аналогии с ndarray, он позволяет читать и записывать небольшие сегменты большого файла, не загружая в память весь массив. Кроме того, у объекта `memmap` точно такие же методы, как

у массива в памяти, поэтому его можно подставить во многие алгоритмы, ожидающие получить ndarray.

Для создания объекта memmap служит функция np.memmap, которой передается путь к файлу, dtype, форма и режим открытия файла:

```
In [214]: mmap = np.memmap('mymmap', dtype='float64', mode='w+', shape=(10000, 10000))

In [215]: mmap
Out[215]:
memmap([[ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        ...,
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.]])
```

При вырезании из memmap возвращается представление данных на диске:

```
In [216]: section = mmap[:5]
```

Если присвоить такому срезу значения, то они буферизуются в памяти (в виде файлового объекта Python) и могут быть записаны на диск позже методом flush:

```
In [217]: section[:] = np.random.randn(5, 10000)

In [218]: mmap.flush()

In [219]: mmap
Out[220]:
memmap([[ 0.7584, -0.6605, 0.8626, ...,  0.6046, -0.6212,  2.0542],
        [-1.2113, -1.0375, 0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375, 0.4329, ...,  1.2914, -0.752 , -0.44  ],
        ...,
        [ 0.    ,  0.    , 0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    , 0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    , 0.    , ...,  0.    ,  0.    ,  0.    ]])

In [220]: del mmap
```

Сброс на диск всех изменений автоматически происходит и тогда, когда объект memmap выходит из области видимости и передается сборщику мусора. При *открытии существующего файла* все равно необходимо указывать dtype и форму, поскольку файл на диске – это просто блок двоичных данных без каких-либо метаданных:

```
In [221]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))

In [222]: mmap
Out[222]:
memmap([[ 0.7584, -0.6605, 0.8626, ...,  0.6046, -0.6212,  2.0542],
```

```
[-1.2113, -1.0375, 0.7093, ..., -1.4117, -0.1719, -0.8957],
[-0.1419, -0.3375, 0.4329, ...,  1.2914, -0.752 , -0.44  ],
...,
[ 0.    ,  0.    , 0.    , ...,  0.    ,  0.    ,  0.    ],
[ 0.    ,  0.    , 0.    , ...,  0.    ,  0.    ,  0.    ],
[ 0.    ,  0.    , 0.    , ...,  0.    ,  0.    ,  0.    ]])
```

Проецирование на память работает также со структурными и вложенными типами dtype, как описано в предыдущем разделе.

HDF5 и другие варианты хранения массива

PyTables и h5py – написанные на Python проекты, в которых реализован ориентированный на NumPy интерфейс для хранения массива в эффективном, допускающем сжатие формате HDF5 (HDF означает *hierarchical data format* – иерархический формат данных). В формате HDF5 можно без опаски хранить сотни гигабайтов и даже терабайты данных. Для получения дополнительных сведений о работе с HDF5 в Python обратитесь к документации по pandas.

A.9. Замечания о производительности

Добиться хорошей производительности программы, написанной с использованием NumPy, обычно нетрудно, поскольку операции над массивами, как правило, заменяют медленные по сравнению с ними циклы на чистом Python. Ниже перечислены некоторые моменты, о которых стоит помнить.

1. Преобразуйте циклы и условную логику Python с операции с массивами и булевыми массивами.
2. Всюду, где только можно, применяйте укладывание.
3. Избегайте копирования данных с помощью представлений массивов (вырезание).
4. Используйте u-функции и их методы.

Если с помощью одних лишь средств NumPy все же никак не удается добиться требуемой производительности, то, возможно, имеет смысл написать часть кода на C, Fortran и особенно на Cython (подробнее об этом ниже). Лично я очень активно использую Cython (http://cython.org) как простой способ получить производительность, сравнимую с C, затратив минимум усилий.

Важность непрерывной памяти

Хотя полное рассмотрение заявленной темы выходит за рамки этой книги, в некоторых приложениях расположение массива в памяти может оказать существенное влияние на скорость вычислений. Отчасти это связано с иерархией процессорных кешей; операции, в которых осуществляется доступ к соседним адресам в памяти (например, суммирование по строкам в массиве,

организованном как в C), обычно выполняются быстрее всего, потому что подсистема памяти буферизует соответствующие участки в сверхбыстром кеше уровня L1 или L2. Кроме того, некоторые ветви написанного на C кода NumPy оптимизированы в расчете на непрерывный случай, когда шагового доступа можно избежать.

Говоря о *непрерывной* организации памяти, мы имеем в виду, что элементы массива хранятся в памяти в том порядке, в котором видны в массиве, организованном по столбцам (как в Fortran) или по строкам (как в C). По умолчанию массивы в NumPy создаются *C-непрерывными*. О массиве, хранящемся по столбцам, например транспонированном C-непрерывном массиве, говорят, что он Fortran-непрерывный. Эти свойства можно явно опросить с помощью атрибута `flags` объекта `ndarray`:

```
In [225]: arr_c = np.ones((1000, 1000), order='C')

In [226]: arr_f = np.ones((1000, 1000), order='F')

In [227]: arr_c.flags
Out[227]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [228]: arr_f.flags
Out[228]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [229]: arr_f.flags.f_contiguous
Out[229]: True
```

В данном случае суммирование строк массива теоретически должно быть быстрее для `arr_c`, чем для `arr_f`, поскольку строки хранятся в памяти непрерывно. Я проверил это с помощью функции `%timeit` в IPython:

```
In [230]: %timeit arr_c.sum(1)
784 us +- 10.4 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)

In [231]: %timeit arr_f.sum(1)
934 us +- 29 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

Часто именно в этом направлении имеет смысл прикладывать усилия, стремясь выжать всю возможную производительность из NumPy. Если мас-

сив размещен в памяти не так, как нужно, можно скопировать его методом copy, передав параметр 'C' или 'F':

```
In [232]: arr_f.copy('C').flags
Out[232]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

При построении представления массива помните, что гарантии непрерывности результата никто не дает:

```
In [233]: arr_c[:50].flags.contiguous
Out[233]: True
```

```
In [234]: arr_c[:, :50].flags
Out[234]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Приложение B. Еще о системе IPython

В главе 2 мы рассмотрели основы оболочки IPython и Jupyter-блокнотов. В этом приложении поговорим о дополнительных возможностях IPython, которые можно взять как из консоли, так и из Jupyter.

B.1. История команд

IPython хранит небольшую базу данных на диске, в которой находятся тексты всех выполненных команд. Она служит нескольким целям:

- поиск, автозавершение и повторное выполнение ранее выполненных команд с минимальными усилиями;
- сохранение истории команд между сеансами;
- протоколирование истории ввода-вывода в файле.

Эти средства больше полезны в оболочке, чем в блокноте, поскольку блокнот изначально хранит всю историю ввода-вывода в каждой ячейке.

Поиск в истории команд и повторное выполнение

Возможность искать и повторно выполнять предыдущие команды для многих является самой полезной функцией. Поскольку IPython рассчитан на итеративную и интерактивную разработку кода, мы часто повторяем одни и те же команды, например `%run`. Допустим, вы выполнили такую команду:

```
In[7]: %run first/second/third/data_script.py
```

и, ознакомившись с результатами работы скрипта (в предположении, что он завершился успешно), обнаружили ошибку в вычислениях. Разобравшись, в чем проблема, и исправив скрипт `data_script.py`, вы можете набрать не-

сколько первых букв команды `%run` и нажать **Ctrl**+**P** или клавишу ↑. В ответ IPython найдет в истории команд первую из предшествующих команд, начинающуюся введенными буквами. При повторном нажатии **Ctrl**+**P** или ↑ поиск будет продолжен. Если вы проскочили мимо нужной команды, ничего страшного. По истории команд можно перемещаться и *вперед* с помощью клавиш **Ctrl**+**N** или ↓. Стоит только попробовать, и вы начнете нажимать эти клавиши, не задумываясь.

Комбинация клавиш **Ctrl**+**R** дает ту же возможность частичного инкрементного поиска, что подсистема `readline`, применяемая в оболочках UNIX, например bash. В Windows функциональность `readline` реализуется самим IPython. Чтобы воспользоваться ею, нажмите **Ctrl**+**R**, а затем введите несколько символов, встречающихся в искомой строке ввода:

```
In [1]: a_command = foo(x, y, z)

(reverse-i-search)`com': a_command = foo(x, y, z)
```

Нажатие **Ctrl**+**R** приводит к циклическому просмотру истории в поисках строк, соответствующих введенным символам.

Входные и выходные переменные

Забыв присвоить результат вызова функции, вы можете горько пожалеть об этом. По счастью, IPython сохраняет ссылки как на входные команды (набранный вами текст), так и на выходные объекты в специальных переменных. Последний и предпоследний выходные объекты хранятся соответственно в переменных _ (один подчерк) и __ (два подчерка):


```
In [24]: 2 ** 27
Out[24]: 134217728

In [25]: _
Out[25]: 134217728
```

Входные команды хранятся в переменных с именами вида `_iX`, где `X` – номер входной строки. Каждой такой входной переменной соответствует выходная переменная `_X`. Поэтому после ввода строки 27 будут созданы две новые переменные `_27` (для хранения выходного объекта) и `_i27` (для хранения входной команды).

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

Поскольку входные переменные – это строки, то их можно повторно вычислить с помощью ключевого слова Python exec:

```
In [30]: exec _i27
```

Есть несколько магических функций, позволяющих работать с историей ввода и вывода. Функция %hist умеет показывать историю ввода полностью или частично, с номерами строк или без них. Функция %reset очищает интерактивное пространство имен и факультативно кеши ввода и вывода. Функция %xdel удаляет все ссылки на *конкретный* объект из внутренних структур данных IPython. Подробнее см. документацию по этим функциям.

Работая с очень большими наборами данных, имейте в виду, что объекты, хранящиеся в истории ввода-вывода IPython, не могут быть удалены из памяти сборщиком мусора, даже если вы удалите соответствующую переменную из интерактивного пространства имен встроенным оператором del. В таких случаях команды %xdel и %reset помогут избежать проблем с памятью.

B.2. Взаимодействие с операционной системой

Еще одна особенность IPython – тесная интеграция с файловой системой и оболочкой операционной системы. Среди прочего это означает, что многие стандартные действия в командной строке можно выполнять в точности так же, как в оболочке Windows или UNIX (Linux, OS X), не выходя из IPython. Речь идет о выполнении команд оболочки, смене рабочего каталога и сохранении результатов команды в объекте Python (строке или списке). Существуют также простые средства для задания псевдонимов команд оболочки и создания закладок на каталоги.

Перечень магических функций и синтаксис вызова команд оболочки представлены в табл. B.1. В следующих разделах я кратко расскажу о них.

Таблица B.1. Команды IPython, относящиеся к операционной системе

Команда	Описание
!cmd	Выполнить команду в оболочке системы
output = !cmd args	Выполнить команду и сохранить в объекте output все выведенное на стандартный вывод
%alias *alias_name cmd*	Определить псевдоним команды оболочки
%bookmark	Воспользоваться системой закладок IPython
%cd *каталог*	Сделать указанный каталог рабочим
%pwd	Вернуть текущий рабочий каталог
%pushd *каталог*	Поместить текущий каталог в стек и перейти в указанный каталог
%popd	Извлечь каталог из стека и перейти в него
%dirs	Вернуть список, содержащий текущее состояние стека каталогов
%dhist	Напечатать историю посещения каталогов
%env	Вернуть переменные среды в виде словаря
%matplotlib	Задать параметры интеграции с matplotlib

Команды оболочки и псевдонимы

Восклицательный знак ! в начале командной строки IPython означает, что все следующее за ним следует выполнить в оболочке системы. Таким образом можно удалять файлы (командой `rm` или `del` в зависимости от ОС), изменять рабочий каталог или исполнять другой процесс.

Все, что команда выводит на консоль, можно сохранить в переменной, присвоив ей значение выражения, начинающегося со знака !. Например, на своей Linux-машине, подключенной к интернету ethernet-кабелем, я могу следующим образом записать в переменную Python свой IP-адрес:

```
In [1]: ip_info = !ifconfig eth0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11 Bcast:10.0.0.255 Mask:255.255.255.0'
```

Возвращенный объект Python `ip_info` – это специализированный список, содержащий различные варианты вывода на консоль.

IPython умеет также подставлять в команды, начинающиеся знаком !, значения переменных Python, определенных в текущем окружении. Для этого имени переменной нужно предпослать знак $:

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py test.py test.xml
```

Магическая функция `%alias` позволяет определять собственные сокращения для команд оболочки, например:

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x 2 root root 69632 2012-01-29 20:36 bin/
drwxr-xr-x 2 root root 4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root 20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x 44 root root 69632 2011-12-26 18:08 lib32/
lrwxrwxrwx 1 root root 3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x 15 root root 4096 2011-10-13 19:03 local/
drwxr-xr-x 2 root root 12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root 12288 2011-11-04 22:53 share/
drwxrwsr-x 24 root src 4096 2011-07-17 18:38 src/
```

Несколько команд можно выполнить как одну, разделив их точками с запятой:

```
In [558]: %alias test_alias (cd ch08; ls; cd ..)

In [559]: test_alias
macrodata.csv spx.csv tips.csv
```

Обратите внимание, что IPython «забывает» все определенные интерактивно псевдонимы после закрытия сеанса. Чтобы создать постоянные псевдонимы, нужно прибегнуть к системе конфигурирования. Она описывается ниже в этой главе.

Система закладок на каталоги

В IPython имеется простая система закладок, позволяющая создавать псевдонимы часто используемых каталогов, чтобы упростить переход в них. Например, пусть требуется создать закладку, указывающую на дополнительные материалы к этой книге:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

После чего с помощью магической команды `%cd` я смогу воспользоваться ранее определенными закладками:

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

Если имя закладки конфликтует с именем подкаталога вашего текущего рабочего каталога, то с помощью флага `-b` можно отдать приоритет закладке. Команда `%bookmark` с флагом `-l` выводит список всех закладок:

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

Закладки, в отличие от псевдонимов, автоматически сохраняются после закрытия сеанса.

B.3. Средства разработки программ

IPython не только является удобной средой для интерактивных вычислений и исследования данных, но и прекрасно оснащен для разработки программ. В приложениях для анализа данных прежде всего важно, чтобы код был *правильным*. К счастью, в IPython встроен отлично интегрированный и улучшенный отладчик Python `pdb`. Кроме того, код должен быть *быстрым*. Для этого в IPython имеются простые в использовании средства хронометража и профилирования. Ниже я расскажу об этих инструментах подробнее.

Интерактивный отладчик

Отладчик IPython дополняет `pdb` завершением по нажатии клавиши **Tab**, подсветкой синтаксиса и контекстом для каждой строки трассировки исключения. Отлаживать программу лучше всего сразу после возникновения ошибки. Команда `%debug`, выполненная сразу после исключения, вызывает

«посмертный» отладчик и переходит в то место стека вызовов, где было возбуждено исключение:

```
In [2]: run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13      throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
     11 def calling_things():
     12      works_fine()
---> 13      throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
      7      a = 5
      8      b = 6
----> 9      assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:

In [3]: %debug
/home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
      8      b = 6
----> 9      assert(a + b == 10)
     10
ipdb>
```

Находясь в отладчике, можно выполнять произвольный Python-код и просматривать все объекты и данные (которые интерпретатор «сохранил живыми») в каждом кадре стека. По умолчанию отладчик оказывается на самом нижнем уровне – там, где произошла ошибка. Клавиши `u` (вверх) и `d` (вниз) позволяют переходить с одного уровня стека на другой:

```
ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
     12      works_fine()
---> 13      throws_an_exception()
     14
```

Команда `%pdb` устанавливает режим, в котором IPython автоматически вызывает отладчик после любого исключения, многие считают этот режим особенно полезным.

Отладчик также помогает разрабатывать код, особенно когда хочется расставить точки останова либо пройти функцию или скрипт в пошаговом ре-

жиме, изучая состояния после каждого шага. Сделать это можно несколькими способами. Первый – воспользоваться функцией %run с флагом -d, которая вызывает отладчик, перед тем как начать выполнение кода в переданном скрипте. Для входа в скрипт нужно сразу же нажать s (step – пошаговый режим):

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

После этого вы сами решаете, каким образом работать с файлом. Например, в приведенном выше примере исключения можно было бы поставить точку останова прямо перед вызовом метода works_fine и выполнить программу до этой точки, нажав c (continue – продолжить):

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_things()
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

В этот момент можно войти внутрь works_fine() командой step или выполнить works_fine() без захода внутрь, т. е. перейти к следующей строке, нажав n (next – дальше):

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

Далее мы можем войти внутрь throws_an_exception, дойти до строки, где возникает ошибка, и изучить переменные в текущей области видимости. Отметим, что у команд отладчика больший приоритет, чем у имен переменных, поэтому для просмотра переменной с таким же именем, как у команды, необходимо предпослать ей знак !.

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(6)throws_an_exception()
      5
```

```
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exception()
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9 assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6
```

Уверенное владение интерактивным отладчиком приходит с опытом и практикой. В табл. B.2 приведен полный перечень команд отладчика. Если вы привыкли к IDE, то консольный отладчик на первых порах может показаться неуклюжим, но со временем это впечатление рассеется. В некоторых IDE для Python имеются отличные графические отладчики, так что всякий пользователь найдет что-то себе по вкусу.

Таблица B.2. Команды отладчика (I)Python

Команда	Действие
h(elp)	Вывести список команд
help *команда*	Показать документацию по *команде*
c(ontinue)	Продолжить выполнение программы
q(uit)	Выйти из отладчика, прекратив выполнение кода
b(reak) *номер*	Поставить точку останова на строке с указанным *номером* в текущем файле
b *путь/к/файлу.ру:номер*	Поставить точку останова на строке с указанным *номером* в указанном файле
s(tep)	Войти внутрь функции
n(ext)	Выполнить текущую строку и перейти к следующей на текущем уровне
u(p) / d(own)	Перемещение вверх и вниз по стеку вызовов
a(rgs)	Показать аргументы текущей функции
debug *предложение*	Выполнить *предложение* в новом (вложенном) отладчике
l(ist) *предложение*	Показать текущую позицию и контекст на текущем уровне стека
w(here)	Распечатать весь стек в контексте текущей позиции

Другие способы работы с отладчиком

Существует еще два полезных способа вызова отладчика. Первый – воспользоваться специальной функцией set_trace (названной так по аналогии с pdb.set_trace), которая по существу является упрощенным вариантом точки останова. Вот два небольших фрагмента, которые вы можете сохранить где-нибудь и использовать в разных программах (я, например, помещаю их в свой профиль IPython):

```
from IPython.core.debugger import Pdb

def set_trace():
    from IPython.core.debugger import Pdb
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

Первая функция, set_trace, совсем простая. Вызывайте ее в той точке кода, где хотели бы остановиться и оглядеться (например, прямо перед строкой, в которой происходит исключение):

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py(16)calling_things()
     15      set_trace()
---> 16      throws_an_exception()
     17
```

При нажатии c (продолжить) выполнение программы возобновится без каких-либо побочных эффектов.

Функция debug позволяет вызвать интерактивный отладчик в момент обращения к любой функции. Допустим, мы написали такую функцию и хотели бы пройти ее в пошаговом режиме:

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

Обычно f используется примерно так: f(1, 2, z=3). А чтобы войти в эту функцию, передайте f в качестве первого аргумента функции debug, а затем ее позиционные и именованные аргументы:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z
ipdb>
```

Мне эти две простенькие функции ежедневно экономят уйму времени.

Наконец, отладчик можно использовать в сочетании с функцией %run. Запустив скрипт командой %run -d, вы попадете прямо в отладчик и сможете расставить точки останова и начать выполнение:

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb>
```

Если добавить еще флаг -b, указав номер строки, то после входа в отладчик на этой строке уже будет стоять точка останова:

```
In [2]: %run -d -b2 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(2)works_fine()
      1 def works_fine():
1---> 2     a = 5
      3     b = 6

ipdb>
```

Хронометраж программы: %time и %timeit

Для больших или долго работающих аналитических приложений бывает желательно измерить время выполнения различных участков кода или даже отдельных предложений или вызовов функций. Интересно получить отчет о том, какие функции занимают больше всего времени в сложном процессе. По счастью, IPython позволяет без труда получить эту информацию по ходу разработки и тестирования программы.

Ручной хронометраж с помощью встроенного модуля time и его функций time.clock и time.time зачастую оказывается скучной и утомительной процедурой, поскольку приходится писать один и тот же неинтересный код:

```
import time
start = time.time()
for i in range(iterations):
    # здесь код, который требует хронометрировать
elapsed_per = (time.time() - start) / iterations
```

Так как эта операция встречается очень часто, в IPython есть две магические функции, %time и %timeit, которые помогают автоматизировать процесс.

Функция %time выполняет предложение один раз и сообщает, сколько было затрачено времени. Допустим, имеется длинный список строк и мы хотим сравнить различные методы выбора всех строк, начинающихся с заданного

префикса. Вот простой список, содержащий 700 000 строк, и два метода выборки тех, что начинаются с 'foo':

```
# очень длинный список строк
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

На первый взгляд производительность должна быть примерно одинаковой, верно? Проверим с помощью функции %time:

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

Наибольший интерес представляет величина Wall time (фактическое время). Похоже, первый метод работает в два раза медленнее второго, но это не очень точное измерение. Если вы несколько раз сами замерите время работы этих двух предложений, то убедитесь, что результаты варьируются. Для более точного измерения воспользуемся магической функцией %timeit. Она получает произвольное предложение и, применяя внутренние эвристики, выполняет его столько раз, сколько необходимо для получения более точного среднего времени:

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

Этот на первый взгляд безобидный пример показывает, насколько важно хорошо понимать характеристики производительности стандартной библиотеки Python, NumPy, pandas и других используемых в книге библиотек. В больших приложениях для анализа данных из миллисекунд складываются часы!

Функция %timeit особенно полезна для анализа предложений и функций, работающих очень быстро, за микросекунды (10^{-6} секунд) или наносекунды (10^{-9} секунд). Вроде бы совсем мизерные промежутки времени, но если функцию, работающую 20 микросекунд, вызвать 1 000 000 раз, то будет потрачено на 15 секунд больше, чем если бы она работала всего 5 микросекунд. В примере выше можно сравнить две операции со строками напрямую, это даст отчетливое представление об их характеристиках в плане производительности:

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

Простейшее профилирование: %prun и %run -p

Профилирование кода тесно связано с хронометражем, только отвечает на вопрос, *где именно* тратится время. В Python основное средство профилирования – модуль `cProfile`, который предназначен отнюдь не только для IPython. `cProfile` исполняет программу или произвольный блок кода и следит за тем, сколько времени проведено в каждой функции.

Обычно `cProfile` запускают из командной строки, профилируют программу целиком и выводят агрегированные временные характеристики каждой функции. Пусть имеется простой скрипт, который выполняет в цикле какой-нибудь алгоритм линейной алгебры (скажем, вычисляет максимальное по абсолютной величине собственное значение для последовательности матриц размерности 100×100):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in range(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print('Самое большое встретившееся: {0}'.format(np.max(some_results)))
```

Этот скрипт можно запустить под управлением `cProfile` из командной строки следующим образом:

```
python -m cProfile cprof_example.py
```

Попробуйте и убедитесь, что результаты отсортированы по имени функции. Такой отчет не позволяет сразу увидеть, где тратится время, поэтому обычно *порядок сортировки* задают с помощью флага `-s`:

```
$ python -m cProfile -s cumulative cprof_example.py
```

Самое большое встретившееся: 11.923204422.

```
     15116 function calls (14927 primitive calls) in 0.720 seconds
```

```
Ordered by: cumulative time

ncalls  tottime percall  cumtime percall  filename:lineno(function)
     1    0.001   0.001    0.721   0.721  cprof_example.py:1(<module>)
   100    0.003   0.000    0.586   0.006  linalg.py:702(eigvals)
   200    0.572   0.003    0.572   0.003  {numpy.linalg.lapack_lite.dgeev}
     1    0.002   0.002    0.075   0.075  __init__.py:106(<module>)
   100    0.059   0.001    0.059   0.001  {method 'randn')
     1    0.000   0.000    0.044   0.044  add_newdocs.py:9(<module>)
     2    0.001   0.001    0.037   0.019  __init__.py:1(<module>)
     2    0.003   0.002    0.030   0.015  __init__.py:2(<module>)
     1    0.000   0.000    0.030   0.030  type_check.py:3(<module>)
     1    0.001   0.001    0.021   0.021  __init__.py:15(<module>)
     1    0.013   0.013    0.013   0.013  numeric.py:1(<module>)
     1    0.000   0.000    0.009   0.009  __init__.py:6(<module>)
     1    0.001   0.001    0.008   0.008  __init__.py:45(<module>)
   262    0.005   0.000    0.007   0.000  function_base.py:3178(add_newdoc)
   100    0.003   0.000    0.005   0.000  linalg.py:162(_assertFinite)
   ...
```

Показаны только первые 15 строк отчета. Читать его проще всего, просматривая сверху вниз столбец `cumtime`, чтобы понять, сколько времени было проведено *внутри* каждой функции. Отметим, что если одна функция вызывает другую, то *таймер не останавливается*. `cProfile` запоминает моменты начала и конца каждого вызова функции и на основе этих данных создает отчет о затраченном времени.

`cProfile` можно запускать не только из командной строки, но и программно для профилирования работы произвольных блоков кода без порождения нового процесса. В IPython имеется удобный интерфейс к этой функциональности в виде команды `%prun` и команды `%run` с флагом `-p`. Команда `%prun` принимает те же «аргументы командной строки», что и `cProfile`, но профилирует произвольное предложение Python, а не py-файл:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
        4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime percall cumtime  percall  filename:lineno(function)
     1    0.000   0.000   0.643    0.643  <string>:1(<module>)
     1    0.001   0.001   0.643    0.643  cprof_example.py:4(run_experiment)
   100    0.003   0.000   0.583    0.006  linalg.py:702(eigvals)
   200    0.569   0.003   0.569    0.003  {numpy.linalg.lapack_lite.dgeev}
   100    0.058   0.001   0.058    0.001  {method 'randn'}
   100    0.003   0.000   0.005    0.000  linalg.py:162(_assertFinite)
   200    0.002   0.000   0.002    0.000  {method 'all' of 'numpy.ndarray' objects}
```

Аналогично команда `%run -p -s cumulative cprof_example.py` дает тот же результат, что рассмотренный выше запуск из командной строки, только не приходится выходить из IPython.

В Jupyter-блокноте для профилирования целого блока кода можно использовать магическую команду %%prun (два знака %). Она открывает отдельное окно, в которое выводится профиль. Это полезно для быстрого ответа на вопросы типа «Почему этот блок так долго работает?».

Существуют и другие инструменты, которые помогают интерпретировать профиль при работе с IPython или Jupyter. Один из них – SnakeViz (https://github.com/jiffyclub/snakeviz/) – порождает интерактивную визуализацию результатов профилирования с помощью библиотеки d3.js.

Построчное профилирование функции

Иногда информации, полученной от %prun (или добытой иным способом профилирования на основе cProfile), недостаточно, чтобы составить полное представление о времени работы функции. Или она настолько сложна, что результаты, агрегированные по имени функции, с трудом поддаются интерпретации. На такой случай есть небольшая библиотека line_profiler (ее поможет установить PyPI или любой другой инструмент управления пакетами). Она содержит расширение IPython, включающее новую магическую функцию %lprun, которая строит построчный профиль выполнения одной или нескольких функций. Чтобы подключить это расширение, нужно модифицировать конфигурационный файл IPython (см. документацию по IPython или раздел, посвященный конфигурированию, ниже), добавив такую строку:

```
# Список имен загружаемых модулей с расширениями IPython.
c.TerminalIPythonApp.extensions = ['line_profiler']
```

Библиотеку line_profiler можно использовать из программы (см. полную документацию), но, пожалуй, наиболее эффективна интерактивная работа с ней в IPython. Допустим, имеется модуль prof_mod, содержащий следующий код, в котором выполняются операции с массивом NumPy:

```
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

Если бы нам нужно было оценить производительность функции add_and_sum, то команда %prun дала бы такие результаты:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)
```

```
In [572]: %prun add_and_sum(x, y)
          4 function calls in 0.049 seconds
   Ordered by: internal time
   ncalls tottime percall cumtime percall filename:lineno(function)
        1   0.036   0.036   0.046   0.046 prof_mod.py:3(add_and_sum)
        1   0.009   0.009   0.009   0.009 {method 'sum' of 'numpy.ndarray' objects}
        1   0.003   0.003   0.049   0.049 <string>:1(<module>)
```

Не слишком полезно. Но после активации расширения IPython `line_profiler` становится доступна новая команда `%lprun`. От `%prun` она отличается только тем, что мы указываем, какую функцию (или функции) хотим профилировать. Порядок вызова такой:

```
%lprun -f func1 -f func2 профилируемое_предложение
```

В данном случае мы хотим профилировать функцию `add_and_sum`, поэтому пишем:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line     # Hits     Time   Per Hit   % Time  Line Contents
==============================================================
   3                                          def add_and_sum(x, y):
   4          1   36510    36510.0     79.5      added = x + y
   5          1    9425     9425.0     20.5      summed = added.sum(axis=1)
   6          1       1        1.0      0.0      return summed
```

Так гораздо понятнее. В этом примере мы профилировали ту же функцию, которая составляла предложение. Но можно было бы вызвать функцию `call_function` из показанного выше модуля и профилировать ее наряду с `add_and_sum`, это дало бы полную картину производительности кода:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line     # Hits     Time   Per Hit   % Time  Line Contents
==============================================================
   3                                          def add_and_sum(x, y):
   4          1     4375    4375.0     79.2      added = x + y
   5          1     1149    1149.0     20.8      summed = added.sum(axis=1)
   6          1       2      2.0        0.0      return summed
File: book_scripts/prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
```

```
Line     # Hits     Time   Per Hit   % Time  Line Contents
==============================================================
   8                                          def call_function():
   9          1    57169   57169.0     47.2      x = randn(1000, 1000)
  10          1    58304   58304.0     48.2      y = randn(1000, 1000)
  11          1     5543    5543.0      4.6      return add_and_sum(x, y)
```

Обычно я предпочитаю использовать `%prun` (`cProfile`) для «макропрофилирования», а `%lprun` (`line_profiler`) – для «микропрофилирования». Полезно освоить оба инструмента.

Явно указывать имена подлежащих профилированию функций в команде `%lprun` необходимо, потому что накладные расходы на трассировку выполнения каждой строки весьма значительны. Трассировка функций, не представляющих интереса, может существенно изменить результаты профилирования.

B.4. Советы по продуктивной разработке кода с использованием IPython

Создание кода таким образом, чтобы его можно было разрабатывать, отлаживать и в конечном счете *использовать* интерактивно, многим может показаться сменой парадигмы. Придется несколько изменить подходы к таким процедурным деталям, как перезагрузка кода, а также сам стиль кодирования.

Поэтому реализация стратегий, описанных в этом разделе, – скорее искусство, чем наука, вы должны будете экспериментально определить наиболее эффективный для себя способ написания Python-кода. Конечная задача – структурировать код так, чтобы с ним было легко работать интерактивно и изучать результаты прогона всей программы или отдельной функции с наименьшими усилиями. Я пришел к выводу, что программу, спроектированную в расчете на IPython, использовать проще, чем аналогичную, но построенную как автономное командное приложение. Это становится особенно важно, когда возникает какая-то проблема и нужно найти ошибку в коде, написанном вами или кем-то еще несколько месяцев или лет назад.

Перезагрузка зависимостей модуля

Когда в Python-программе впервые встречается предложение `import some_lib`, выполняется код из модуля `some_lib` и все переменные, функции и импортированные модули сохраняются во вновь созданном пространстве имен модуля `some_lib`. При следующей обработке предложения `import some_lib` будет возвращена ссылка на уже существующее пространство имен модуля. При интерактивной разработке кода возникает проблема: как быть, когда, скажем, с помощью команды `%run` выполняется скрипт, зависящий от другого модуля, в который вы внесли изменения? Допустим, в файле *test_script.py* находится такой код:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

Если выполнить `%run test_script.py`, а затем изменить *some_lib.py*, то при следующем выполнении `%run test_script.py` мы получим *старую версию some_lib.py* из-за принятого в Python механизма однократной загрузки. Такое поведение отличается от некоторых других сред анализа данных, например от MATLAB, в которых изменения кода распространяются автоматически[1]. Справиться с этой проблемой можно двумя способами. Во-первых, использовать функцию `reload` из модуля `importlib` стандартной библиотеки:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

При этом гарантируется получение новой копии *some_lib.py* при каждом запуске *`test_script.py`*. Очевидно, что если глубина вложенности зависимостей больше единицы, то вставлять `reload` повсюду становится утомительно. Поэтому в IPython имеется специальная функция `dreload` (*не* магическая), выполняющая «глубокую» (рекурсивную) перезагрузку модулей. Если в файле *some_lib.py* имеется предложение `dreload(some_lib)`, то интерпретатор постарается перезагрузить как модуль `some_lib`, так и все его зависимости. К сожалению, это работает не во всех случаях, но если работает, то оказывается куда лучше перезапуска всего IPython.

Советы по проектированию программ

Простых рецептов здесь нет, но некоторыми общими соображениями, которые лично мне кажутся эффективными, я все же поделюсь.

Сохраняйте ссылки на нужные объекты и данные

Программы, рассчитанные на запуск из командной строки, нередко структурируются, как показано в следующем тривиальном примере:

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
```


[1] Поскольку модуль или пакет может импортироваться в нескольких местах программы, Python кеширует код модуля при первом импортировании, а не выполняет его каждый раз. В противном случае следование принципам модульности и правильной организации кода могло бы поставить под угрозу эффективность приложения.

```
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

Вы уже видите, что случится, если эту программу запустить в IPython? После ее завершения все результаты или объекты, определенные в функции `main`, будут недоступны в оболочке IPython. Лучше, если любой код, находящийся в `main`, будет исполняться прямо в глобальном пространстве имен модуля (или в блоке `if __name__ == '__main__':`, если вы хотите, чтобы и сам модуль был импортируемым). Тогда после выполнения кода командой `%run` сможете просмотреть все переменные, определенные в `main`. Это эквивалентно определению переменных верхнего уровня в ячейках Jupyter-блокнота.

Плоское лучше вложенного

Глубоко вложенный код напоминает мне луковицу. Сколько чешуй придется снять при тестировании или отладке функции, чтобы добраться до интересующего кода? Идея «плоское лучше вложенного» – часть «Дзен Python», применимая и к разработке кода, предназначенного для интерактивного использования. Чем более модульными являются классы и функции и чем меньше связей между ними, тем проще их тестировать (если вы пишете автономные тесты), отлаживать и использовать интерактивно.

Перестаньте бояться длинных файлов

Если вы раньше работали с Java (или аналогичным языком), то, наверное, вам говорили, что чем файл короче, тем лучше. Во многих языках это разумный совет; длинный файл несет в себе дурной запашок и наводит на мысль о необходимости рефакторинга или реорганизации. Однако при разработке кода в IPython наличие десяти мелких (скажем, не более чем из 100 строчек) взаимосвязанных файлов с большей вероятностью вызовет проблемы, чем при работе всего с одним, двумя или тремя файлами подлиннее. Чем меньше файлов, тем меньше нужно перезагружать модулей и тем реже приходится переходить от файла к файлу в процессе редактирования. Я пришел к выводу, что сопровождение крупных модулей с высокой степенью *внутренней* сцепленности гораздо полезнее и лучше соответствует духу Python. По мере приближения к окончательному решению, возможно, имеет смысл разбить большие файлы на мелкие.

Понятно, что я не призываю бросаться из одной крайности в другую, т. е. помещать весь код в один гигантский файл. Для отыскания разумной и интуитивно очевидной структуры модулей и пакетов, составляющих большую программу, нередко приходится потрудиться, но при коллективной работе это очень важно. Каждый модуль должен обладать внутренней сцепленностью, а местонахождение функций и классов, относящихся к каждой области функциональности, должно быть как можно более очевидным.

B.5. Дополнительные возможности IPython

Если вы решите в полной мере задействовать систему IPython, то, наверное, станете писать код немного иначе или придется залезать в дебри конфигурационных файлов.

Делайте классы дружественными к IPython

В IPython предпринимаются все меры к тому, чтобы вывести на консоль понятное строковое представление инспектируемых объектов. Для многих объектов, в частности словарей, списков и кортежей, красивое форматирование обеспечивается за счет встроенного модуля `pprint`. Однако в классах, определенных пользователем, порождение строкового представления возлагается на автора. Рассмотрим такой простенький класс:

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

Вы будете разочарованы тем, как такой класс распечатывается по умолчанию:

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython принимает строку, возвращенную магическим методом `__repr__` (выполняя предложение `output = repr(obj)`), и выводит ее на консоль. Но раз так, то мы можем включить в класс простой метод `__repr__`, который создает более полезное представление:

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg

In [579]: x = Message('У меня есть секрет')

In [580]: x
Out[580]: Message: У меня есть секрет
```

Профили и конфигурирование

Многие аспекты внешнего вида (цвета, приглашение, расстояние между строками и т. д.) и поведения оболочки IPython настраиваются с помощью развитой системы конфигурирования. Приведем лишь несколько примеров того, что можно сделать.

1. Изменить цветовую схему.
2. Изменить вид приглашений ввода и вывода или убрать пустую строку, печатаемую после `Out` и перед следующим приглашением `In`.
3. Выполнить список произвольных предложений Python. Это может быть, например, импорт постоянно используемых модулей или вообще все, что должно выполняться сразу после запуска IPython.
4. Включить расширения IPython, например магическую функцию `%lprun` в модуле `line_profiler`.
5. Включить расширения Jupyter.
6. Определить собственные магические функции или псевдонимы системных.

Конфигурационные параметры задаются в файле *ipython_config.py*, находящемся в подкаталоге *.ipython/* вашего домашнего каталога. Конфигурирование производится на основе конкретного *профиля*. При обычном запуске IPython загружается *профиль по умолчанию*, который хранится в каталоге *profile_default*. Следовательно, в моей Linux-системе полный путь к конфигурационному файлу IPython по умолчанию будет таким:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

Для инициализации этого файла в своей системе выполните в терминале команду

```
ipython profile create
```

Не стану останавливаться на технических деталях содержимого этого файла. По счастью, все параметры в нем подробно прокомментированы, так что оставляю их изучение и изменение читателю. Еще одна полезная возможность – поддержка сразу *нескольких профилей*. Допустим, имеется альтернативная конфигурация IPython для конкретного приложения или проекта. Чтобы создать новый профиль, нужно всего лишь ввести такую строку:

```
ipython profile create secret_project
```

Затем отредактируйте конфигурационные файлы во вновь созданном каталоге *profile_secret_project* и запустите IPython следующим образом:

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016, 05:22:56)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

IPython profile: secret_project
```

Как всегда, дополнительные сведения о профилях и конфигурировании можно найти в документации по IPython в Сети.

Для Jupyter конфигурирование устроено несколько иначе, потому что его блокноты можно использовать не только с Python, но и с другими языками. Чтобы создать аналогичный конфигурационный файл Jupyter, выполните команду:

```
jupyter notebook --generate-config
```

Она создаст конфигурационный файл по умолчанию в подкаталоге *.jupyter/jupyter_notebook_config.py* вашего начального каталога. Вы можете отредактировать его и переименовать, например:


```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

Тогда при запуске Jupyter добавьте аргумент `--config`:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

B.6. Заключение

Когда вы будете прорабатывать примеры кода в этой книге и расширять свои навыки программирования на Python, рекомендую постоянно интересоваться экосистемами IPython и Jupyter. Эти проекты создавались специально, чтобы повысить продуктивность пользователя, поэтому работать с ними проще, чем на самом языке Python с его вычислительными библиотеками.

А на сайте nbviewer (https://nbviewer.jupyter.org/) вы найдете много интересных Jupyter-блокнотов.

Предметный указатель

Д

З

И

К

Л

М

Т

У

Ф

F

G

H

I

J

K

L

Q

R

S

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Уэс Маккини

Python и анализ данных

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И., Юрьева В. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 43,88. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**