

Python и анализ данных



Уэс Маккинни



O'REILLY®

Уэс Маккинли

Python и анализ данных

Python for Data Analysis

Wes McKinney

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

УДК 004.438Python:004.6

ББК 32.973.22

M15

M15 Уэс Маккинли

Python и анализ данных / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2015. – 482 с.: ил.

ISBN 978-5-97060-315-4

Книгу можно рассматривать как современное практическое введение в разработку научных приложений на Python, ориентированных на обработку данных. Описаны те части языка Python и библиотеки для него, которые необходимы для эффективного решения широкого круга аналитических задач: интерактивная оболочка IPython, библиотеки NumPy и pandas, библиотека для визуализации данных matplotlib и др.

Издание идеально 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

Original English language edition published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. Copyright © 2013 O'Reilly Media, Inc. Russian-language edition copyright © 2015 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-449-31979-3 (англ.)
ISBN 978-5-97060-315-4 (рус.)

Copyright © 2013 Wes McKinney.
© Оформление, перевод на русский язык,
издание, ДМК Пресс, 2015



ОГЛАВЛЕНИЕ

Предисловие	12
Графические выделения	12
Глава 1. Предварительные сведения.....	13
О чем эта книга?	13
Почему именно Python?	14
Python как клей.....	14
Решение проблемы «двух языков».....	15
Недостатки Python	15
Необходимые библиотеки для Python.....	16
NumPy.....	16
pandas.....	16
matplotlib.....	17
IPython	17
SciPy	18
Установка и настройка	18
Windows	19
Apple OS X	21
GNU/Linux	22
Python 2 и Python 3	23
Интегрированные среды разработки (IDE)	24
Сообщество и конференции	24
Структура книги	25
Примеры кода	25
Данные для примеров	25
Соглашения об импорте	25
Жаргон.....	26
Благодарности	26
Глава 2. Первые примеры	28
Набор данных 1.usa.gov с сайта bit.ly	28
Подсчет часовых поясов на чистом Python	30
Подсчет часовых поясов с помощью pandas	32
Набор данных MovieLens 1M	38
Измерение несогласия в оценках	42
Имена, которые давали детям в США за период с 1880 по 2010 год.....	43
Анализ тенденций в выборе имен.....	48
Выводы и перспективы	56

Глава 3. IPython: интерактивные вычисления и среда разработки	57
Основы IPython	58
Завершение по нажатию клавиши Tab	59
Интропекция	60
Команда %run	61
Исполнение кода из буфера обмена.....	63
Комбинации клавиш.....	64
Исключения и обратная трассировка	65
Магические команды.....	66
Графическая консоль на базе Qt.....	68
Интеграция с matplotlib и режим pylab	68
История команд.....	70
Поиск в истории команд и повторное выполнение.....	70
Входные и выходные переменные	71
Протоколирование ввода-вывода	72
Взаимодействие с операционной системой	73
Команды оболочки и псевдонимы	73
Система закладок на каталоги.....	75
Средства разработки программ	75
Интерактивный отладчик.....	75
Хронометраж программы: %time и %timeit	80
Простейшее профилирование: %prun и %run -r.....	82
Построчное профилирование функции	83
HTML-блокнот в IPython	86
Советы по продуктивной разработке кода с использованием IPython ...	86
Перезагрузка зависимостей модуля	87
Советы по проектированию программ	88
Дополнительные возможности IPython	90
Делайте классы дружественными к IPython	90
Профили и конфигурирование	90
Благодарности	92
Глава 4. Основы NumPy: массивы и векторные вычисления ...	93
NumPy ndarray: объект многомерного массива	94
Создание ndarray	95
Тип данных для ndarray	97
Операции между массивами и скалярами	100
Индексирование и вырезание	100
Булево индексирование	104
Прихотливое индексирование	107
Транспонирование массивов и перестановка осей.....	108
Универсальные функции: быстрые поэлементные операции над массивами.....	109
Обработка данных с применением массивов	112
Запись логических условий в виде операций с массивами.....	113

Математические и статистические операции	115
Методы булевых массивов	116
Сортировка	117
Устранение дубликатов и другие теоретико-множественные операции.....	118
Файловый ввод-вывод массивов	119
Хранение массивов на диске в двоичном формате.....	119
Сохранение и загрузка текстовых файлов	120
Линейная алгебра.....	121
Генерация случайных чисел	122
Пример: случайное блуждание.....	123
Моделирование сразу нескольких случайных блужданий	125
Глава 5. Первое знакомство с pandas	127
Введение в структуры данных pandas	128
Объект Series	128
Объект DataFrame	131
Индексные объекты.....	137
Базовая функциональность.....	139
Переиндексация	139
Удаление элементов из оси	142
Доступ по индексу, выборка и фильтрация	143
Арифметические операции и выравнивание данных	146
Применение функций и отображение	150
Сортировка и ранжирование	151
Индексы по осям с повторяющимися значениями	154
Редукция и вычисление описательных статистик	155
Корреляция и ковариация	158
Уникальные значения, счетчики значений и членство	160
Обработка отсутствующих данных	162
Фильтрация отсутствующих данных	163
Иерархическое индексирование.....	166
Уровни переупорядочения и сортировки	169
Сводная статистика по уровню	170
Работа со столбцами DataFrame.....	170
Другие возможности pandas	172
Доступ по целочисленному индексу	172
Структура данных Panel.....	173
Глава 6. Чтение и запись данных, форматы файлов	175
Чтение и запись данных в текстовом формате	175
Чтение текстовых файлов порциями	181
Вывод данных в текстовом формате.....	182
Ручная обработка данных в формате с разделителями.....	184
Данные в формате JSON	186
XML и HTML: разбор веб-страниц	188
Разбор XML с помощью lxml.objectify	190
Двоичные форматы данных	192

Взаимодействие с HTML и Web API.....	194
Взаимодействие с базами данных	196
Чтение и сохранение данных в MongoDB.....	198
Глава 7. Переформатирование данных: очистка, преобразование, слияние, изменение формы	199
Комбинирование и слияние наборов данных	199
Слияние объектов DataFrame как в базах данных.....	200
Слияние по индексу	204
Конкатенация вдоль оси.....	207
Комбинирование перекрывающихся данных	211
Изменение формы и поворот	212
Изменение формы с помощью иерархического индексирования	213
Поворот из «длинного» в «широкий» формат	215
Преобразование данных	217
Устранение дубликатов	217
Преобразование данных с помощью функции или отображения	218
Замена значений.....	220
Переименование индексов осей	221
Дискретизация и раскладывание	222
Обнаружение и фильтрация выбросов	224
Перестановки и случайная выборка	226
Вычисление индикаторных переменных.....	227
Манипуляции со строками	229
Методы строковых объектов	230
Регулярные выражения	232
Векторные строковые функции в pandas	235
Пример: база данных о продуктах питания министерства сельского хозяйства США	237
Глава 8. Построение графиков и визуализация	244
Краткое введение в API библиотеки matplotlib.....	245
Рисунки и подграфики.....	246
Цвета, маркеры и стили линий	249
Риски, метки и надписи	251
Аннотации и рисование в подграфике	254
Сохранение графиков в файле	256
Конфигурирование matplotlib	257
Функции построения графиков в pandas	258
Линейные графики	258
Столбчатые диаграммы.....	260
Гистограммы и графики плотности.....	264
Диаграммы рассеяния	266
Нанесение данных на карту: визуализация данных о землетрясении на Гаити	267
Инструментальная экосистема визуализации для Python	273

Chaco	274
mayavi	274
Прочие пакеты	275
Будущее средство визуализации	275
Глава 9. Агрегирование данных и групповые операции.....	276
Механизм GroupBy	277
Обход групп	280
Выборка столбца или подмножества столбцов.....	281
Группировка с помощью словарей и объектов Series.....	282
Группировка с помощью функций.....	284
Группировка по уровням индекса	284
Агрегирование данных.....	285
Применение функций, зависящих от столбца, и нескольких функций.....	287
Возврат агрегированных данных в «неиндексированном» виде	289
Групповые операции и преобразования.....	290
Метод apply: часть общего принципа разделения–применения–объединения	292
Квантильный и интервальный анализ	294
Пример: подстановка зависящих от группы значений вместо отсутствующих.....	296
Пример: случайная выборка и перестановка	297
Пример: групповое взвешенное среднее и корреляция.....	299
Пример: групповая линейная регрессия	301
Сводные таблицы и кросс-табуляция.....	302
Таблицы сопряженности	304
Пример: база данных федеральной избирательной комиссии за 2012 год	305
Статистика пожертвований по роду занятий и месту работы	308
Распределение суммы пожертвований по интервалам.....	311
Статистика пожертвований по штатам	313
Глава 10. Временные ряды.....	316
Типы данных и инструменты, относящиеся к дате и времени	317
Преобразование между строкой и datetime	318
Основы работы с временными рядами	321
Индексирование, выборка, подмножества	322
Временные ряды с неуникальными индексами.....	324
Диапазоны дат, частоты и сдвиг	325
Генерация диапазонов дат	325
Частоты и смещения дат	326
Сдвиг данных (с опережением и с запаздыванием)	329
Часовые пояса	331
Локализация и преобразование	332
Операции над объектами Timestamp с учетом часового пояса	333
Операции между датами из разных часовых поясов	334
Периоды и арифметика периодов	335
Преобразование частоты периода	336

Квартальная частота периода	337
Преобразование временных меток в периоды и обратно	339
Создание PeriodIndex из массивов	340
Передискретизация и преобразование частоты.....	341
Понижающая передискретизация	342
Повышающая передискретизация и интерполяция	345
Передискретизация периодов	346
Графики временных рядов.....	348
Скользящие оконные функции	350
Экспоненциально взвешенные функции.....	353
Бинарные скользящие оконные функции.....	353
Скользящие оконные функции, определенные пользователем.....	355
Замечания о быстродействии и потреблении памяти	356
Глава 11. Финансовые и экономические приложения	358
О переформатировании данных	358
Временные ряды и выравнивание срезов.....	358
Операции над временными рядами с различной частотой	361
Время суток и выборка данных «по состоянию на»	364
Сращивание источников данных	366
Индексы доходности и кумулятивная доходность.....	368
Групповые преобразования и анализ	370
Оценка воздействия групповых факторов	372
Децильный и квартильный анализ	373
Другие примеры приложений	375
Стохастический граничный анализ	375
Роллинг фьючерсных контрактов.....	377
Скользящая корреляция и линейная регрессия.....	380
Глава 12. Дополнительные сведения о библиотеке NumPy ...	383
Иерархия типов данных в NumPy	384
Дополнительные манипуляции с массивами	385
Изменение формы массива	385
Упорядочение элементов массива в C и в Fortran	387
Конкатенация и разбиение массива	388
Повторение элементов: функции tile и repeat	390
Эквиваленты прихотливого индексирования: функции take и put.....	391
Укладывание.....	393
Укладывание по другим осям	394
Установка элементов массива с помощью укладывания.....	397
Дополнительные способы использования универсальных функций	398
Методы экземпляра u-функций.....	398
Пользовательские u-функции.....	400
Структурные массивы.....	401
Вложенные типы данных и многомерные поля	402
Зачем нужны структурные массивы?	403

Манипуляции со структурными массивами: <code>numpy.lib.recfunctions</code>	403
Еще о сортировке	403
Косвенная сортировка: методы <code>argsort</code> и <code>lexsort</code>	405
Альтернативные алгоритмы сортировки.....	406
Метод <code>numpy.searchsorted</code> : поиск элементов в отсортированном массиве	407
Класс <code>matrix</code> в NumPy.....	408
Дополнительные сведения о вводе-выводе массивов	410
Файлы, спроектированные на память.....	410
HDF5 и другие варианты хранения массива.....	412
Замечание о производительности	412
Важность непрерывной памяти	412
Другие возможности ускорения: Cython, f2py, C	414
Приложение. Основы языка Python	415
Интерпретатор Python	416
Основы	417
Семантика языка.....	417
Скалярные типы	425
Поток управления.....	431
Структуры данных и последовательности	437
Список	439
Встроенные функции последовательностей.....	443
Словарь	445
Множество	448
Списковое, словарное и множественное включение	450
Функции	452
Пространства имен, области видимости и локальные функции	453
Возврат нескольких значений	454
Функции являются объектами	455
Анонимные (лямбда) функции	456
Замыкания: функции, возвращающие функции.....	457
Расширенный синтаксис вызова с помощью <code>*args</code> и <code>**kwargs</code>	459
Каррирование: частичное фиксирование аргументов.....	459
Генераторы	460
Генераторные выражения.....	462
Модуль <code>itertools</code>	462
Файлы и операционная система	463
Предметный указатель	466

ПРЕДИСЛОВИЕ

За последние 10 лет вокруг языка Python образовалась и активно развивается целая экосистема библиотек с открытым исходным кодом. К началу 2011 года у меня сложилось стойкое ощущение, что нехватка централизованных источников учебных материалов по анализу данных и математической статистике становится камнем преткновения на пути молодых программистов на Python, которым такие приложения нужны по работе. Основные проекты, связанные с анализом данных (в особенности NumPy, IPython, matplotlib и pandas), к тому времени стали уже достаточно зрелыми, чтобы про них можно было написать книгу, которая не устареет сразу после выхода. Поэтому я набрался смелости заняться этим делом. Я был бы очень рад, если бы такая книга существовала в 2007 году, когда я приступал к использованию Python для анализа данных. Надеюсь, вам она окажется полезной, и вы сумеете с успехом воспользоваться описываемыми инструментами в собственной работе.

Графические выделения

В книге применяются следующие графические выделения:

Курсив

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

Моношириинный

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

Моношириинный полужирный

Команды или иной текст, который должен быть введен пользователем буквально.

Моношириинный курсив

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



Так обозначается совет, рекомендация или замечание общего характера.



Так обозначается предупреждение или предостережение.



ГЛАВА 1.

Предварительные сведения

О чём эта книга?

Эта книга посвящена вопросам преобразования, обработки, очистки данных и вычислениям на языке Python. Кроме того, она представляет собой современное практическое введение в научные и инженерные расчеты на Python, ориентированное на приложения для обработки больших объемов данных. Это книга о тех частях языка Python и написанных для него библиотек, которые необходимы для эффективного решения широкого круга задач анализа данных. Но в ней вы *не* найдете объяснений аналитических методов с привлечением Python в качестве языка реализации.

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

- многомерные списки (матрицы);
- табличные данные, когда данные в разных столбцах 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;
- 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.

Этот список далеко не полный. Значительную часть наборов данных можно преобразовать к структурированному виду, более подходящему для анализа и моделирования, хотя сразу не всегда очевидно, как это сделать. В тех случаях, когда это не удается, иногда 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.

Почему именно Python?

Для многих людей (и меня в том числе) Python — язык, в который нельзя не влюбиться. С момента своего появления в 1991 году Python стал одним из самых популярных динамических языков программирования наряду с Perl, Ruby и другими. 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*. Применение Python для этой цели в промышленных и академических кругах значительно расширилось с начала 2000-х годов.

В области анализа 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 другими. Сравнительно недавнее появление улучшенных библиотек для Python (прежде всего, pandas) сделало его серьезным конкурентом в решении задач манипулирования данными. В сочетании с достоинствами Python как универсального языка программирования это делает его отличным выбором для создания приложений обработки данных.

Python как клей

Своим успехом в качестве платформы для 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 других. Поэтому многочисленные компании и национальные лаборатории используют Python как «клей» для объединения написанных за 30 лет программ.

Многие программы содержат небольшие участки кода, на выполнение которых уходит большая часть времени, и большие куски «склеивающего кода», который выполняется нечасто. Во многих случаях время выполнения склеивающего кода несущественно, реальную отдачу дает оптимизация узких мест, которые иногда имеет смысл переписать на низкоуровневом языке типа C.

За последние несколько лет на одно из первых мест в области создания быстрых компилируемых расширений Python и организации интерфейса с кодом на C и C++ вышел проект Cython (<http://cython.org>).

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 стадий исследования и создания прототипа, но и для построения самих производственных систем. Я полагаю, что компании все чаще будут выбирать этот путь, потому что использование одного и того же набора программных средств учеными и технологами несет несомненные выгоды организации.

Недостатки Python

Python – великолепная среда для создания приложений для научных расчетов и большинства систем общего назначения, но тем не менее существуют задачи, для которых 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 обработка больших объектов данных для обеспечения приемлемого времени 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; просто такой код нельзя выполнять в одном процессе Python. Например, в проекте Cython реализована простая интеграция с OpenMP, написанной на С библиотеке параллельных вычислений, позволяющая распараллеливать циклы и тем самым значительно ускорять работу численных алгоритмов.

Необходимые библиотеки для Python

Для читателей, плохо знакомых с экосистемой Python и используемыми в книге библиотеками, я приведу 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 библиотек. 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;
- средства для интеграции с кодом, написанным на C, C++ или Fortran.

Помимо быстрых средств 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.

pandas

Библиотека pandas предоставляет структуры данных и функции, призванные сделать работу со структуризованными данными простым, быстрым и выразительным делом. Как вы вскоре убедитесь, это один из основных компонентов, превращающих Python в мощный инструмент продуктивного анализа данных. Основной объект pandas, используемый в этой книге, – DataFrame – двумерная таблица, в которой строки и столбцы имеют метки:

```
>>> frame
      total_bill    tip     sex   smoker  day    time     size
1       16.99    1.01  Female    No   Sun  Dinner      2
2       10.34    1.66   Male     No   Sun  Dinner      3
3       21.01    3.50   Male     No   Sun  Dinner      3
4       23.68    3.31   Male     No   Sun  Dinner      2
5       24.59    3.61  Female    No   Sun  Dinner      4
6       25.29    4.71   Male     No   Sun  Dinner      4
7        8.77    2.00   Male     No   Sun  Dinner      2
8       26.88    3.12   Male     No   Sun  Dinner      4
9       15.04    1.96   Male     No   Sun  Dinner      2
10      14.78    3.23   Male     No   Sun  Dinner      2
```

Библиотека pandas сочетает высокую производительность средств работы с массивами, присущую NumPy, с гибкими возможностями манипулирования дан-

ными, свойственными 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. В этой книге библиотека pandas будет нашим основным инструментом.

Для разработки финансовых приложений pandas предлагает богатый набор высокопроизводительных средств анализа временных рядов, специально ориентированных на финансовые данные. На самом деле, я изначально проектировал pandas как удобный инструмент анализа именно финансовых данных.

Пользователям языка статистических расчетов R название DataFrame покажется знакомым, потому что оно выбрано по аналогии с объектом `data.frame` в R. Однако они не идентичны: функциональность `data.frame` является собственным подмножеством той, что предлагает `DataFrame`. Хотя эта книга посвящена Python, я время от времени буду проводить сравнения с R, потому что это одна из самых распространенных сред анализа данных с открытым исходным кодом, знакомая многим читателям.

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, JDH), а теперь сопровождается большой группой разработчиков. Она отлично подходит для создания графиков, пригодных для публикации. Интегрирована с IPython (см. ниже), что позволяет организовать удобное интерактивное окружение для визуализации и исследования данных. Графики *интерактивны* – можно увеличить масштаб какого-то участка графика и выполнять панорамирование с помощью панели инструментов в окне графика.

IPython

IPython – компонент стандартного набора инструментов научных расчетов на Python, который связывает все воедино. Он обеспечивает надежную высокопродуктивную среду для интерактивных и исследовательских расчетов. Это оболочка Python с дополнительными возможностями, имеющая целью ускорить написание, тестирование и отладку кода на Python. Особенно она полезна для работы с данными и их визуализации с помощью `matplotlib`. Я почти всегда использую IPython в собственной работе для прогона, отладки и тестирования кода.

Помимо стандартных средств консольной оболочки, IPython предоставляет:

- HTML-блокнот в духе программы Mathematica для подключения к IPython с помощью веб-браузера (подробнее об этом ниже);

- консоль с графическим интерфейсом пользователя на базе библиотеки Qt, включающую средства построения графиков, многострочный редактор и подсветку синтаксиса;
- инфраструктуру для интерактивных параллельных и распределенных вычислений.

Я посвятил отдельную главу оболочке IPython и способам оптимальной работы с ней. Настоятельно рекомендую использовать ее при чтении этой книги.

SciPy

SciPy – собрание пакетов, предназначенных для решения различных стандартных вычислительных задач. Вот несколько 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;
- `scipy.weave`: средство для встраивания кода на C++ с целью ускорения вычислений с массивами.

Совместно NumPy и SciPy образуют достаточно полную замену значительной части системы MATLAB и многочисленных дополнений к ней.

Установка и настройка

Поскольку Python используется в самых разных приложениях, не существует единственной 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 приведу подробные инструкции для разных операционных систем. Я рекомендую использовать следующие базовые дистрибутивы Python:

Enthought Python Distribution: ориентированный на научные применения дистрибутив от компании Enthought (<http://www.enthought.com>). Включает EPDFree – бесплатный дистрибутив (содержит NumPy, SciPy, matplotlib, Chaco и IPython), и

EPD Full – полный комплект, содержащий более 100 научных пакетов для разных предметных областей. EPD Full бесплатно поставляется для академического использования, а прочим пользователям предлагается платная годовая подписка.

Python(x,y) (<http://pythonxy.googlecode.com>): бесплатный ориентированный на научные применения дистрибутив для Windows.

В инструкциях ниже подразумевается EPDFree, но вы можете выбрать любой другой подход, все зависит от ваших потребностей. На момент написания этой книги EPD включает версию Python 2.7, хотя в будущем это может измениться. После установки на вашей машине появятся следующие готовые к импорту пакеты:

- базовые пакеты для научных расчетов: NumPy, SciPy, matplotlib и IPython (входят в EPDFree);
- зависимости для IPython Notebook: tornado и pyzmq (также входят в EPDFree);
- pandas (версии 0.8.2 или выше).

По ходу чтения книги вам могут понадобиться также следующие пакеты: statsmodels, PyTables, PyQt (или эквивалентный ему PySide), xlrd, lxml, basemap, pymongo и requests. Они используются в разных примерах. Устанавливать их не обязательно, и я рекомендую не торопиться с этим до момента, когда они действительно понадобятся. Например, сборка PyQt или PyTables из исходных кодов в OS X или Linux – довольно муторное дело. А пока нам важно получить минимальную работоспособную среду: EPDFree и pandas.

Сведения обо всех Python-пакетах, ссылки на установщики двоичных версий и другую справочную информацию можно найти в указателе пакетов Python (Python Package Index – PyPI, <http://pypi.python.org>). Заодно это отличный источник для поиска новых Python-пакетов.



Во избежание путаницы я не стану обсуждать более сложные средства управления окружением такие, как pip и virtualenv. В Интернете можно найти немало отличных руководств по ним.



Некоторым пользователям могут быть интересны альтернативные реализации Python, например IronPython, Jython или PyPy. Но для работы с инструментами, представленными в этой книге, в настоящее время необходим стандартный написанный на С интерпретатор Python, известный под названием CPython.

Windows

Для начала установки в Windows скачайте установщик EPDFree с сайта <http://www.enthought.com>; это файл с именем epd_free-7.3-1-winx86.msi. Запустите его и согласитесь с предлагаемым по умолчанию установочным каталогом C:\Python27. Если в этом каталоге уже был установлен Python, то рекомендую

предварительно удалить его вручную (или с помощью средства «Установка и удаление программ»).

Затем нужно убедиться, что Python прописался в системной переменной PATH и что не возникло конфликтов с ранее установленными версиями Python. Откройте консоль (выберите из меню «Пуск» пункт «Выполнить...» и наберите cmd.exe). Попробуйте запустить интерпретатор Python, введя команду python. Должно появиться сообщение, в котором указана установленная версия EPDFree:

```
C:\Users\Wes>python
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 14:30:37) on win32
Type "credits", "demo" or "enthought" for more information.
>>>
```

Если в сообщении указана другая версия EPD или вообще ничего не запускается, то нужно привести в порядок переменные среды Windows. В Windows 7 начните вводить фразу «environment variables» в поле поиска программ и выберите раздел Edit environment variables for your account. В Windows XP нужно перейти в Панель управления > Система > Дополнительно > Переменные среды. В появляющемся окне найдите переменную Path. В ней должны присутствовать следующие два каталога, разделенные точкой с запятой:

```
C:\Python27;C:\Python27\Scripts
```

Если вы ранее устанавливали другие версии Python, удалите относящиеся к Python каталоги из системы и из переменных Path. После манипуляций с путями консоль необходимо перезапустить, чтобы изменения вступили в силу.

После того как Python удалось успешно запустить из консоли, необходимо установить pandas. Проще всего для этой цели загрузить подходящий двоичный установщик с сайта <http://pypi.python.org/pypi/pandas>. Для EPDFree это будет файл pandas-0.9.0.win32-py2.7.exe. После того как установщик отработает, запустим IPython и проверим, что все установилось правильно, для этого импортируем pandas и построим простой график с помощью matplotlib:

```
C:\Users\Wes>ipython --pylab
Python 2.7.3 |EPD_free 7.3-1 (32-bit)|
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- 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.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

Если все нормально, то не будет никаких сообщений об ошибках и появится окно с графиком. Можно также проверить, что HTML-блокнот IPython HTML работает правильно:

```
$ ipython notebook --pylab=inline
```



Если в Windows вы обычно используете Internet Explorer, то для работы блокнота IPython, скорее всего, придется установить Mozilla Firefox или Google Chrome.

Дистрибутив EPDFree для Windows содержит только 32-разрядные исполняемые файлы. Если вам необходим 64-разрядный дистрибутив, то проще всего взять EPD Full. Если же вы предпочитаете устанавливать все с нуля и не платить EPD за подписку, то Кристоф Гольке (Christoph Gohlke) из Калифорнийского университета в Ирвайне опубликовал неофициальные двоичные 32- и 64-разрядные установщики для всех используемых в книге пакетов (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>).

Apple OS X

Перед тем как приступать к установке в OS X, необходимо установить Xcode – комплект средств разработки ПО от Apple. Нам понадобится из него gcc – комплект компиляторов для C и C++. Установщик Xcode можно найти на установочном DVD, поставляемом вместе с компьютером, или скачать непосредственно с сайта Apple.

После установки Xcode запустите терминал (Terminal.app), перейдя в меню Applications > Utilities. Введите gcc и нажмите клавишу Enter. Должно появиться такое сообщение:

```
$ gcc  
i686-apple-darwin10-gcc-4.2.1: no input files
```

Теперь необходимо установить EPDFree. Скачайте установщик, который должен представлять собой образ диска с именем вида epd_free-7.3-1-macosx-i386.dmg. Дважды щелкните мышью по файлу dmg-файлу, чтобы смонтировать его, а затем дважды щелкните по mpkg-файлу, чтобы запустить установщик.

Установщик автоматически добавит путь к исполняемому файлу EPDFree в ваш файл .bash_profile, его полный путь /Users/ваше_имя/.bash_profile:

```
# Setting PATH for EPD_free-7.3-1  
PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"  
export PATH
```

Если на последующих шагах возникнут проблемы, проверьте файл .bash_profile – быть может, придется добавить указанный каталог в переменную PATH вручную.

Пришло время установить pandas. Выполните в терминале такую команду:

```
$ sudo easy_install pandas
Searching for pandas
Reading http://pypi.python.org/simple/pandas/
Reading http://pandas.pydata.org
Reading http://pandas.sourceforge.net
Best match: pandas 0.9.0
Downloading http://pypi.python.org/packages/source/p/pandas/pandas-0.9.0.zip
Processing pandas-0.9.0.zip
Writing /tmp/easy_install-H5mIX6/pandas-0.9.0/setup.cfg
Running pandas-0.9.0/setup.py -q bdist_egg --dist-dir /tmp/easy_install-H5mIX6/
pandas-0.9.0/egg-dist-tmp-RhLG0z

Adding pandas 0.9.0 to easy-install.pth file
Installed /Library/Frameworks/Python.framework/Versions/7.3/lib/python2.7/
site-packages/pandas-0.9.0-py2.7-macosx-10.5-i386.egg
Processing dependencies for pandas
Finished processing dependencies for pandas
```

Чтобы убедиться в работоспособности, запустите IPython в режиме Pylab и проверьте импорт pandas и интерактивное построение графика:

```
$ ipython --pylab
22:29 ~/VirtualBox VMs/WindowsXP $ ipython
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 11:28:34)
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- 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.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.
```

In [1]: import pandas

In [2]: plot(arange(10))

Если все нормально, появится окно графика, содержащее прямую линию.

GNU/Linux



В некоторые, но не во все дистрибутивы Linux включены достаточно актуальные версии всех необходимых Python-пакетов, и их можно установить с помощью встроенного средства управления пакетами, например apt. Я продемонстрирую установку на примере EPDFree, потому что она типична для разных дистрибутивов.

Детали варьируются в зависимости от дистрибутива Linux, я буду ориентироваться на дистрибутив Debian, на базе которого построены такие системы, как

Ubuntu и Mint. Установка в основных чертах производится так же, как для OS X, отличается только порядок установки EPDFree. 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 `epd_free-7.3-1-rh5-x86_64.sh`. Для установки нужно выполнить такую команду:

```
$ bash epd_free-7.3-1-rh5-x86_64.sh
```

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

После того как установщик отработает, добавьте в свою переменную `$PATH` подкаталог `bin` EPDFree. Если вы работаете с оболочкой `bash` (в Ubuntu она подразумевается по умолчанию), то нужно будет добавить такую строку в свой файл `.bashrc`:

```
export PATH=/home/wesm/epd/bin:$PATH
```

Естественно, вместо `/home/wesm/epd/` следует подставить свой установочный каталог. Затем запустите новый процесс терминала или повторно выполните файл `.bashrc` командой `source ~/.bashrc`.

Далее понадобится компилятор C, например `gcc`; во многие дистрибутивы Linux `gcc` включен, но это необязательно. В системах на базе Debian установка `gcc` производится командой:

```
sudo apt-get install gcc
```

Если набрать в командной строке слово `gcc`, то должно быть напечатано сообщение:

```
$ gcc
gcc: no input files
```

Теперь можно устанавливать `pandas`:

```
$ easy_install pandas
```

Если вы устанавливали EPDFree от имени пользователя `root`, то, возможно, придется добавить в начало команды слово `sudo` и ввести пароль. Проверка работоспособности производится так же, как в случае OS X.

Python 2 и Python 3

Сообщество Python в настоящее время переживает затянувшийся переход от семейства версий Python 2 к семейству Python 3. До появления Python 3.0 весь код на Python был обратно совместимым. Сообщество решило, что ради прогресса

языка необходимо внести некоторые изменения, которые лишат код этого свойства.

При написании этой книги я взял за основу Python 2.7, потому что большая часть научного сообщества Python еще не перешла на Python 3. Впрочем, если не считать немногих исключений, у вас не возникнет трудностей с исполнением приведенного в книге кода, даже если работать с Python 3.2.

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. Полезно также иметь возможность интерактивно экспериментировать с данными и визуально проверять, что в результате определенных манипуляций получается ожидаемый результат. Библиотеки pandas и NumPy спроектированы с учетом простоты использования в оболочке.

Но кто-то по-прежнему предпочитает работать в IDE, а не в текстовом редакторе. Интегрированные среды действительно предлагают много полезных «фенечек» типа автоматического завершения или быстрого доступа к документации по функциям и классам. Вот некоторые доступные варианты:

- Eclipse с подключаемым модулем PyDev;
- Python Tools для Visual Studio (для работающих в Windows);
- PyCharm;
- Spyder;
- Komodo IDE.

Сообщество и конференции

Помимо поиска в Интернете, существуют полезные списки рассылки, посвященные использованию Python в научных расчетах. 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;
- rpystatsmodels: вопросы, касающиеся statsmodels и pandas;
- numpy-discussion: вопросы, касающиеся NumPy;
- 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 проводят конференции для программистов на Python. PyCon и EuroPython – две самых крупных, проходящие соответственно в США и в Европе. SciPy и EuroSciPy – конференции, ориентированные на научные применения Python, где вы найдете немало «собратьев», если, прочитав эту книгу, захотите более плотно заняться анализом данных с помощью Python.

Структура книги

Если вы раньше никогда не программировали на Python, то имеет смысл начать с *конца* книги, где я поместил очень краткое руководство по синтаксису Python, языкковым средствам и встроенным структурам данных: кортежам, спискам и словарям. Эти знания необходимы для чтения книги.

Книга начинается знакомством со средой IPython. Затем следует краткое введение в основные возможности NumPy, а более продвинутые рассматриваются в другой главе ближе к концу книги. Далее я знакомлю читателей с библиотекой pandas, а все остальные главы посвящены анализу данных с помощью pandas, NumPy и matplotlib (для визуализации). Я старался располагать материал последовательно, но иногда главы все же немного перекрываются.

Файлы с данными и материалы, относящиеся к каждой главе, размещаются в репозитории git на сайте GitHub:

```
http://github.com/pydata/pydata-book
```

Призываю вас скачать данные и пользоваться ими при воспроизведении примеров кода и экспериментах с описанными в книге инструментами. Я буду благодарен за любые добавления, скрипты, блокноты IPython и прочие материалы, которые вы захотите поместить в репозиторий книги для всеобщей пользы.

Примеры кода

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

```
In [5]: код  
Out [5]: результат
```

Иногда для большей ясности несколько примеров кода показаны рядом. Их следует читать слева направо и исполнять по отдельности.

In [5]: код	In [6]: код2
Out [5]: результат	Out [6]: результат2

Данные для примеров

Наборы данных для примеров из каждой главы находятся в репозитории на сайте GitHub: <http://github.com/pydata/pydata-book>. Вы можете получить их либо с помощью командной утилиты системы управления версиями git, либо скачав zip-файл репозитория с сайта.

Я стремился, 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 wesmckinn@gmail.com.

Соглашения об импорте

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

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

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.

- *Синтаксическая глазурь (syntactic sugar)*

Синтаксическая конструкция, которая не добавляет какую-то новую функциональность, а лишь вносит дополнительное удобство или позволяет сделать код короче.

Благодарности

Я не смог бы написать эту книгу без помощи со стороны многих людей.

Из сотрудников издательства O'Reilly я крайне признателен своим редакторам Меган Бланшетт (Meghan Blanchette) и Джуллии Стил (Julie Steele), которые направляли меня на протяжении всей работы. Майк Лукидес (Mike Loukides) работал со мной на этапе подготовки предложения и помог превратить замысел в реальность.

Техническое рецензирование осуществляла большая группа людей. В частности, Мартин Блез (Martin Blais) и Хью Уайт (Hugh White) оказали неоценимую помощь в подборе примеров, повышении ясности изложения и улучшении структуры книги в целом. Джеймс Лонг (James Long), Дрю Конвей (Drew Conway), Фернандо Перес (Fernando Pérez), Брайан Грейндженер (Brian Granger), Томас Клюйвер (Thomas Kluyver), Адам Клейн (Adam Klein), Джош Клейн (Josh Klein), Чань Ши (Chang She) и Стефан ван дер Валт (Stéfan van der Walt) просмотрели по одной или по нескольким глав под разными углами зрения.

Много идей по поводу примеров и наборов данных мне предложили друзья и коллеги по сообществу анализа данных, в том числе: Майк Дьюар (Mike Dewar),

Джефф Хаммербахер (Jeff Hammerbacher), Джеймс Джондроу (James Johndrow), Кристиан Лам (Kristian Lum), Адам Клейн, Хилари Мейсон (Hilary Mason), Чань Ши и Эшли Уильямс (Ashley Williams).

Разумеется, я в долгу перед многими лидерами сообщества «научного Python», которые создали открытый исходный код, легший в основу моих исследований, и оказывали мне поддержку на протяжении всей работы над книгой: группа разработки ядра IPython (Фернандо Перес, Брайан Грейнджен, Мин Рэган-Келли (Min Ragan-Kelly), Томас Клюйвер и другие), Джон Хантер (John Hunter), Скиппер Сиболд (Skipper Seabold), Трэвис Олифант (Travis Oliphant), Питер Уонг (Peter Wang), Эрик Джоунз (Eric Jones), Роберт Керн (Robert Kern), Джозеф Перктольд (Josef Perktold), Франческ Олтед (Francesc Altad), Крис Фоннесбек (Chris Fonnesbeck) и многие, многие другие, перечислять которых здесь нет никакой возможности. Меня также поддерживали, подбадривали и делились идеями Дрю Конвей, Шон Тэйлор (Sean Taylor), Джузеппе Палеолого (Giuseppe Paleologo), Джаред Ландер (Jared Lander), Дэвид Эпштейн (David Epstein), Джо Кровас (John Krowas), Джошуа Блум (Joshua Bloom), Дэн Пилсфорт (Den Pilsworth), Джон Майлз-Уайт (John Myles-White) и многие другие, имена которых я сейчас не могу вспомнить.

Я также благодарен многим, кто оказал влияние на мое становление как ученика. 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 Sargent), Октай Курбанов (Oktay Kurbanov), Мэттью Щантц (Matthew Tschantz), Рони Израэлов (Roni Israelov), Майкл Кац (Michael Katz), Крис Уга (Chris Uga), Прасад Раманан (Prasad Ramanan), Тед 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, ту, которая терпела перепады моего настроения, когда я пытался собрать окончательный вариант рукописи в дополнение к своему и так уже перегруженному графику. А также моим родителям, Биллу и Ким, которые учили меня никогда не отступать от мечты и не соглашаться на меньшее.



ГЛАВА 2.

Первые примеры

В этой книге рассказывается об инструментах, позволяющих продуктивно работать с данными в программах на языке Python. Хотя конкретные цели читателей могут быть различны, почти любую задачу можно отнести к одной из нескольких широких групп:

- *Взаимодействие с внешним миром*
Чтение и запись данных, хранящихся в файлах различных форматов и в базах данных.
- *Подготовка*
Очистка, переформатирование, комбинирование, нормализация, изменение формы, формирование продольных и поперечных срезов, преобразование данных для анализа.
- *Преобразование*
Применение математических и статистических операций к группам наборов данных для порождения новых наборов. Например, агрегирование большой таблицы по групповым переменным.
- *Моделирование и расчет*
Соединение данных со статистическими моделями, алгоритмами машинного обучения и другими вычислительными средствами.
- *Презентация*
Создание интерактивных или статических графических представлений или текстовых сводок.

В этой главе я продемонстрирую несколько наборов данных и что с ними можно делать. Примеры преследуют только одну цель — возбудить у вас интерес, поэтому объяснения будут весьма общими. Не расстраивайтесь, если у вас пока нет опыта работы с описываемыми инструментами; они будут подробно рассматриваться на протяжении всей книги. В примерах кода вы встретите строки вида `In [15] :`, они взяты напрямую из оболочки IPython.

Набор данных 1.usa.gov с сайта bit.ly

В 2011 году служба сокращения URL-адресов bit.ly заключила партнерское соглашение с сайтом правительства США `usa.gov` о синхронном предоставлении

анонимных данных о пользователях, которые сокращают ссылки, заканчивающиеся на .gov или .mil. На момент написания этой книги помимо синхронной ленты, каждый час формируются мгновенные снимки, доступные в виде текстовых файлов¹.

В мгновенном снимке каждая строка представлена в формате JSON (JavaScript Object Notation), широко распространенном в веб. Например, первая строка файла выглядит примерно так:

```
In [15]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'  
In [16]: open(path).readline()  
Out[16]: '{ "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\\1\\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 скачанного мной файла:

```
import json  
path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'  
records = [json.loads(line) for line in open(path)]
```

Для тех, кто никогда не программировал на Python, скажу, что выражение в последней строке называется *списковым включением*; это краткий способ применить некую операцию (в данном случае json.loads) к коллекции строк или других объектов. Очень удобно – в случае, когда итерация применяется к описателю открытого файла, мы получаем последовательность прочитанных из него строк. Получившийся в результате объект records представляет собой список словарей Python:

```
In [18]: records[0]  
Out[18]:  
{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like  
Gecko) Chrome/17.0.963.78 Safari/535.11',  
u'al': u'en-US,en;q=0.8',  
u'c': u'US',  
u'cy': u'Danvers',  
u'g': u'A6qOVH',  
u'gr': u'MA',  
u'h': u>wfLQtf',  
u'hc': 1331822918,  
u'hh': u'1.usa.gov',  
u'l': u'orofrog',  
u'll': [42.576698, -70.954903],
```

¹ <http://www.usa.gov/About/developer-resources/1usagov.shtml>

```
u'nk': 1,
u'r': u'http://www.facebook.com/1/7AQEFzjSi/1.usa.gov/wfLQtF',
u't': 1331923247,
u'tz': u'America/New_York',
u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Отметим, что в Python индексы начинаются с 0, а не с 1, как в некоторых других языках (например, R). Теперь нетрудно выделить интересующие значения из каждой записи, передав строку, содержащую ключ:

```
In [19]: records[0]['tz']
Out[19]: u'America/New_York'
```

Буква `u` перед знаком кавычек означает *unicode* – стандартную кодировку строк. Отметим, что IPython показывает *представление* объекта строки часового пояса, а не его печатный эквивалент:

```
In [20]: print records[0]['tz']
America/New_York
```

Подсчет часовых поясов на чистом Python

Допустим, что 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 [25]: time_zones = [rec['tz'] for rec in records]
-----
KeyError                                                 Traceback (most recent call last)
/home/wesm/book_scripts/whetting/<ipython> in <module>()
----> 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 [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]
In [27]: time_zones[:10]
Out[27]:
[u'America/New_York',
 u'America/Denver',
 u'America/New_York',
 u'America/Sao_Paulo',
 u'America/New_York',
 u'America/New_York',
 u'Europe/Warsaw',
 u'', 
 u'', 
 u'']
```

Мы видим, что уже среди первых 10 часовых поясов встречаются неизвестные (пустые). Их можно было бы тоже отфильтровать, но я пока оставлю. Я 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
```

Зная стандартную библиотеку Python немного лучше, можно было бы записать то же самое короче:

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # values will initialize to 0
    for x in sequence:
        counts[x] += 1
    return counts
```

Чтобы можно было повторно воспользоваться этим кодом, я поместил его в функцию. Чтобы применить его к часовым поясам, достаточно передать этой функции список time_zones:

```
In [31]: counts = get_counts(time_zones)

In [32]: counts['America/New_York']
Out[32]: 1251

In [33]: len(time_zones)
Out[33]: 3440
```

Чтобы получить только первые 10 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 [35]: top_counts(counts)
Out[35]:
[(33, u'America/Sao_Paulo'),
```

```
(35, u'Europe/Madrid'),
(36, u'Pacific/Honolulu'),
(37, u'Asia/Tokyo'),
(74, u'Europe/London'),
(191, u'America/Denver'),
(382, u'America/Los_Angeles'),
(400, u'America/Chicago'),
(521, u''),
(1251, u'America/New_York'])
```

Пошарив в стандартной библиотеке Python, можно найти класс `collections.Counter`, который позволяет решить задачу гораздо проще:

```
In [49]: from collections import Counter

In [50]: counts = Counter(time_zones)

In [51]: counts.most_common(10)
Out[51]:
[(u'America/New_York', 1251),
 (u'', 521),
 (u'America/Chicago', 400),
 (u'America/Los_Angeles', 382),
 (u'America/Denver', 191),
 (u'Europe/London', 74),
 (u'Asia/Tokyo', 37),
 (u'Pacific/Honolulu', 36),
 (u'Europe/Madrid', 35),
 (u'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 [289]: from pandas import DataFrame, Series

In [290]: import pandas as pd

In [291]: frame = DataFrame(records)

In [292]: frame

Out[292]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3560 entries, 0 to 3559
Data columns:
_heartbeat_    120 non-null values
a              3440 non-null values
al             3094 non-null values
c               2919 non-null values
cy              2919 non-null values
g               3440 non-null values
```

```
gr           2919 non-null values
h            3440 non-null values
hc           3440 non-null values
hh           3440 non-null values
kw             93 non-null values
l            3440 non-null values
ll           2919 non-null values
nk           3440 non-null values
r            3440 non-null values
t            3440 non-null values
tz           3440 non-null values
u            3440 non-null values
dtypes: float64(4), object(14)
```

```
In [293]: frame['tz'][:10]
Out[293]:
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
```

На выходе по запросу `frame` мы видим сводное представление, которое показывается для больших объектов DataFrame. Объект Series, возвращаемый в ответ на запрос `frame['tz']`, имеет метод `value_counts`, который дает как раз то, что нам нужно:

```
In [294]: tz_counts = frame['tz'].value_counts()

In [295]: tz_counts[:10]
Out[295]:
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
```

После этого можно с помощью библиотеки `matplotlib` построить график этих данных. Возможно, придется слегка подправить их, подставив какое-нибудь значение вместо неизвестных и отсутствующих часовых поясов. Заменить отсутствующие (NA) значения позволяет функция `fillna`, а неизвестные значения (пустые строки) можно заменить с помощью булевой индексации массива:

```
In [296]: clean_tz = frame['tz'].fillna('Missing')
In [297]: clean_tz[clean_tz == ''] = 'Unknown'
In [298]: tz_counts = clean_tz.value_counts()

In [299]: tz_counts[:10]
Out[299]:
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
```

Для построения горизонтальной столбчатой диаграммы можно применить метод `plot` к объектам `counts`:

```
In [301]: tz_counts[:10].plot(kind='barh', rot=0)
```

Результат показан на рис. 2.1. Ниже мы рассмотрим и другие инструменты для работы с такими данными. Например, поле `a` содержит информацию о браузере, устройстве или приложении, выполнившем сокращение URL:

```
In [302]: frame['a'][1]
Out[302]: u'GoogleMaps/RochesterNY'
```

```
In [303]: frame['a'][50]
Out[303]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'
```

```
In [304]: frame['a'][51]
Out[304]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533'
```

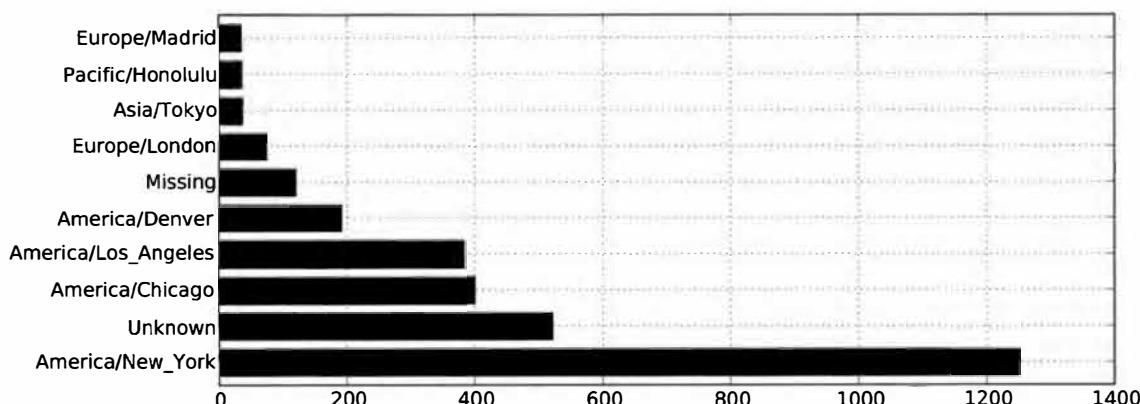


Рис. 2.1. Первые 10 часовых поясов из набора данных 1.usa.gov

Выделение всей интересной информации из таких строк «пользовательских агентов» поначалу может показаться пугающей задачей. По счастью, на деле все не так плохо – нужно только освоить встроенные в Python средства для работы со строками и регулярными выражениями. Например, вот как можно вырезать из строки первую лексему (грубо описывающую возможности браузера) и представить поведение пользователя в другом разрезе:

```
In [305]: results = Series([x.split()[0] for x in frame.a.dropna()])  
  
In [306]: results[:5]  
Out[306]:  
0    Mozilla/5.0  
1    GoogleMaps/RochesterNY  
2    Mozilla/4.0  
3    Mozilla/5.0  
4    Mozilla/5.0  
  
In [307]: results.value_counts()[:8]  
Out[307]:  
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
```

Предположим теперь, что требуется разделить пользователей в первых 10 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 исключаю:

```
In [308]: cframe = frame[frame.a.notnull()]
```

Мы хотим вычислить значение, показывающее, относится строка к пользователю Windows или нет:

```
In [309]: operating_system = np.where(cframe['a'].str.contains('Windows'),  
... : 'Windows', 'Not Windows')  
  
In [310]: operating_system[:5]  
Out[310]:  
0        Windows  
1    Not Windows  
2        Windows  
3    Not Windows  
4        Windows  
Name: a
```

Затем мы можем сгруппировать данные по часовому поясу и только что сформированному столбцу с типом операционной системы:

```
In [311]: 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 [312]: agg_counts = by_tz_os.size().unstack().fillna(0)
```

```
In [313]: agg_counts[:10]
```

```
Out[313]:
```

	Not Windows	Windows
a	245	276
tz		
Africa/Cairo	0	3
Africa/Casablanca	0	1
Africa/Ceuta	0	2
Africa/Johannesburg	0	1
Africa/Lusaka	0	1
America/Anchorage	4	1
America/Argentina/Buenos_Aires	1	0
America/Argentina/Cordoba	0	1
America/Argentina/Mendoza	0	1

Наконец, выберем из полученной таблицы первые 10 часовых поясов. Для этого я построю массив косвенных индексов `agg_counts` по счетчикам строк:

```
# Нужен для сортировки в порядке возрастания
```

```
In [314]: indexer = agg_counts.sum(1).argsort()
```

```
In [315]: indexer[:10]
```

```
Out[315]:
```

tz	
Africa/Cairo	24
Africa/Casablanca	20
Africa/Ceuta	21
Africa/Johannesburg	92
Africa/Lusaka	87
America/Anchorage	53
America/Argentina/Buenos_Aires	54
America/Argentina/Cordoba	57
America/Argentina/Mendoza	26
	55

А затем с помощью `take` расположу строки в порядке, определяемом этим индексом, и оставлю только последние 10:

```
In [316]: count_subset = agg_counts.take(indexer)[-10:]
```

```
In [317]: count_subset
```

```
Out[317]:
```

a	Not Windows	Windows
tz		
America/Sao_Paulo	13	20
Europe/Madrid	16	19
Pacific/Honolulu	0	36
Asia/Tokyo	2	35
Europe/London	43	31
America/Denver	132	59
America/Los_Angeles	130	252
America/Chicago	115	285
	245	276
America/New_York	339	912

Теперь можно построить столбчатую диаграмму, как и в предыдущем примере. Только на этот раз я сделаю ее штабельной, передав параметр `stacked=True` (см. рис. 2.2):

```
In [319]: count_subset.plot(kind='barh', stacked=True)
```

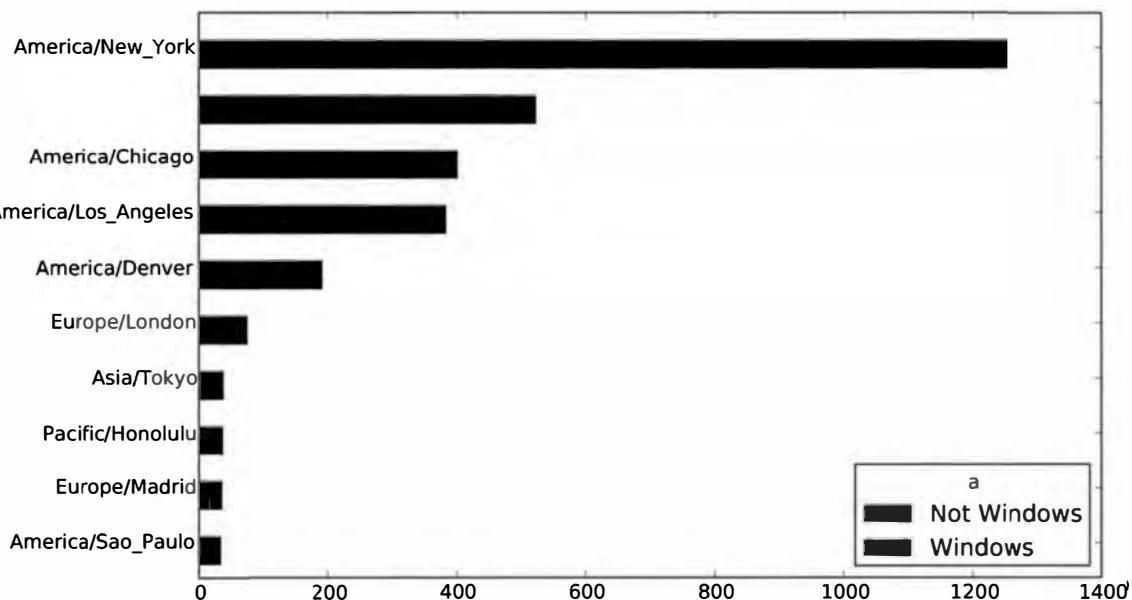


Рис. 2.2. Первые 10 часовых поясов с выделением пользователей Windows и прочих

Из этой диаграммы трудно наглядно представить, какова процентная доля пользователей Windows в каждой группе, но строки легко можно нормировать, так чтобы в сумме получилась 1, а затем построить диаграмму еще раз (рис. 2.3):

```
In [321]: normed_subset = count_subset.div(count_subset.sum(1), axis=0)
In [322]: normed_subset.plot(kind='barh', stacked=True)
```

Все использованные нами методы будут подробно рассмотрены в последующих главах.

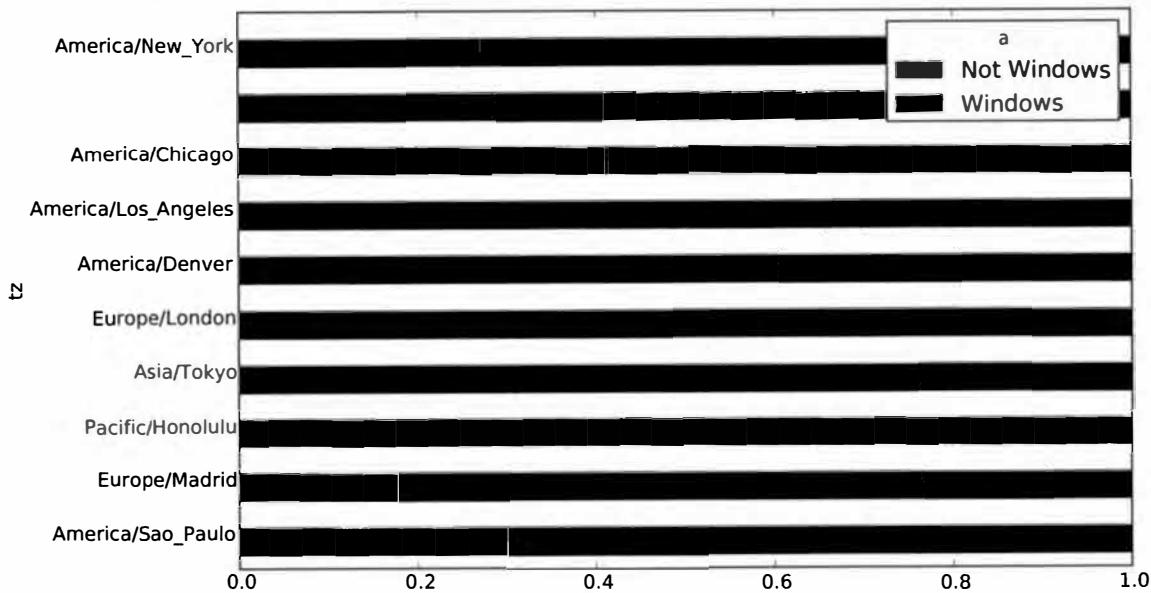


Рис. 2.3. Процентная доля пользователей Windows и прочих в первых 10 часовых поясах

Набор данных MovieLens 1M

Исследовательская группа GroupLens Research (<http://www.grouplens.org/node/73>) предлагает несколько наборов данных о рейтингах фильмов, проставленных пользователями сайта MovieLens в конце 1990-х – начале 2000-х. 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 срезы таких наборов данных с целью привести их к нужному виду.

Набор MovieLens 1M содержит 1 миллион 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

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('ml-1m/users.dat', sep='::', header=None,
                      names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('ml-1m/ratings.dat', sep='::', header=None,
                       names=rnames)

mnames = ['movie_id', 'title', 'genres']
```

```
movies = pd.read_table('ml-1m/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 [334]: users[:5]
```

```
Out[334]:
```

	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 [335]: ratings[:5]
```

```
Out[335]:
```

	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 [336]: movies[:5]
```

```
Out[336]:
```

	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 [337]: ratings
```

```
Out[337]:
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id    1000209    non-null values
movie_id   1000209    non-null values
rating     1000209    non-null values
timestamp  1000209    non-null values
dtypes: int64(4)
```

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 объединить все данные в одну таблицу. Применяя функцию `merge` из библиотеки pandas, мы сначала объединим `ratings` с `users`, а затем результат объединим с `movies`. Pandas определяем, по каким столбцам объединять (или *соединять*), ориентируясь на совпадение имен:

```
In [338]: data = pd.merge(pd.merge(ratings, users), movies)
In [339]: data

Out[339]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id    1000209 non-null values
movie_id   1000209 non-null values
rating     1000209 non-null values
timestamp  1000209 non-null values
gender     1000209 non-null values
age        1000209 non-null values
occupation 1000209 non-null values
zip        1000209 non-null values
title      1000209 non-null values
genres     1000209 non-null values
dtypes: int64(6), object(4)

In [340]: data.ix[0]
Out[340]:
user_id          1
movie_id         1
rating           5
timestamp       978824268
gender           F
age              1
occupation      10
zip              48067
title          Toy Story (1995)
genres          Animation|Children's|Comedy
Name: 0
```

В таком виде агрегирование рейтингов, сгруппированных по одному или нескольким атрибутам пользователя или фильма, для человека, хоть немного знакомого с pandas, не представляет никаких трудностей. Чтобы получить средние рейтинги фильмов при группировке по полу, воспользуемся методом `pivot_table`:

```
In [341]: mean_ratings = data.pivot_table('rating', rows='title',
                                         cols='gender', aggfunc='mean')

In [342]: mean_ratings[:5]
Out[342]:
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 являются общее количество оценок фильма, 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 [343]: ratings_by_title = data.groupby('title').size()

In [344]: ratings_by_title[:10]
Out[344]:
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

In [345]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [346]: active_titles
Out[346]:
Index([''burbs, The (1989), 10 Things I Hate About You (1999),
       101 Dalmatians (1961), ..., Young Sherlock Holmes (1985),
       Zero Effect (1998), eXistenZ (1999)], dtype=object)
```

Затем для отбора строк из приведенного выше объекта `mean_ratings` воспользуемся индексом фильмов, получивших не менее 250 оценок:

```
In [347]: mean_ratings = mean_ratings.ix[active_titles]

In [348]: mean_ratings
Out[348]:
<class 'pandas.core.frame.DataFrame'>
Index: 1216 entries, ''burbs, The (1989) to eXistenZ (1999)
Data columns:
F      1216 non-null values
M      1216 non-null values
dtypes: float64(2)
```

Чтобы найти фильмы, оказавшиеся на первом месте у зрителей-женщин, мы можем отсортировать результат по столбцу `F` в порядке убывания:

```
In [350]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)

In [351]: top_female_ratings[:10]
Out[351]:
gender
Close Shave, A (1995)          F      M
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 [352]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Сортировка по столбцу `'diff'` дает фильмы с наибольшей разностью оценок, которые больше нравятся женщинам:

```
In [353]: sorted_by_diff = mean_ratings.sort_index(by='diff')
```

```
In [354]: sorted_by_diff[:15]
```

```
Out[354]:
```

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
French Kiss (1995)	3.535714	3.056962	-0.478752
Little Shop of Horrors, The (1960)	3.650000	3.179688	-0.470312
Guys and Dolls (1955)	4.051724	3.583333	-0.468391
Mary Poppins (1964)	4.197740	3.730594	-0.467147
Patch Adams (1998)	3.473282	3.008746	-0.464536

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 15 строк
```

```
In [355]: sorted_by_diff[::-1][:15]
```

```
Out[355]:
```

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
Porky's (1981)	2.296875	2.836364	0.539489
Animal House (1978)	3.628906	4.167192	0.538286
Exorcist, The (1973)	3.537634	4.067239	0.529605
Fright Night (1985)	2.973684	3.500000	0.526316
Barb Wire (1996)	1.585366	2.100386	0.515020

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 [356]: rating_std_by_title = data.groupby('title')['rating'].std()

# Оставляем только active_titles
In [357]: rating_std_by_title = rating_std_by_title.ix[active_titles]

# Упорядочиваем Series по значению в порядке убывания
In [358]: rating_std_by_title.order(ascending=False)[:10]
Out[358]:
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
```

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. Ниже я еще вернусь к этому набору данных и покажу, как это сделать.

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.

```
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 уже рассмотренных инструментов большая часть этих задач решается очень просто, и я это кратко продемонстрирую. Призываю вас скачать и исследовать этот набор данных самостоятельно. Если вы обнаружите интересную закономерность, буду рад узнать про нее.

На момент написания этой 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 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 10 строк каждого файла (в Windows можно воспользоваться командой `more` или открыть файл в текстовом редакторе):

```
In [367]: !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 [368]: import pandas as pd  
  
In [369]: names1880 = pd.read_csv('names/yob1880.txt', names=['name', 'sex', 'births'])  
  
In [370]: names1880  
Out[370]:  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 2000 entries, 0 to 1999  
Data columns:  
name      2000 non-null values  
sex       2000 non-null values  
births    2000 non-null values  
dtypes: int64(1), object(2)
```

В эти файлы включены только имена, которыми были названы не менее 5 младенцев в году, поэтому для простоты сумму значений в столбце sex можно считать общим числом родившихся в данном году младенцев:

```
In [371]: names1880.groupby('sex').births.sum()  
Out[371]:  
sex  
F      90993  
M      110493  
Name: births
```

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:

```
# На данный момент 2010 – последний доступный год  
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.

Выглядит объект names следующим образом:

```
In [373]: names
Out[373]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784 non-null values
sex       1690784 non-null values
births    1690784 non-null values
year      1690784 non-null values
dtypes: int64(2), object(2)
```

Имея эти данные, мы уже можем приступить к агрегированию на уровне года и пола, используя метод groupby или pivot_table (см. рис. 2.4):

```
In [374]: total_births = names.pivot_table('births', rows='year',
                                          ....: cols='sex', aggfunc=sum)

In [375]: total_births.tail()
Out[375]:
sex          F          M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382
In [376]: total_births.plot(title='Total births by sex and year')
```

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):
    # При целочисленном делении производится округление с недостатком
    births = group.births.astype(float)

    group['prop'] = births / births.sum()
    return group

names = names.groupby(['year', 'sex']).apply(add_prop)
```



Напомним, что поскольку тип поля births – целое, для вычисления дробного числа необходимо привести числитель или знаменатель к типу с плавающей точкой (если только вы не работаете с Python 3!).

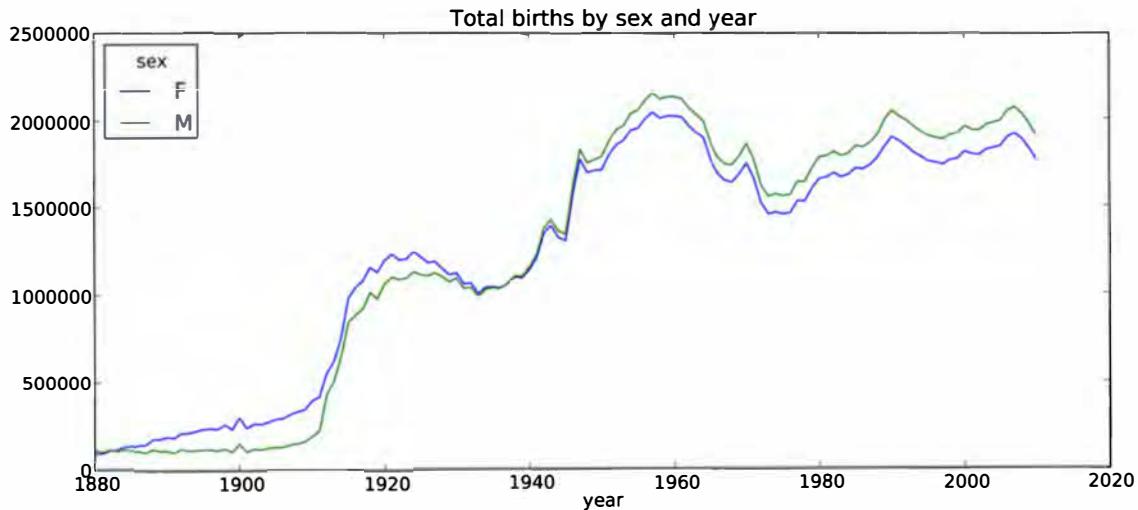


Рис. 2.4. Общее количество родившихся по полу и году

Получившийся в результате пополненный набор данных состоит из таких столбцов:

```
In [378]: names
Out[378]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784 non-null values
sex       1690784 non-null values
births    1690784 non-null values
year      1690784 non-null values
prop      1690784 non-null values
dtypes: float64(1), int64(2), object(2)
```

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. Поскольку это данные с плавающей точкой, воспользуемся методом `np.allclose`, который проверяет, что сумма по группам достаточно близка к 1 (хотя может и не быть равна в точности).

```
In [379]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)
Out[379]: True
```

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)
```

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 [382]: top1000
Out[382]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 261877 entries, 0 to 261876
Data columns:
name      261877 non-null values
sex       261877 non-null values
births    261877 non-null values
year      261877 non-null values
prop      261877 non-null values
dtypes: float64(1), int64(2), object(2)
```

Этот набор, содержащий первые 1000 записей, 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 Топ 1000 на части, относящиеся к мальчикам и девочкам.

```
In [383]: boys = top1000[top1000.sex == 'M']
```

```
In [384]: 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 по именам:

```
In [385]: total_births = top1000.pivot_table('births', rows='year', cols='name',
.....:                 aggfunc=sum)
```

Теперь можно нанести на график несколько имен, воспользовавшись методом `plot` объекта `DataFrame`:

```
In [386]: total_births
Out[386]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6865 entries, Aaden to Zuri
dtypes: float64(6865)
```

```
In [387]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]
```

Имена, которые давали детям в США за период с 1880 по 2010 год

```
In [388]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
.....:             title="Number of births per year")
```

Результат показан на рис. 2.5. Глядя на него, можно сделать вывод, что эти имена в Америке вышли из моды. Но на самом деле картина несколько сложнее, как станет ясно в следующем разделе.

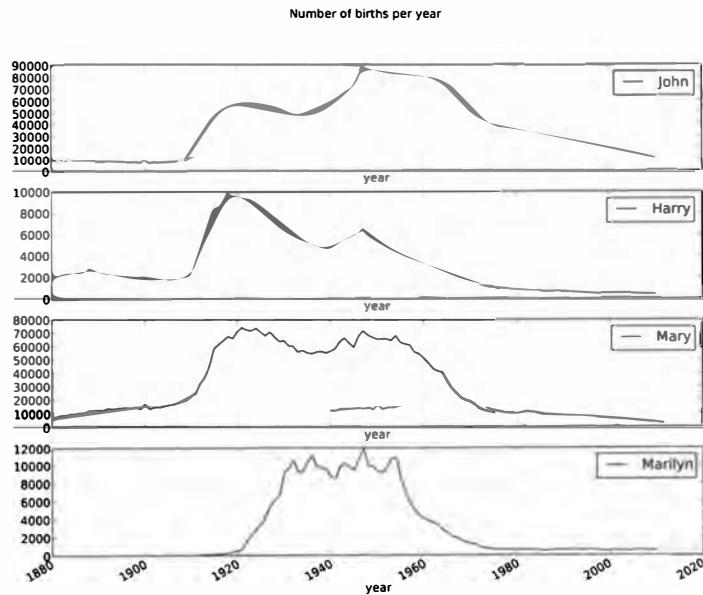


Рис. 2.5. Распределение нескольких имен мальчиков и девочек по годам

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:

```
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))
```

Результат показан на рис. 2.6. 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 [392]: df = boys[boys.year == 2010]

In [393]: df
```

```
Out[393]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 260877 to 261876
Data columns:
name      1000 non-null values
sex       1000 non-null values
births    1000 non-null values
year      1000 non-null values
prop      1000 non-null values
dtypes: float64(1), int64(2), object(2)
```

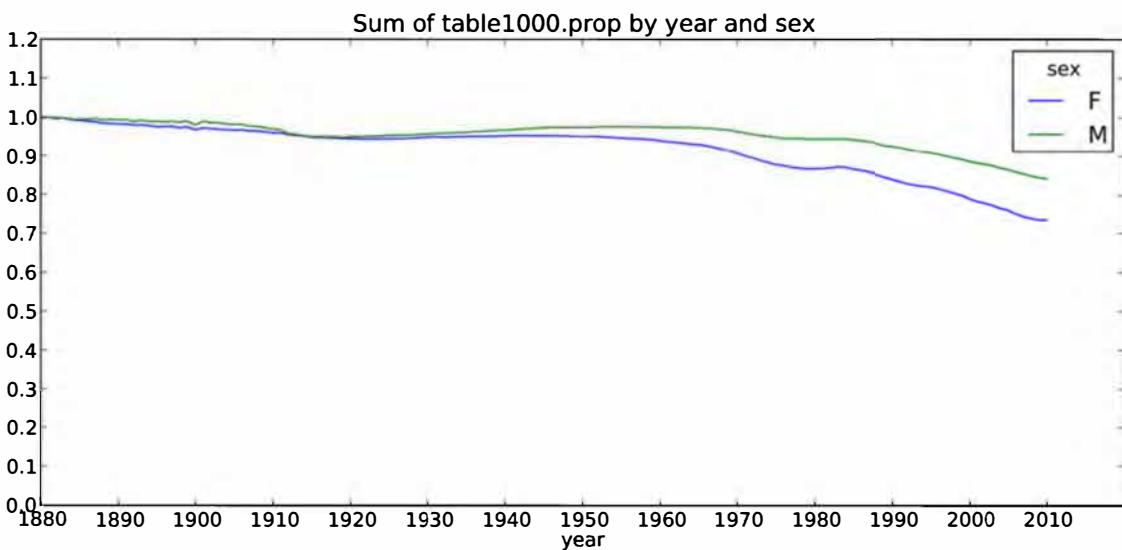


Рис. 2.6. Доля родившихся мальчиков и девочек, представленных в первой тысяче имен

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 [394]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()
```

```
In [395]: prop_cumsum[:10]
```

```
Out[395]:
```

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
```

```
In [396]: prop_cumsum.searchsorted(0.5)
Out[396]: 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 [397]: df = boys[boys.year == 1900]
```

```
In [398]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()
```

```
In [399]: in1900.searchsorted(0.5) + 1
Out[399]: 25
```

Теперь нетрудно применить эту 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 нужный показатель для каждой группы:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_index(by='prop', ascending=False)
    return group.prop.cumsum().searchsorted(q) + 1
```

```
diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

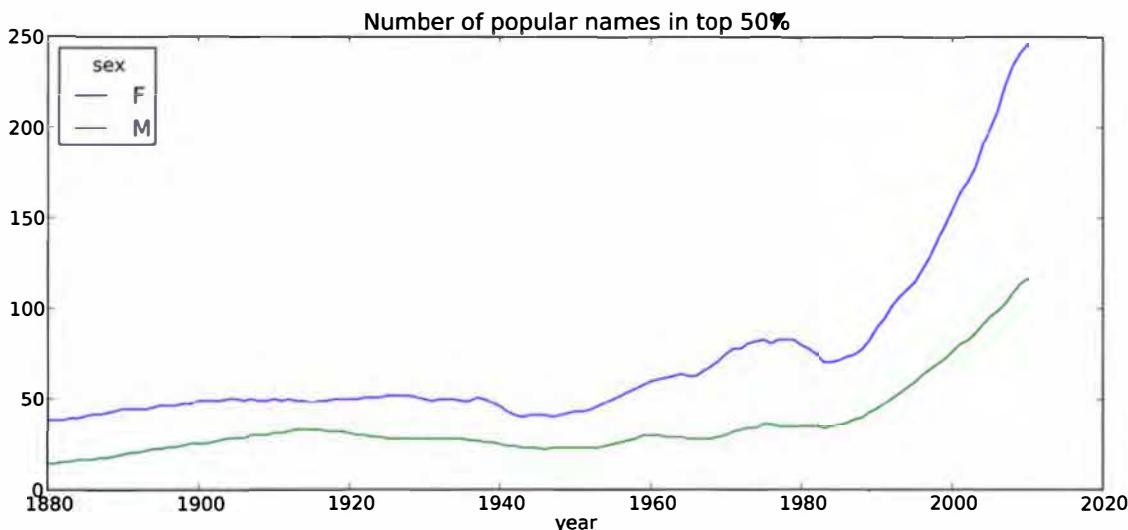


Рис. 2.7. График зависимости разнообразия от года

В получившемся объекте DataFrame с именем `diversity` хранится два временных рядов, по одному для каждого поля, индексированные по году. Его можно исследовать в IPython и, как и раньше, нанести на график (рис. 2.7).

```
In [401]: diversity.head()
Out[401]:
sex      F      M
year
1880    38     14
1881    38     14
1882    38     15
1883    39     15
1884    39     16

In [402]: diversity.plot(title="Number of popular names in top 50%")
```

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 рост числа 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', rows=last_letters,
                           cols=['sex', 'year'], aggfunc=sum)
```

Затем выберу из всего периода три репрезентативных года и напечатаю первые несколько строк:

```
In [404]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [405]: subtable.head()
Out[405]:
          F                               M
sex      1910    1960    2010    1910    1960    2010
year
last_letter
a        108376   691247   670605    977    5204   28438
b         NaN      694      450     411    3912   38859
c          5       49      946     482   15476   23125
d         6750     3729     2607   22111   262112  44398
e        133569   435013   313833   28655   178823  129012
```

Далее я пронормирую эту таблицу на общее число родившихся, чтобы вычислить новую таблицу, содержащую долю от общего родившихся для каждого пола и каждой последней буквы:

```
In [406]: subtable.sum()
```

```
Out[406]:
```

sex	year	count
F	1910	396416
	1960	2022062
	2010	1759010
M	1910	194198
	1960	2132588
	2010	1898382

```
In [407]: letter_prop = subtable / subtable.sum().astype(float)
```

Зная доли букв, я теперь могу нарисовать столбчатые диаграммы для каждого пола, разбив их по годам. Результат показан на рис. 2.8.

```
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)
```

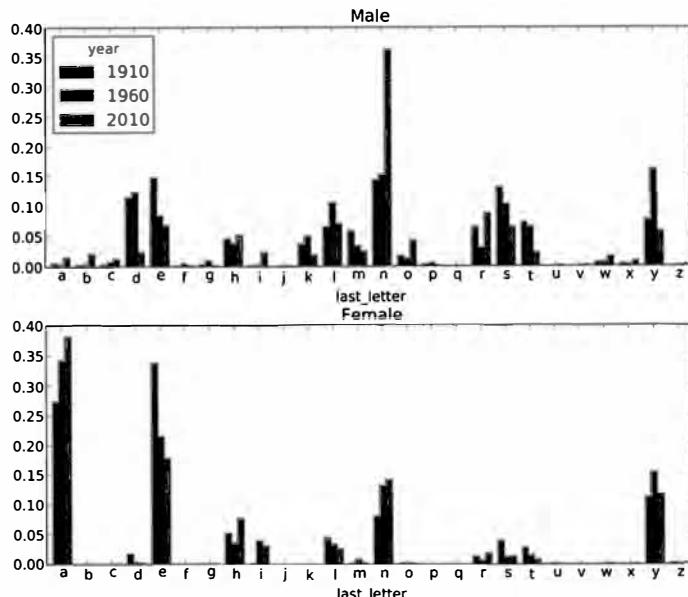


Рис. 2.8. Доли имен мальчиков и девочек, заканчивающихся на каждую букву

Как видим, с 1960-х годов доля имен мальчиков, заканчивающихся буквой «н», 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 [410]: letter_prop = table / table.sum().astype(float)
```

```
In [411]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T
```

```
In [412]: dny_ts.head()
```

Out[412]:

	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

Имея этот объект DataFrame, содержащие временные ряды, я могу все тем же методом `plot` построить график изменения тенденций в зависимости от времени (рис. 2.9):

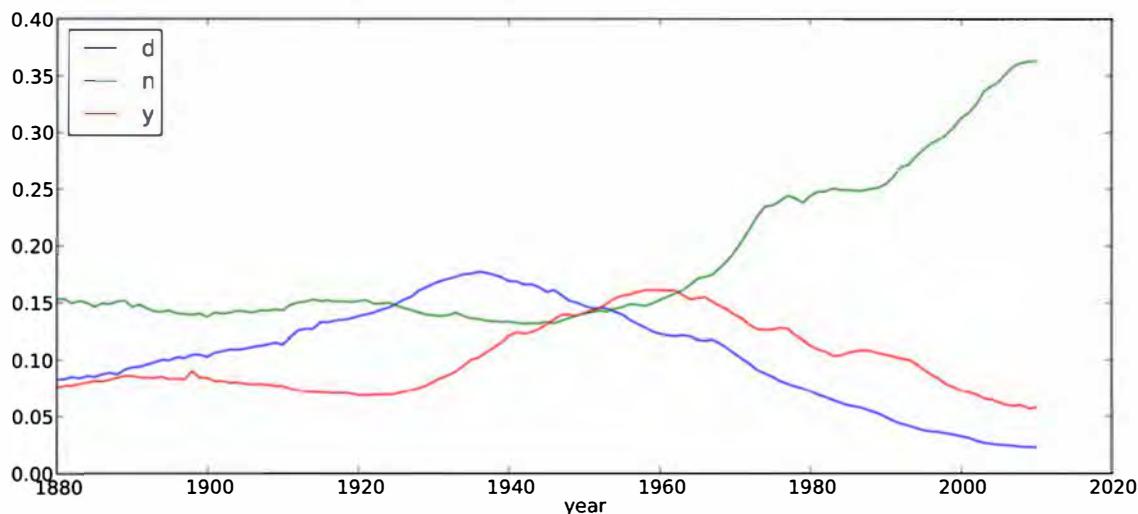
In [414]: `dny_ts.plot()`

Рис. 2.9. Зависимость доли мальчиков с именами, заканчивающимися на буквы d, n, y, от времени

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 [415]: `all_names = top1000.name.unique()`In [416]: `mask = np.array(['lesl' in x.lower() for x in all_names])`In [417]: `lesley_like = all_names[mask]`In [418]: `lesley_like`Out[418]: `array(['Leslie', 'Lesley', 'Leslee', 'Lesli', 'Lesly'], dtype=object)`

Далее можно оставить только эти имена и просуммировать количество родившихся, сгруппировав по имени, чтобы найти относительные частоты:

```
In [419]: filtered = top1000[top1000.name.isin(lesley_like)]
```

```
In [420]: filtered.groupby('name').births.sum()
```

```
Out[420]:
```

	name	births
Leslee	1082	
Lesley	35022	
Lesli	929	
Leslie	370429	
Lesly	10067	
Name: births		

Затем агрегируем по полу и году и нормируем в пределах каждого года:

```
In [421]: table = filtered.pivot_table('births', rows='year',
...                                 cols='sex', aggfunc='sum')
```

```
In [422]: table = table.div(table.sum(1), axis=0)
```

```
In [423]: table.tail()
```

```
Out[423]:
```

year	sex	F	M
2006	1	NaN	
2007	1	NaN	
2008	1	NaN	
2009	1	NaN	
2010	1	NaN	

Наконец, нетрудно построить график распределения по полу в зависимости от времени (рис. 2.10).

```
In [425]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

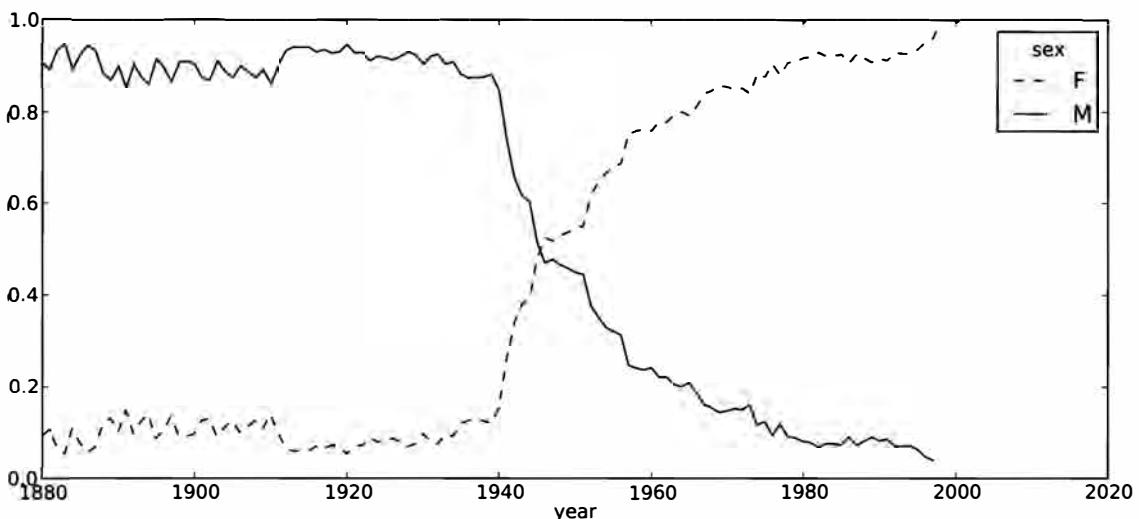


Рис. 2.10. Изменение во времени доли мальчиков и девочек с именами, похожими на Lesley

Выводы и перспективы

Приведенные в этой главе примеры довольно просты, но они позволяют составить представление о том, чего ожидать от последующих глав. Эта книга посвящена, прежде всего, инструментам, а не демонстрации более сложных аналитических методов. Овладев описываемыми приемами, вы сможете без труда реализовать собственные методы анализа (если, конечно, знаете, чего хотите!).



ГЛАВА 3.

IPython: интерактивные вычисления и среда разработки

Действуй, не делая. Работай, не прилагая усилий. Думай о малом, как о большом, и о нескольких, как о многих. Работай над трудной задачей, пока она еще проста; реализуй великую цель через множество небольших шагов.

Лао Цзы

Меня часто спрашивают: «В какой среде разработки на Python вы работаете?». Почти всегда я отвечаю «IPython и текстовый редактор». Вы можете заменить текстовый редактор интегрированной средой разработки (IDE), если хотите иметь графические инструменты и средства автоматического завершения кода. Но и в этом случае я советую не отказываться от IPython. Некоторые IDE даже включают интеграцию с IPython, так что вы получаете лучшее из обоих миров.

Проект *IPython* в 2001 году основал Фернандо Перес как побочный продукт по ходу создания усовершенствованного интерактивного интерпретатора Python. Впоследствии он превратился в один из самых важных инструментов в арсенале ученых, работающих на Python. Сам по себе он не предлагает ни вычислительных, ни аналитических средств, но изначально спроектирован с целью повысить производительность интерактивных вычислений и разработки ПО. В его основе лежит последовательность действий «выполни и посмотри» вместо типичной для многих языков «отредактируй, откомпилируй и запусти». Он также очень тесно интегрирован с оболочкой операционной системы и с файловой системой. Поскольку анализ данных подразумевает исследовательскую работу, применение метода проб и ошибок и итеративный подход, то IPython почти во всех случаях позволяет ускорить выполнение работы.

Разумеется, сегодняшний IPython – это куда больше, чем просто усовершенствованная интерактивная оболочка Python. В его состав входит развитая графическая консоль с встроенными средствами построения графиков, интерактивный веб-блокнот и облегченный движок для быстрых параллельных вычислений. И подобно многим другим инструментам, созданным программистами для программистов, он настраивается в очень широких пределах. Некоторые из вышеупомянутых возможностей будут рассмотрены ниже.

Поскольку интерактивность неотъемлема от IPython, некоторые описываемые далее возможности трудно в полной мере продемонстрировать без «живой» консоли. Если вы читаете об IPython впервые, я рекомендую параллельно прорабатывать примеры, чтобы понять, как все это работает на практике. Как и в любой среде, основанной на активном использовании клавиатуры, для полного освоения инструмента необходимо запомнить комбинации клавиш для наиболее употребительных команд.



При первом чтении многие части этой главы (к примеру, профилирование и отладка) можно пропустить без ущерба для понимания последующего материала. Эта глава задумана как независимый, достаточно полный обзор функциональности IPython.

Основы IPython

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

```
$ ipython
Python 2.7.2 (default, May 27 2012, 21:26:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- 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 [542]: data = {i : randn() for i in range(7)}

In [543]: data
Out[543]:
{0: 0.6900018528091594,
 1: 1.0015434424937888,
 2: -0.5030873913603446,
 3: -0.6222742250596455,
 4: -0.9211686080130108,
 5: -0.726213492660829,
 6: 0.2228955458351768}
```

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.

Завершение по нажатию клавиши Tab

На первый взгляд, оболочка IPython очень похожа на стандартный интерпретатор Python с мелкими косметическими изменениями. Пользователям программы Mathematica знакомы пронумерованные строки ввода и вывода. Одно из существенных преимуществ над стандартной оболочкой Python – завершение по нажатию клавиши **Tab**, функция, реализованная в большинстве интерактивных сред анализа данных. 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
```



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.

Завершение по нажатию **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 [3]: book_scripts/<Tab>
book_scripts/cprof_example.py      book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py        book_scripts/prof_mod.py

In [3]: path = 'book_scripts/<Tab>
book_scripts/cprof_example.py      book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py        book_scripts/prof_mod.py
```

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 и самого знака `=`).

Инроспекция

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

```
In [545]: b?
Type:      list
String Form: [1, 2, 3]
Length:    3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Это называется *инроспекцией объекта*. Если объект представляет собой функцию или метод экземпляра, то будет показана строка документации, если она существует. Допустим, мы написали такую функцию:

```
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 [547]: add_numbers?
Type:      function
String Form: <function add_numbers at 0x5fad848>
File:      book_scripts/<ipython-input-546-5473012eeb65>
Definition: add_numbers(a, b)
Docstring:
Сложить два числа
Возвращает
-----
the_sum : тип аргументов
```

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

```
In [548]: add_numbers??
Type:      function
String Form: <function add_numbers at 0x5fad848>
File:      book_scripts/<ipython-input-546-5473012eeb65>
Definition: add_numbers(a, b)
Source:
def add_numbers(a, b):
"""
Сложить два числа

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

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 [549]: np.*load*?
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 [550]: %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 [551]: c
Out[551]: 7.5
```

```
In [552]: result
Out[552]: 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`.

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 стечения 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 с помощью диспетчера задач ОС.

Исполнение кода из буфера обмена

Когда нужно быстро и без хлопот выполнить код в IPython, можно просто взять его из буфера обмена. На первый взгляд, неряшливо, но на практике очень полезно. Например, при разработке сложного или долго работающего приложения иногда желательно исполнять скрипт по частям, останавливаясь после каждого шага, чтобы проверить загруженные данные и результаты. Или, допустим, вы нашли какой-то фрагмент кода в Интернете и хотите поэкспериментировать с ним, не создавая новый ру-файл.

Для извлечения фрагмента кода из буфера обмена во многих случаях достаточно нажать `<Ctrl-Shift-V>`. Отметим, что это не стопроцентно надежный способ, потому в таком режиме имитируется ввод каждой строки в IPython, а символы новой строки трактуются как нажатие `<Enter>`. Это означает, что если извлекается код, содержащий блок с отступом, и в нем присутствует пустая строка, то IPython будет считать, что блок закончился. При выполнении следующей строки в блоке возникнет исключение `IndentationError`. Например, такой код:

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

y = 8
```

после вставки из буфера обмена работать не будет:

```
In [1]: x = 5
In [2]: y = 7
In [3]: if x > 5:
...:     x += 1
...:
In [4]: y = 8
IndentationError: unexpected indent

If you want to paste code into IPython, try the %paste and %cpaste
magic functions.
```

В сообщении об ошибке предлагается использовать магические функции `%paste` и `%cpaste`. Функция `%paste` принимает текст, находящийся в буфере обмена, и исполняет его в оболочке как единый блок:

```
In [6]: %paste
x = 5
y = 7
if x > 5:
    x += 1

y = 8
## -- Конец вставленного текста --
```



В зависимости от платформы и способа установки Python может случиться, что функция `%paste` откажется работать, хотя это маловероятно. В пакетных дистрибутивах типа EPDFree (описан во введении) такой проблемы быть не должно.

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

```
In [7]: %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>`.

Ниже мы познакомимся с HTML-блокнотом IPython, который выводит поблочный анализ на новый уровень – с помощью работающего в браузере блокнота с исполняемыми ячейками, содержащими код.

Взаимодействие IPython с редакторами и IDE

Для некоторых текстовых редакторов, например Emacs и vim, существуют расширения, позволяющие отправлять блоки кода напрямую из редактора в запущенную оболочку IPython. Для получения дополнительных сведений зайдите на сайт IPython или поищите в Интернете.

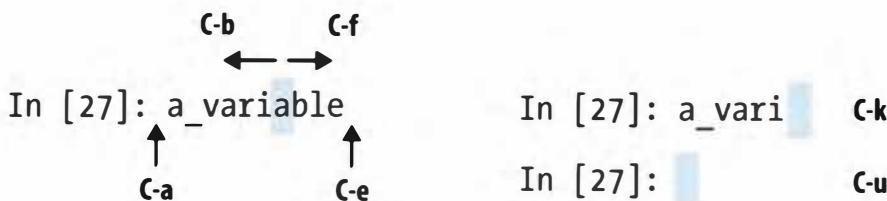
Для некоторых IDE существуют подключаемые модули, например PyDev для Eclipse или Python Tools для Visual Studio от Microsoft (а, возможно, и для других), обеспечивающие интеграцию с консольным приложением IPython. Если вы хотите работать в IDE, не отказываясь от консольных функций IPython, то это может стать удачным решением.

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. 3.1 перечислены наиболее употребительные комбинации, а на рис. 3.1 некоторые из них, например перемещение курсора, проиллюстрированы.

Таблица 3.1. Стандартные комбинации клавиш IPython

Команда	Описание
Ctrl-P или стрелка-вверх	Просматривать историю команд назад в поисках команд, начинающихся с введенной строки
Ctrl-N или стрелка-вниз	Просматривать историю команд вперед в поисках команд, начинающихся с введенной строки
Ctrl-R 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. 3.1.** Иллюстрация некоторых комбинаций клавиш IPython

Исключения и обратная трассировка

Если при выполнении любого предложения или скрипта, запущенного командой `%run`, возникнет исключение, то по умолчанию IPython напечатает все содержимое стека вызовов (обратную трассировку), сопроводив каждый вызов несколькими строками контекста.

```
In [553]: %run ch03/ipython_bug.py
-----
AssertionError                                     Traceback (most recent call last)
/home/wesm/code/ipython/IPython/utils/py3compat.pyc in execfile(fname, *where)
 176 else:
 177     filename = fname
--> 178 __builtin__.execfile(filename, *where)
book_scripts/ch03/ipython_bug.py in <module>()
 13 throws_an_exception()
 14
```

```

--> 15 calling_things()
book_scripts/ch03/ipython_bug.py in calling_things()
    11 def calling_things():
    12     works_fine()
--> 13 throws_an_exception()
    14
    15 calling_things()
book_scripts/ch03/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 само по себе является большим преимуществом по сравнению со стандартным интерпретатором Python (который не выводит контекст). Объемом контекста можно управлять с помощью магической команды `%xmode` – от минимального (как в стандартном интерпретаторе Python) до подробного (когда включаются значения аргументов функции и другая информация). Ниже в этой главе мы увидим, что можно перемещаться по стеку (с помощью магических команд `%debug` и `%pdb`) после ошибки, это дает возможность производить посмертную отладку.

Магические команды

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 [554]: a = np.random.randn(100, 100)

In [555]: %timeit np.dot(a, a)
10000 loops, best of 3: 69.1 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 [1]: %reset?
Возвращает пространство имен в начальное состояние, удаляя все имена,
определенные пользователем.

```

Параметры

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

-f : принудительная очистка без запроса подтверждения.

-s : 'Мягкая' очистка: очищается только ваше пространство имен, а история остается. Ссылки на объекты можно удержать. По умолчанию (без этого параметра) выполняется 'жесткая' очистка, в результате чего вы получаете новый сеанс, а все ссылки на объекты в текущем сеансе удаляются.

Примеры

In [6]: a = 1

In [7]: a

Out[7]: 1

In [8]: 'a' in _ip.user_ns

Out[8]: True

In [9]: %reset -f

In [1]: 'a' in _ip.user_ns

Out[1]: False

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.

Поскольку к документации по IPython легко можно обратиться из системы, я рекомендую изучить все имеющиеся специальные команды, набрав %quickref или %magic. Я расскажу только о нескольких наиболее важных для продуктивной работы в области интерактивных вычислений и разработки в среде IPython.

Таблица 3.2. 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
%cpaste	Открыть специальное приглашение для ручной вставки Python-кода, подлежащего выполнению

Команда	Описание
%reset	Удалить все переменные и прочие имена, определенные в интерактивном пространстве имен
%page <i>OBJECT</i>	Сформировать красиво отформатированное представление объекта и вывести его постранично
%run <i>script.py</i>	Выполнить Python-скрипт из IPython
%prun <i>предложение</i>	Выполнить <i>предложение</i> под управлением cProfile и вывести результаты профилирования
%time <i>предложение</i>	Показать время выполнения одного предложения
%timeit <i>предложение</i>	Выполнить <i>предложение</i> 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 <i>переменная</i>	Удалить переменную и попытаться очистить все ссылки на объект во внутренних структурах данных IPython

Графическая консоль на базе Qt

Команда IPython разработала графическую консоль на базе библиотеки Qt (рис. 3.2), цель которой – скрестить возможность чисто консольного приложения со средствами, предоставляемыми виджетом обогащенного текста, в том числе встраиваемыми изображениями, режимом многострочного редактирования и подсветкой синтаксиса. Если на вашей машине установлен пакет PyQt или PySide, то можно запустить приложение с встроенным построением графиков:

```
ipython qtconsole --pylab=inline
```

Qt-консоль позволяет запускать несколько процессов IPython в отдельных вкладках и тем самым переключаться с одной задачи на другую. Она также может разделять процесс с HTML-блокнотом IPython, о котором я расскажу ниже.

Интеграция с matplotlib и режим pylab

IPython так широко используется в научном сообществе отчасти потому, что он спроектирован как дополнение к библиотекам типа matplotlib и другим графическим инструментам. Если вы раньше никогда не работали с matplotlib, ничего страшного; ниже мы обсудим эту библиотеку во всех подробностях. Создав окно графика matplotlib в стандартной оболочке Python, вы будете неприятно поражены тем, что цикл обработки событий ГИП «перехватывает контроль» над сеансом Python до тех пор, пока окно не будет закрыто. Для интерактивного анализа данных и визуализации это не годится, поэтому в IPython реализована специальная логика для всех распространенных библиотек построения ГИП, так чтобы обеспечить органичную интеграцию с оболочкой.

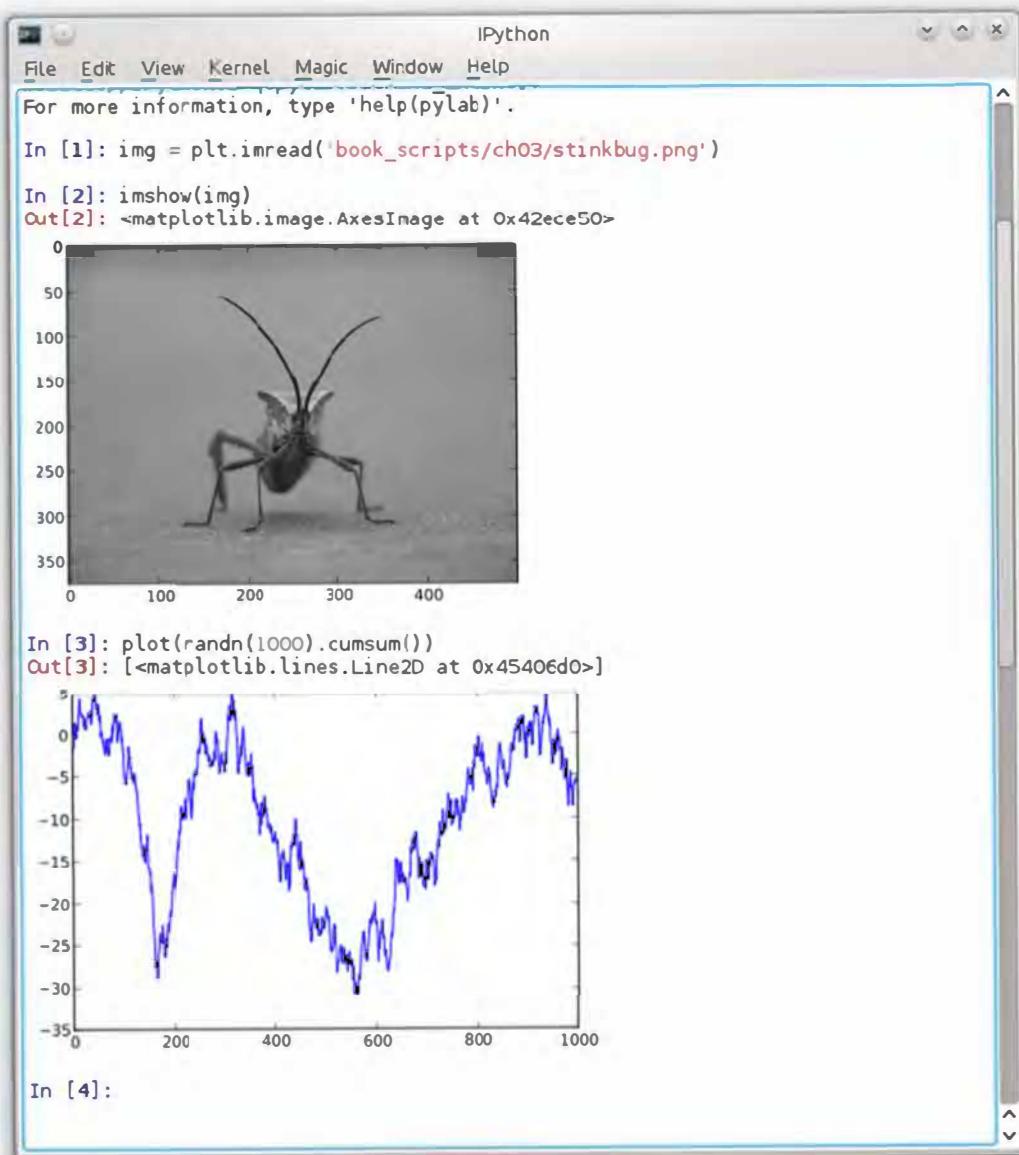


Рис. 3.2. Qt-консоль IPython

Для запуска IPython в режиме интеграции с matplotlib достаточно просто добавить флаг `--pylab` (дефисов должно быть два).

```
$ ipython --pylab
```

При этом произойдет несколько вещей. Во-первых, IPython запустится в режиме интеграции с ГИП по умолчанию, что позволит без проблем создавать окна графиков matplotlib. Во-вторых, в интерактивное пространство имен верхнего уровня будет импортирована большая часть NumPy и matplotlib, в результате чего создается среда интерактивных вычислений, напоминающая MATLAB и другие

предметно-ориентированные среды (рис. 3.3). Такую же конфигурацию можно задать и вручную, воспользовавшись командой %gui (наберите %gui?, чтобы узнать, как).

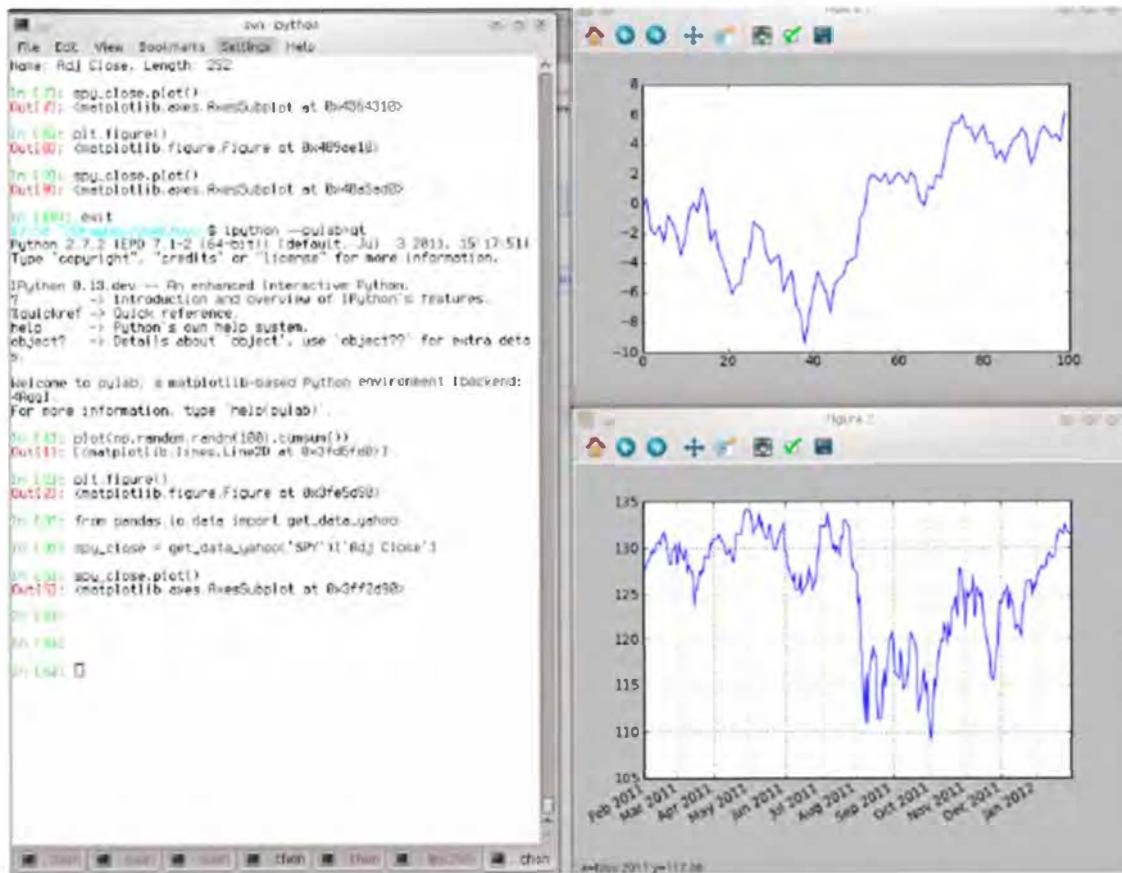


Рис. 3.3. Режим Pylab: IPython с окнами matplotlib

История команд

IPython поддерживает небольшую базу данных на диске, в которой хранятся 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.

Поиск в истории команд и повторное выполнение

Возможность искать и повторно выполнять предыдущие команды для многих является самой полезной функцией. Поскольку IPython рассчитан на итератив-

ную 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 несколько первых букв команды `%run` и нажать `<Ctrl-P>` или клавишу `<стрелка вверх>`. В ответ IPython найдет в истории команд первую из предшествующих команд, начинающуюся введенными буквами. При повторном нажатии `<Ctrl-P>` или `<стрелки вверх>` поиск будет продолжен. Если вы проскочили мимо нужной команды, ничего страшного. По истории команд можно перемещаться и *вперед* с помощью клавиш `<Ctrl-N>` или `<стрелка вниз>`. 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 ей, 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 выходной объект хранятся соответственно в переменных `_` (один подчерк) и `__` (два подчерка):

```
In [556]: 2 ** 27
Out[556]: 134217728
```

```
In [557]: _
Out[557]: 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 новых переменных `_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 физически очищает ввод и вывод. 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.

Протоколирование ввода-вывода

IPython умеет протоколировать весь консольный сеанс, в том числе ввод и вывод. Режим протоколирования включается командой %logstart:

```
In [3]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename      : ipython_log.py
Mode          : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State         : active
```

Включить протоколирование IPython можно в любое время, и записан будет весь сеанс (в том числе и ранее выполненные команды). Таким образом, если в процессе работы вы решите сохранить все сделанное к этому моменту, достаточно будет включить протоколирование. Дополнительные параметры описаны в строке документации по функции %logstart (в частности, как изменить путь к файлу журнала) и связанным с ней функциям %logoff, %logon, %logstate и %logstop.

Взаимодействие с операционной системой

Еще одна важная особенность IPython — очень тесная интеграция с 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. 3.3. В следующих разделах я кратко расскажу о них.

Таблица 3.3. 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

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. Можно даже запустить процесс, который перенимает управление у IPython, даже еще один интерпретатор Python:

```
In [2]: !python
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
```

```
Type "packages", "demo" or "enthought" for more information.  
>>>
```

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:192.168.1.137 Bcast:192.168.1.255 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, я регулярно захожу в каталог Dropbox, поэтому могу определить закладку, которая дает возможность быстро перейти в него:

```
In [6]: %bookmark db /home/wesm/Dropbox/
```

После этого с помощью магической команды `%cd` я смогу воспользоваться ранее определенными закладками:

```
In [7]: cd db  
(bookmark:db) -> /home/wesm/Dropbox/  
/home/wesm/Dropbox
```

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:  
db -> /home/wesm/Dropbox/
```

Закладки, в отличие от псевдонимов, автоматически сохраняются после закрытия сеанса.

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 нажатию 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 ch03/ipython_bug.py  
-----  
AssertionError Traceback (most recent call last)  
/home/wesm/book_scripts/ch03/ipython_bug.py in <module>()
```

```

13      throws_an_exception()
14
---> 15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in calling_things()
    11 def calling_things():
    12     works_fine()
---> 13     throws_an_exception()
    14
    15 calling_things()

/home/wesm/book_scripts/ch03/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/book_scripts/ch03/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/book_scripts/ch03/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 режиме, 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 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1

```

```
NOTE: Enter 'c' at the ipdb> prompt to start your script.  
> <string>(1)<module>()
```

```
ipdb> s  
--Call--  
> /home/wesm/book_scripts/ch03/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/book_scripts/ch03/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 `s` или выполнить `works_fine()` без захода внутрь, т. е. перейти к следующей строке, нажав `n` (`next` – дальше):

```
ipdb> n  
> /home/wesm/book_scripts/ch03/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/book_scripts/ch03/ipython_bug.py(6)throws_an_exception()  
      5  
----> 6 def throws_an_exception():  
      7     a = 5  
  
ipdb> n  
> /home/wesm/book_scripts/ch03/ipython_bug.py(7)throws_an_exception()  
      6 def throws_an_exception():  
----> 7     a = 5  
      8     b = 6  
  
ipdb> n
```

```
> /home/wesm/book_scripts/ch03/ipython_bug.py(8) throws_an_exception()
    7      a = 5
----> 8      b = 6
    9      assert(a + b == 10)

ipdb> n
> /home/wesm/book_scripts/ch03/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. 3.3 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 большинстве IDE для Python имеются отличные графические отладчики, но обычно отладка в самом IPython оказывается намного продуктивнее.

Таблица 3.4. Команда отладчика (!)Python

Команда	Действие
h(elp)	Вывести список команд
help команда	Показать документацию по <i>команде</i>
c(ontinue)	Продолжить выполнение программы
q(uit)	Выйти из отладчика, прекратив выполнение кода
b(reak) <i>номер</i>	Поставить точку остановки на строке с <i>номером</i> в текущем файле
b <i>путь/к/файлу.py:номер</i>	Поставить точку остановки на строке с <i>номером</i> в <i>указанном файле</i>
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 <i>предложение</i>	Выполнить <i>предложение</i> в новом (вложенном) отладчике
l(ist) <i>предложение</i>	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):

```
def set_trace():
    from IPython.core.debugger import Pdb
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    from IPython.core.debugger import Pdb
    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 ch03/ipython_bug.py
> /home/wesm/book_scripts/ch03/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:

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

и хотели бы пройти ее в пошаговом режиме. 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 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/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 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/book_scripts/ch03/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
```

Поскольку 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 сравнительно 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, порядка микросекунд (10^{-6} секунд) или наносекунд (10^{-9} секунд). Вроде бы совсем мизерные промежутки времени, но если функцию, работающую 20 микросекунд, вызвать миллион 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 -r

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 xrange(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: %e' % np.max(some_results)
```

Незнание NumPy пусть вас не пугает. Это 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.

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  prof_mod.py:3(add_and_sum)
      1    0.009    0.009    0.009 {method 'sum' of 'numpy.ndarray' objects}
      1    0.003    0.003    0.049 <string>:1(<module>)
      1    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

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
```

Согласитесь, 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» времени выполнения каждой строки весьма значительны. Трассировка функций, не представляющих интереса, может существенно изменить результаты профилирования.

HTML-блокнот в IPython

В 2011 году команда разработчиков IPython, возглавляемая Брайаном Грейнджером, приступила к разработке основанного на веб-технологиях формата интерактивного вычислительного документа под названием блокнот IPython (IPython Notebook). Со временем он превратился в чудесный инструмент для интерактивных вычислений и идеальное средство для воспроизводимых исследований и преподавания. Я пользовался им при написании большинства примеров для этой книги, призываю и вас не пренебрегать им.

Формат ipynb-документа основан на JSON и позволяет легко обмениваться кодом, результатами и рисунками. На недавних конференциях по Python получил широкое распространение такой подход к демонстрациям: создать ipynb-файлы в блокноте и разослать их всем желающим для экспериментов.

Блокнот реализован в виде облегченного серверного процесса, запускаемого из командной строки, например:

```
$ ipython notebook --pylab=inline  
[NotebookApp] Using existing profiledir: u'/home/wesm/.config/ipython/profile_default'  
[NotebookApp] Serving notebooks from /home/wesm/book_scripts  
[NotebookApp] The IPython Notebook is running at: http://127.0.0.1:8888/  
[NotebookApp] Use Control-C to stop this server and shut down all kernels.
```

На большинстве платформ автоматически откроется браузер, подразумевающий по умолчанию, и в нем появится информационная панель блокнота (рис. 3.4). Иногда приходится переходить на нужный URL-адрес самостоятельно. После этого можно создать новый блокнот и приступить к исследованиям.

Поскольку блокнот работает внутри браузера, серверный процесс можно запустить где угодно. Можно даже организовать безопасное соединение с блокнотами, работающими в облаке, например Amazon EC2. На момент написания этой книги уже существовал новый проект NotebookCloud (<http://notebookcloud.appspot.com>), который позволяет без труда запускать блокноты в EC2.

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 изложенное 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 анало-

гичную, 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.

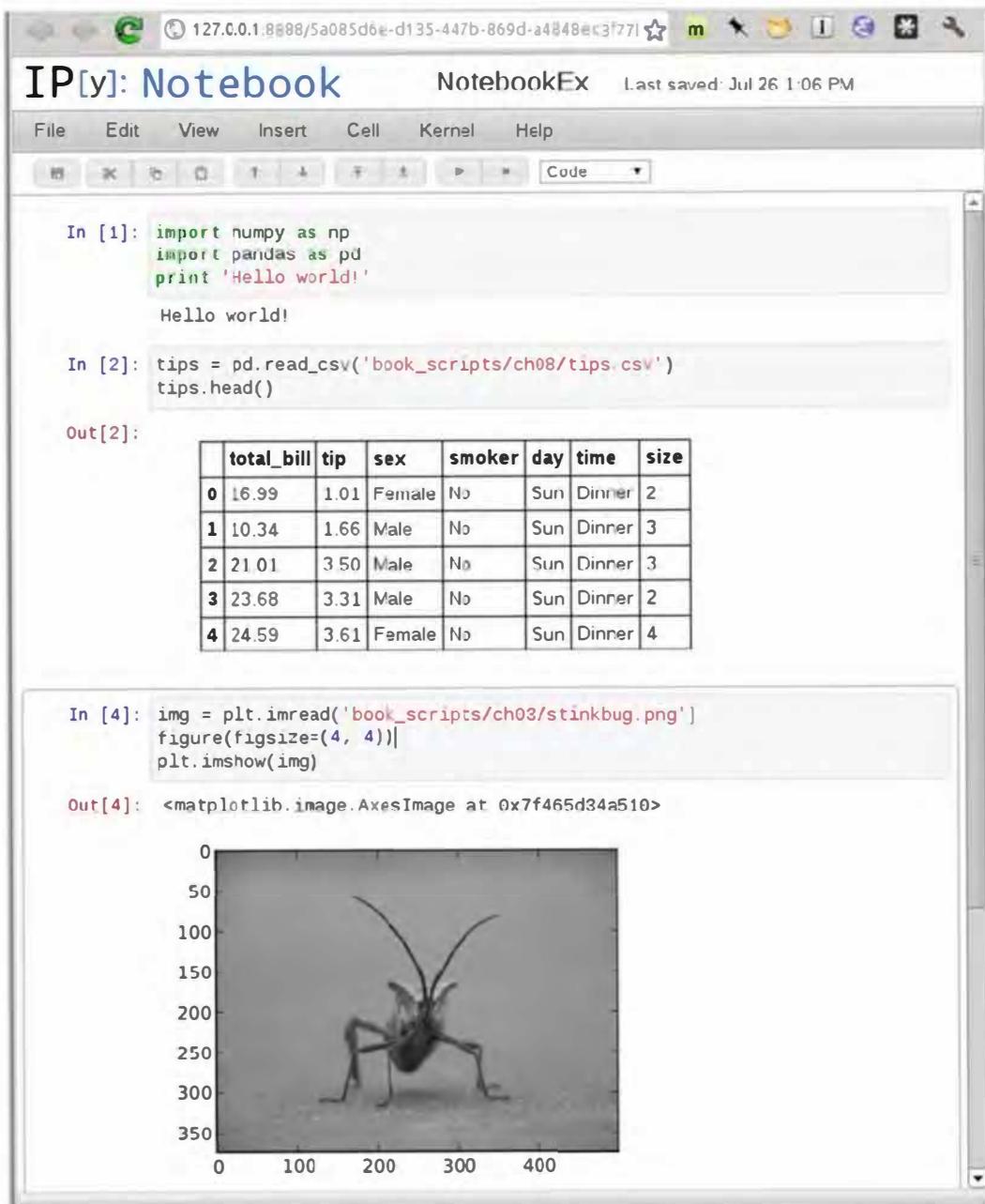


Рис. 3.4. Блокнот IPython

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 предложсния `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` из-за принятого в Python механизма однократной загрузки. Такое поведение отличается от некоторых других сред анализа данных, например MATLAB, в которых изменения кода распространяются автоматически². Справиться с этой проблемой можно двумя способами. Во-первых, использовать встроенную в Python функцию `reload`, изменив `test_script.py` следующим образом:

```
import some_lib
reload(some_lib)

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

При этом гарантируется получение новой копии `some_lib` 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 бы я написал `import some_lib`, а затем `dreload(some_lib)`, то был бы перезагружен 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:

² Поскольку модуль или пакет может импортироваться в нескольких местах программы, Python кэширует 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.

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    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` вы сможете просмотреть все переменные, определенные в `main`. В таком простом примере это неважно, но далее в книге будут рассмотрены сложные задачи анализа данных, в которых участвуют большие наборы, и их исследование может оказаться весьма полезным.

Плоское лучше вложенного

Глубоко вложенный код напоминает мне чешуи луковицы. Сколько чешуй придется снять при тестировании или отладке функции, чтобы добраться до интересующего кода? Идея «плоское лучше вложенного» – часть «Свода мудрости 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 10 мелких (скажем, не более 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 более мелкие.

Понятно, что я не призываю бросаться из одной крайности в другую, т. е. помешать весь код в один гигантский файл. Для отыскания разумной и интуитив-

по 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.

Дополнительные возможности IPython

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.

- Изменить цветовую схему.
- Изменить вид приглашений ввода и вывода или убрать пустую строку, печатаемую после Out и перед следующим приглашением In.
- Выполнить список произвольных предложений Python. Это может быть, например, импорт постоянно используемых модулей или вообще все, что должно выполняться сразу после запуска IPython.
- Включить расширения IPython, например магическую функцию %lprun в модуле line_profiler.
- Определить собственные магические функции или псевдонимы системных.

Все эти параметры задаются в конфигурационном файле ipython_config.py, находящемся в каталоге ~/.config/ipython/ в UNIX-системе или в каталоге %HOME%/.ipython/ в Windows. Где находится ваш домашний каталог, зависит от системы. 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/.config/ipython/profile_default/ipython_config.py
```

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 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.13 -- 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

In [1]:
```

Как всегда, дополнительные сведения о профилях и конфигурировании можно найти в документации по IPython в сети.

Благодарности

Материалы этой главы частично были заимствованы из великолепной документации, подготовленной разработчиками IPython. Я испытываю к ним бесконечную благодарность за создание этого восхитительного набора инструментов.



ГЛАВА 4.

Основы NumPy: массивы и векторные вычисления

Numerical Python, или сокращенно NumPy – краеугольный пакет для высокопроизводительных научных расчетов и анализа данных. Это фундамент, на котором возведены почти все описываемые в этой книге высокоуровневые инструменты. Вот лишь часть того, что предлагается.

- ndarray, быстрый и потребляющий мало памяти многомерный массив, предоставляющий векторные арифметические операции и возможность *укладывания*.
- Стандартные 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

Последний пункт – один из самых важных с точки зрения экосистемы. 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 приложением ему динамического и простого в использовании интерфейса.

Хотя сам по себе пакет NumPy почти не содержит средств высокоуровневого анализа данных, понимание массивов NumPy и ориентированных на эти массивы вычислений поможет гораздо эффективнее использовать инструменты типа pandas. Если вы только начинаете изучать Python и просто хотите познакомиться с тем, как pandas позволяет работать с данными, можете лишь бегло просмотреть эту главу. Более сложные средства NumPy, в частности укладывание, рассматриваются в главе 12.

В большинстве приложений для анализа данных основной интерес представляет следующая функциональность:

- быстрые векторные операции для персформатирования 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). Подробнее об этом см. главу 5.

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 высокогенеральный 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.



И в этой главе, и далее в книге я использую стандартное принятное в NumPy соглашение – всегда включать предложение `import numpy as np`. Конечно, никто не мешает добавить в программу предложение `from numpy import *`, чтобы не писать всюду `np.`, но это дурная привычка, от которой я хотел бы вас предостеречь.

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:

In [8]: `data`

Out [8]:
`array([[0.9526, -0.246 , -0.8856],
[0.5639, 0.2379, 0.9104]])`

In [9]: `data * 10`
Out [9]:

In [10]: `data + data`
Out [10]:

```
array([[ 9.5256, -2.4601, -8.8565],           array([[ 1.9051, -0.492 , -1.7713],  
       [ 5.6385,  2.3794,  9.104 ]])            [ 1.1277,  0.4759,  1.8208]])
```

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 [11]: data.shape  
Out[11]: (2, 3)  
  
In [12]: data.dtype  
Out[12]: 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 [13]: data1 = [6, 7.5, 8, 0, 1]  
In [14]: arr1 = np.array(data1)  
  
In [15]: arr1  
Out[15]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

Вложенные последовательности, например список списков одинаковой длины, можно преобразовать в многомерный массив:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]  
In [17]: arr2 = np.array(data2)  
  
In [18]: arr2  
Out[18]:  
array([[1, 2, 3, 4],  
       [5, 6, 7, 8]])  
  
In [19]: arr2.ndim
```

```
In [19]: 2
```

```
In [20]: arr2.shape
Out[20]: (2, 4)
```

Если не определено явно (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 [21]: arr1.dtype
Out[21]: dtype('float64')
```

```
In [22]: arr2.dtype
Out[22]: 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 [23]: np.zeros(10)
Out[23]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

```
In [24]: np.zeros((3, 6))
Out[24]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [25]: np.empty((2, 3, 2))
Out[25]:
array([[[ 4.94065646e-324,   4.94065646e-324],
        [ 3.87491056e-297,   2.46845796e-130],
        [ 4.94065646e-324,   4.94065646e-324]],
       [[ 1.90723115e+083,   5.73293533e-053],
        [-2.33568637e+124,  -6.70608105e-012],
        [ 4.42786966e+160,  1.27100354e+025]]])
```



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 [26]: np.arange(15)
Out[26]: 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
eye, identity	Создают единичную квадратную матрицу $N \times N$ (элементы на главной диагонали равны 1, все остальные – 0)

Тип данных для ndarray

Тип данных, или dtype – это специальный объект, который содержит информацию, необходимую ndarray для интерпретации содержимого блока памяти:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [29]: arr1.dtype In [30]: arr2.dtype

Out[29]: dtype('float64') Out[30]: 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 С 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 изучению. Часто нужно заботиться только об общем виде обрабатываемых данных, например: числа с плавающей точкой, комплексные, целые, булевы значения, строки или общие объекты 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
<code>int8, uint8</code>	<code>i1, u1</code>	Знаковое и беззнаковое 8-разрядное (1 байт) целое
<code>int16, uint16</code>	<code>i2, u2</code>	Знаковое и беззнаковое 16-разрядное (2 байта) целое
<code>int32, uint32</code>	<code>i4, u4</code>	Знаковое и беззнаковое 32-разрядное (4 байта) целое
<code>int64, uint64</code>	<code>i8, u8</code>	Знаковое и беззнаковое 64-разрядное (8 байт) целое
<code>float16</code>	<code>f2</code>	С плавающей точкой половинной точности
<code>float32</code>	<code>f4</code>	Стандартный тип с плавающей точкой одинарной точности. Совместим с типом C <code>float</code>
<code>float64</code>	<code>f8 или d</code>	Стандартный тип с плавающей точкой двойной точности. Совместим с типом C <code>double</code> и с типом Python <code>float</code>
<code>float128</code>	<code>f16</code>	С плавающей точкой расширенной точности
<code>complex64, complex128, complex256</code>	<code>c8, c16, c32</code>	Комплексные числа, вещественная и мнимая части которых представлены соответственно типами <code>float32</code> , <code>float64</code> и <code>float128</code>
<code>bool</code>	<code>?</code>	Булев тип, способный хранить значения <code>True</code> и <code>False</code>
<code>object</code>	<code>O</code>	Тип объекта Python
<code>string_</code>	<code>S</code>	Тип строки фиксированной длины (1 байт на символ). Например, строка длиной 10 имеет тип <code>S10</code> .
<code>unicode_</code>	<code>U</code>	Тип Unicode-строки фиксированной длины (количество байтов на символ зависит от платформы). Семантика такая же, как у типа <code>string_</code> (например, <code>U10</code>).

Можно явно преобразовать, или *привести* массив одного типа к другому, воспользовавшись методом `astype`:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])
In [32]: arr.dtype
Out[32]: dtype('int64')
In [33]: float_arr = arr.astype(np.float64)
In [34]: float_arr.dtype
Out[34]: 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 [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [36]: arr
```

```
Out[36]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [37]: arr.astype(np.int32)
```

```
Out[37]: 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 [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
```

```
In [39]: numeric_strings.astype(float)
```

```
Out[39]: array([ 1.25, -9.6, 42. ])
```

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 [40]: int_array = np.arange(10)
```

```
In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
```

```
In [42]: int_array.astype(calibers.dtype)
```

```
Out[42]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

На `dtype` можно сослаться также с помощью коротких кодов типа:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')
```

```
In [44]: empty_uint32
```

```
Out[44]:  
array([ 0, 0, 65904672, 0, 64856792, 0,  
       39438163, 0], dtype=uint32)
```



При вызове `astype` всегда создается новый массив (данные копируются), даже если новый `dtype` не отличается от старого.



Следует иметь в виду, что числа с плавающей точкой, например типа `float64` или `float32`, предоставляют дробные величины приближенно. В сложных вычислениях могут накапливаться ошибки округления, из-за которых сравнение возможно только с точностью до определенного числа десятичных знаков.

Операции между массивами и скалярами

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 [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])  
  
In [46]: arr  
Out[46]:  
array([[ 1.,  2.,  3.],  
       [ 4.,  5.,  6.]])  
  
In [47]: arr * arr  
Out[47]:  
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]])  
  
In [48]: arr - arr  
Out[48]:  
array([[ 0.,  0.,  0.],  
       [ 0.,  0.,  0.]])
```

Как легко догадаться, арифметические операции, в которых участвует скаляр, применяются к каждому элементу массива:

```
In [49]: 1 / arr  
Out[49]:  
array([[ 1. ,  0.5 ,  0.3333],  
       [ 0.25 ,  0.2 ,  0.1667]])  
  
In [50]: arr ** 0.5  
Out[50]:  
array([[ 1. ,  1.4142,  1.7321],  
       [ 2. ,  2.2361,  2.4495]])
```

Операции между массивами разного размера называются *укладыванием*, мы будем подробно рассматривать их в главе 12. 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 [51]: arr = np.arange(10)  
  
In [52]: arr  
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  
  
In [53]: arr[5]  
Out[53]: 5  
  
In [54]: arr[5:8]  
Out[54]: array([5, 6, 7])  
  
In [55]: arr[5:8] = 12  
  
In [56]: arr
```

```
Out[56]: 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.

```
In [57]: arr_slice = arr[5:8]
```

```
In [58]: arr_slice[1] = 12345
```

```
In [59]: arr
```

```
Out[59]: array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

```
In [60]: arr_slice[:] = 64
```

```
In [61]: arr
```

```
Out[61]: 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 тоже больше. В случае двумерного массива результатом индексирования с одним индексом является не скаляр, а одномерный массив:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [63]: arr2d[2]
```

```
Out[63]: 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 [64]: arr2d[0][2]
```

```
Out[64]: 3
```

```
In [65]: arr2d[0, 2]
```

```
Out[65]: 3
```

Рисунок 4.1 иллюстрирует индексирование двумерного массива.

		ось 1			
		0	1	2	
ось 0		0	0, 0	0, 1	0, 2
		1	1, 0	1, 1	1, 2
		2	2, 0	2, 1	2, 2

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 $2 \times 2 \times 3$:

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [67]: arr3d
Out[67]:
array([[[1, 2, 3],
       [4, 5, 6]],
      [[7, 8, 9],
       [10, 11, 12]]])
```

Тогда $\text{arr3d}[0]$ – массив размерности 2×3 :

```
In [68]: arr3d[0]
Out[68]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Выражению $\text{arr3d}[0]$ можно присвоить как скалярное значение, так и массив:

```
In [69]: old_values = arr3d[0].copy()
```

```
In [70]: arr3d[0] = 42
```

```
In [71]: arr3d
Out[71]:
array([[[42, 42, 42],
       [42, 42, 42]],
      [[7, 8, 9],
       [10, 11, 12]]])
```

```
In [72]: arr3d[0] = old_values
```

```
In [73]: arr3d
Out[73]:
```

```
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 [74]: arr3d[1, 0]
Out[74]: 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 [75]: arr[1:6]
Out[75]: array([ 1,  2,  3,  4, 64])
```

Для объектов большей размерности вариантов больше, потому что вырезать можно по одной или нескольким осям, сочетая с выбором отдельных элементов с помощью целых индексов. Вернемся к рассмотренному выше двумерному массиву `arr2d`. Вырезание из него выглядит несколько иначе:

In [76]: arr2d	In [77]: arr2d[:2]
Out[76]:	Out[77]:
array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])	array([[1, 2, 3], [4, 5, 6]])

Как видите, вырезание производится вдоль оси 0, первой оси. Поэтому срез содержит диапазон элементов вдоль этой оси. 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:

In [79]: arr2d[1, :2]	In [80]: arr2d[2, :1]
Out[79]: array([4, 5])	Out[80]: array([7])

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 [81]: arr2d[:, :1]
Out[81]:
array([[1],
       [4],
       [7]])
```

Разумеется, присваивание выражению-срезу означает присваивание всем элементам этого среза:

```
In [82]: arr2d[:2, 1:] = 0
```

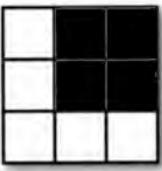
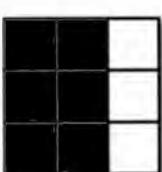
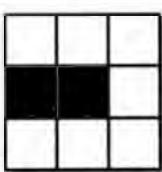
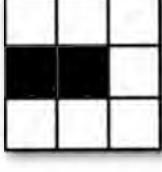
Выражение	Форма
<code>arr[:2, 1:]</code>	$(2, 2)$
	
<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	$(3,)$ $(3,)$ $(1, 3)$
	
<code>arr[:, :2]</code>	$(3, 2)$
	
<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	$(2,)$ $(1, 2)$
	

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 [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Will', 'Joe', 'Joe'])
```

```
In [84]: data = randn(7, 4)
```

```
In [85]: names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Will', 'Joe', 'Joe'],
```

```
In [86]: data  
Out[86]:
```

```
array([[-0.048 ,  0.5433, -0.2349,  1.2792],  
      [-0.268 ,  0.5465,  0.0939, -2.0445],  
      [-0.047 , -2.026 ,  0.7719,  0.3103],  
      [ 2.1452,  0.8799, -0.0523,  0.0672],  
      [-1.0023, -0.1698,  1.1503,  1.7289],  
      [ 0.1913,  0.4544,  0.4519,  0.5535],  
      [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

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 [87]: names == 'Bob'  
Out[87]: array([ True, False, False, True, False, False], dtype=bool)
```

Этот булев массив можно использовать для индексирования другого массива:

```
In [88]: data[names == 'Bob']  
Out[88]:  
array([[-0.048 ,  0.5433, -0.2349,  1.2792],  
      [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

Длина булева массива должна совпадать с длиной индексируемой оси. Можно даже сочетать булевые массивы со срезами и целыми числами (или последовательностями целых чисел, о чем речь пойдет ниже):

```
In [89]: data[names == 'Bob', 2:]  
Out[89]:  
array([[-0.2349,  1.2792],  
      [-0.0523,  0.0672]])
```

```
In [90]: data[names == 'Bob', 3]  
Out[90]: array([ 1.2792,  0.0672])
```

Чтобы выбрать все, кроме 'Bob', можно либо воспользоваться оператором сравнения `!=`, либо применить отрицание условия, обозначаемое знаком `-`:

```
In [91]: names != 'Bob'  
Out[91]: array([False, True, True, False, True, True, True], dtype=bool)  
  
In [92]: data[-(names == 'Bob')]  
Out[92]:  
array([[-0.268 ,  0.5465,  0.0939, -2.0445],  
      [-0.047 , -2.026 ,  0.7719,  0.3103],  
      [-1.0023, -0.1698,  1.1503,  1.7289],  
      [ 0.1913,  0.4544,  0.4519,  0.5535],  
      [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

Чтобы сформировать составное булево условие, включающее два из трех имен, воспользуемся булевыми операторами & (И) и | (ИЛИ):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')

In [94]: mask
Out[94]: array([True, False, True, True, True, False], dtype=bool)

In [95]: data[mask]
Out[95]:
array([[ -0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289]])
```

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.

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 [96]: data[data < 0] = 0

In [97]: data
Out[97]:
array([[ 0.      ,  0.5433,  0.      ,  1.2792],
       [ 0.      ,  0.5465,  0.0939,  0.      ],
       [ 0.      ,  0.      ,  0.7719,  0.3103],
       [ 2.1452,  0.8799,  0.      ,  0.0672],
       [ 0.      ,  0.      ,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.      ,  0.      ]])
```

Задать целые строки или столбцы с помощью одномерного булева массива тоже просто:

```
In [98]: data[names != 'Joe'] = 7

In [99]: data
Out[99]:
array([[ 7.      ,  7.      ,  7.      ,  7.      ],
       [ 0.      ,  0.5465,  0.0939,  0.      ],
       [ 7.      ,  7.      ,  7.      ,  7.      ],
       [ 7.      ,  7.      ,  7.      ,  7.      ],
       [ 7.      ,  7.      ,  7.      ,  7.      ],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.      ,  0.      ]])
```

Прихотливое индексирование

Термином *прихотливое индексирование* (fancy indexing) в NumPy обозначается индексирование с помощью целочисленных массивов. Допустим, имеется массив 8×4 :

```
In [100]: arr = np.empty((8, 4))

In [101]: for i in range(8):
....:     arr[i] = i

In [102]: arr
Out[102]:
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 [103]: arr[[4, 3, 0, 6]]
Out[103]:
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 [104]: arr[[-3, -5, -7]]
Out[104]:
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:

```
# о функции reshape см. главу 12
In [105]: arr = np.arange(32).reshape((8, 4))

In [106]: arr
Out[106]:
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 [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Давайте разберемся, что здесь происходит: отбираются элементы в позициях $(1, 0), (5, 3), (7, 1)$ и $(2, 2)$. В данном случае поведение прихотливого индексирования отличается от того, что ожидают многие пользователи (я в том числе): получить прямоугольный регион, образованный подмножеством строк и столбцов матрицы. Добиться этого можно, например, так:

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[108]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Другой способ – воспользоваться функцией `pr.ix_`, которая преобразует два одномерных массива целых чисел в индексатор, выбирающий квадратный регион:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
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 [110]: arr = np.arange(15).reshape((3, 5))

In [111]: arr
Out[111]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [112]: arr.T
Out[112]:
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 $X^T X$ с помощью метода `np.dot`:

```
In [113]: arr = np.random.randn(6, 3)
```

```
In [114]: np.dot(arr.T, arr)
Out[114]:
array([[ 2.584 ,  1.8753,  0.8888],
       [ 1.8753,  6.6636,  0.3884],
       [ 0.8888,  0.3884,  3.9781]])
```

Для массивов большей размерности метод `transpose` принимает кортеж номе-
ров осей, описывающий их перестановку (чтобы ум за разум совсем заехал):

```
In [115]: arr = np.arange(16).reshape((2, 2, 4))
```

```
In [116]: arr
Out[116]:
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7]],
      [[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
In [117]: arr.transpose((1, 0, 2))
Out[117]:
array([[[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]],
      [[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

Обычное транспонирование с помощью `.T` – частный случай перестановки
осей. У объекта `ndarray` имеется метод `swapaxes`, который принимает пару номе-
ров осей:

```
In [118]: arr
Out[118]:
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7]],
      [[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
In [119]: arr.swapaxes(1, 2)
Out[119]:
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.

Универсальные функции: быстрые поэлементные операции над массивами

Универсальной функцией, или *u-функцией* называется функция, которая вы-
полняет 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 [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.        ,  1.        ,  1.4142   ,  1.7321   ,  2.        ,
       2.2361   ,  2.4495   ,  2.6458   ,  2.8284   ,  3.        ])

In [122]: np.exp(arr)
Out[122]:
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 2 массива (и потому называются *бинарными*) и возвращают один результирующий массив:

```
In [123]: x = randn(8)
In [124]: y = randn(8)
In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655,  0.9222,  0.446 ,
       -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323,  0.4788,  0.4315,
       -0.7147])

In [127]: np.maximum(x, y) # поэлементный максимум
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
       -0.7147])
```

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 [128]: arr = randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808,  0.0636, -0.386 ,  0.1393, -0.8806,  0.9363, -0.883 ]),
 array([-2.,  4., -3.,  5., -3.,  3., -6.]))
```

В таблицах 4.3 и 4.4 перечислены имеющиеся u-функции.

Таблица 4.3. Унарные и-функции

Функция	Описание
abs, fabs	Вычислить абсолютное значение целых, вещественных или комплексных элементов массива. Для вещественных данных <code>fabs</code> работает быстрее
sqrt	Вычислить квадратный корень из каждого элемента. Эквивалентно <code>arr ** 0.5</code>
square	Вычислить квадрат каждого элемента. Эквивалентно <code>arr ** 2</code>
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 <code>dtype</code>
modf	Вернуть дробные и целые части массива в виде отдельных массивов
isnan	Вернуть булев массив, показывающий, какие значения являются <code>NaN</code> (не числами)
isfinite, isinf	Вернуть булев массив, показывающий, какие элементы являются конечными (не <code>inf</code> и не <code>NaN</code>) 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 <code>not x</code> для каждого элемента. Эквивалентно <code>-arr</code> .

Таблица 4.4. Бинарные и-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

Функция	Описание
power	Возвести элементы первого массива в степени, указанные во втором массиве
maximum, fmax	Поэлементный максимум. Функция <code>fmax</code> игнорирует значения <code>NaN</code>
minimum, fmin	Поэлементный минимум. Функция <code>fmin</code> игнорирует значения <code>NaN</code>
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 <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code>
logical_and, logical_or, logical_xor	Вычислить логическое значение истинности логических операций. Эквивалентны инфиксным операторам <code>&</code> , <code> </code> , <code>^</code>

Обработка данных 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 говоря, векторные операции с массивами выполняются на один-два (а то и больше) порядка быстрее, чем эквивалентные операции на чистом Python. Позже, в главе 12 я расскажу об *укладывании*, единственном 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 [130]: points = np.arange(-5, 5, 0.01) # 1000 равноотстоящих точек
```

```
In [131]: xs, ys = np.meshgrid(points, points)
```

```
In [132]: ys
```

```
Out[132]:
```

```
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 [134]: import matplotlib.pyplot as plt
```

```
In [135]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [136]: z
```

```
Out[136]:
```

```
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]])
```

```
In [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
```

```
Out[137]: <matplotlib.colorbar.Colorbar instance at 0x4e46d40>
```

```
In [138]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values")
Out[138]: <matplotlib.text.Text at 0x4565790>
```

На рис. 4.3 показан результат применения функции `imshow` из библиотеки `matplotlib` для создания изображения по двумерному массиву значений функции.

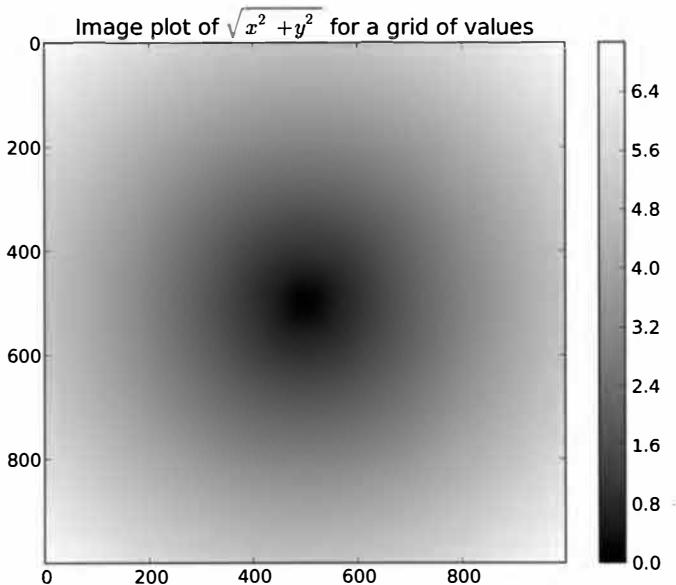


Рис. 4.3. График функции двух переменных на сетке

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 [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
In [142]: cond = np.array([True, False, True, True, False])
```

Допустим, что 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 [143]: result = [(x if c else y)
....: for x, y, c in zip(xarr, yarr, cond)]
```

```
In [144]: result
```

```
Out[144]: [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 [145]: result = np.where(cond, xarr, yarr)
```

```
In [146]: result
```

```
Out[146]: 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 применяется, чтобы создать новый массив на основе существующего. Предположим, имеется матрица со случайными данными, и мы хотим заменить все положительные значение на 2, а все отрицательные – на -2. С помощью `np.where` сделать это очень просто:

```
In [147]: arr = randn(4, 4)
```

```
In [148]: arr
```

```
Out[148]:
```

```
array([[ 0.6372, 2.2043, 1.7904, 0.0752],
[-1.5926, -1.1536, 0.4413, 0.3483],
[-0.1798, 0.3299, 0.7827, -0.7585],
[ 0.5857, 0.1619, 1.3583, -1.3865]])
```

```
In [149]: np.where(arr > 0, 2, -2)
```

```
Out[149]:
```

```
array([[ 2, 2, 2, 2],
[-2, -2, 2, 2],
[-2, 2, 2, -2],
[ 2, 2, 2, -2]])
```

```
In [150]: np.where(arr > 0, 2, arr) # только положительные заменить на 2
```

```
Out[150]:
```

```
array([[ 2., 2., 2., 2.],
[-1.5926, -1.1536, 2., 2.]])
```

```
[ -0.1798,      2. , 2. , -0.7585] ,
[ 2. ,      2. , 2. , -1.3865]])
```

Передавать `where` можно не только массивы одинакового размера или скаляры. При некоторой изобретательности `where` позволяет выразить и более сложную логику. Пусть есть два булева массива `cond1` и `cond2`, и мы хотим сопоставить различные значения каждой из четырех возможных комбинаций двух булевых значений:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)
```

Хотя сразу это не очевидно, показанный цикл `for` можно преобразовать во вложенное выражение `where`:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

В этом конкретном примере можно также воспользоваться тем фактом, что во всех вычислениях булевы значения трактуются как 0 и 1, поэтому выражение можно записать и в виде такой арифметической операции (хотя выглядит она загадочно):

```
result = 1 * cond1 + 2 * cond2 + 3 * -(cond1 | cond2)
```

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 как с помощью метода экземпляра массива, так и функции на верхнем уровне NumPy:

```
In [151]: arr = np.random.randn(5, 4) # нормально распределенные данные

In [152]: arr.mean()
Out[152]: 0.062814911084854597

In [153]: np.mean(arr)
Out[153]: 0.062814911084854597

In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Функции типа `mean` и `sum` принимают необязательный аргумент `axis`, при наличии которого вычисляется статистика по заданной оси, и в результате порождается массив на единицу меньшей размерности:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833, 0.2844, 0.6574, 0.6743, -0.0187])

In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189, 1.4044, 4.5712])
```

Другие методы, например `cumsum` и `cumprod`, ничего не агрегируют, а порождают массив промежуточных результатов:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [158]: arr.cumsum(0)           In [159]: arr.cumprod(1)
Out[158]: array([[ 0,  1,  2],
                 [ 3,  5,  7],
                 [ 9, 12, 15]])          Out[159]: 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
<code>sum</code>	Сумма элементов всего массива или вдоль одной оси. Для массивов нулевой длины функция <code>sum</code> возвращает 0
<code>mean</code>	Среднее арифметическое. Для массивов нулевой длины равно <code>NaN</code>
<code>std</code> , <code>var</code>	Стандартное отклонение и дисперсия, соответственно. Может быть задано число степеней свободы (по умолчанию знаменатель равен <code>n</code>)
<code>min</code> , <code>max</code>	Минимум и максимум
<code>argmin</code> , <code>argmax</code>	Индексы минимального и максимального элемента
<code>cumsum</code>	Нарастающая сумма с начальным значением 0
<code>cumprod</code>	Нарастающее произведение с начальным значением 1

Методы булевых массивов

В вышеупомянутых методах булевые значения приводятся к 1 (`True`) и 0 (`False`). Поэтому функция `sum` часто используется для подсчета значений `True` в булевом массиве:

```
In [160]: arr = randn(100)

In [161]: (arr > 0).sum() # Количество положительных значений
Out[161]: 44
```

Но существуют 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` – что все значения в массиве равны `True`:

```
In [162]: bools = np.array([False, False, True, False])
```

```
In [163]: bools.any()
```

```
Out[163]: True
```

```
In [164]: bools.all()
```

```
Out[164]: 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 [165]: arr = randn(8)
```

```
In [166]: arr
```

```
Out[166]:
```

```
array([ 0.6903,  0.4678,  0.0968, -0.1349,  0.9879,  0.0185, -1.3147,
       -0.5425])
```

```
In [167]: arr.sort()
```

```
In [168]: arr
```

```
Out[168]:
```

```
array([-1.3147, -0.5425, -0.1349,  0.0185,  0.0968,  0.4678,  0.6903,
       0.9879])
```

В многомерных массивах можно сортировать на месте одномерные секции вдоль любой оси, для этого нужно передать `sort` номер оси:

```
In [169]: arr = randn(5, 3)
```

```
In [170]: arr
```

```
Out[170]:
```

```
array([[ -0.7139, -1.6331, -0.4959],
       [ 0.8236, -1.3132, -0.1935],
       [-1.6748,  3.0336, -0.863 ],
       [-0.3161,  0.5362, -2.468 ],
       [ 0.9058,  1.1184, -1.0516]])
```

```
In [171]: arr.sort(1)
```

```
In [172]: arr
```

```
Out[172]:
```

```
array([[-1.6331, -0.7139, -0.4959],
```

```
[ -1.3132, -0.1935,  0.8236],
[ -1.6748, -0.863 ,  3.0336],
[ -2.468 , -0.3161,  0.5362],
[ -1.0516,  0.9058,  1.1184]])
```

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 [173]: large_arr = randn(1000)
In [174]: large_arr.sort()

In [175]: large_arr[int(0.05 * len(large_arr))] # 5%-ный квантиль
Out[175]: -1.5791023260896004
```

Дополнительные сведения о методах сортировки в NumPy и о более сложных приемах, например косвенной сортировке, см. главу 12. 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 имеется основные теоретико-множественные операции для одномерных массивов. Пожалуй, самой употребительной является `np.unique`, она возвращает отсортированное множество уникальных значений в массиве:

```
In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [177]: np.unique(names)
Out[177]:
array(['Bob', 'Joe', 'Will'],
      dtype='|S4')

In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [179]: np.unique(ints)
Out[179]: array([1, 2, 3, 4])
```

Сравните `np.unique` с альтернативой на чистом Python:

```
In [180]: sorted(set(names))
Out[180]: ['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
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

Файловый ввод-вывод массивов

NumPy умеет сохранять на диске и загружать с диска данные в текстовом или двоичном формате. В последующих главах мы узнаем, как в pandas считываются в память табличные данные.

Хранение массивов на диске в двоичном формате

`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 [183]: arr = np.arange(10)
```

```
In [184]: np.save('some_array', arr)
```

Если путь к файлу не заканчивается суффиксом .npy, то он будет добавлен. Хранящийся на диске массив можно загрузить в память функцией `np.load`:

```
In [185]: np.load('some_array.npy')
Out[185]: 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 [186]: 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 [187]: arch = np.load('array_archive.npz')

In [188]: arch['b']
Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Сохранение и загрузка текстовых файлов

Загрузка текста из файлов – вполне стандартная задача. Многообразие имеющихся в Python функций для чтения и записи файлов может смутиТЬ начинающего, поэтому я буду рассматривать главным образом функции `read_csv` и `read_table` из библиотеки `pandas`. Иногда бывает полезно загружать данные в массивы NumPy с помощью функции `np.loadtxt` или более специализированной `np.genfromtxt`.

У этих функций много параметров, которые позволяют задавать разные разделители и функции-конвертеры для столбцов, пропускать строки и делать много других вещей. Рассмотрим простой случай загрузки файла с разделителями-запятыми (CSV):

```
In [191]: !cat array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
-0.126410,0.268607,-0.695724,0.047428
-1.484413,0.004176,-0.744203,0.005487
2.302869,0.200131,1.670238,-1.881090
-0.193230,1.047233,0.482803,0.960334
```

Его можно следующим образом загрузить в двумерный массив:

```
In [192]: arr = np.loadtxt('array_ex.txt', delimiter=',')
In [193]: arr
Out[193]:
array([[ 0.5801,   0.1867,   1.0407,   1.1344],
       [ 0.1942,  -0.6369,  -0.9387,   0.1241],
       [-0.1264,   0.2686,  -0.6957,   0.0474],
       [-1.4844,   0.0042,  -0.7442,   0.0055],
       [ 2.3029,   0.2001,   1.6702,  -1.8811],
       [-0.1932,   1.0472,   0.4828,   0.9603]])
```

Функция `np.savetxt` выполняет обратную операцию: записывает массив в текстовый файл с разделителями. Функция `genfromtxt` аналогична `loadtxt`, но ориентирована на структурные массивы и обработку отсутствующих данных. Подробнее о структурных массивах см. главу 12.



Дополнительные сведения о чтении и записи файлов, особенно табличных, приведены в последующих главах, касающихся библиотеки `pandas` и объектов `DataFrame`.

Линейная алгебра

Операции линейной алгебры – 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` – как в виде метода массива, так и в виде функции в пространстве имен `numpy`:

```
In [194]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
In [195]: y = np.array([[6., 23.], [-1, 7], [8, 9]])
In [196]: x
Out[196]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
In [197]: y
Out[197]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
In [198]: x.dot(y)      # эквивалентно np.dot(x, y)
Out[198]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

Произведение двумерного массива и одномерного массива подходящего размера дает одномерный массив:

```
In [199]: np.dot(x, np.ones(3))
Out[199]: array([ 6., 15.])
```

В модуле `numpy.linalg` имеет 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, поставляемой компанией Intel:

```
In [201]: from numpy.linalg import inv, qr
In [202]: x = randn(5, 5)
In [203]: mat = x.T.dot(x)
In [204]: inv(mat)
Out[204]:
array([[ 3.0361, -0.1808, -0.6878, -2.8285, -1.1911],
       [-0.1808,  0.5035,  0.1215,  0.6702,  0.0956],
       [-0.6878,  0.1215,  0.2904,  0.8081,  0.3049],
       [-2.8285,  0.6702,  0.8081,  3.4152,  1.1557],
       [-1.1911,  0.0956,  0.3049,  1.1557,  0.6051]])
```



```
In [205]: mat.dot(inv(mat))
Out[205]:
```

```
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 [206]: q, r = qr(mat)
```

```
In [207]: r
```

```
Out[207]:
```

```
array([[-6.9271,  7.389 ,  6.1227, -7.1163, -4.9215],
       [ 0.        , -3.9735, -0.8671,  2.9747, -5.7402],
       [ 0.        ,  0.        , -10.2681,  1.8909,  1.6079],
       [ 0.        ,  0.        ,  0.        , -1.2996,  3.3577],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.5571]])
```

В табл. 4.7 перечислены наиболее употребительные функции линейной алгебры.

Таблица 4.7. Наиболее употребительные функции из модуля `numpy.linalg`

Функция	Описание
<code>diag</code>	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
<code>dot</code>	Вычисляет произведение матриц
<code>trace</code>	Вычисляет след матрицы – сумму диагональных элементов
<code>det</code>	Вычисляет определитель матрицы
<code>eig</code>	Вычисляет собственные значения и собственные векторы квадратной матрицы
<code>inv</code>	Вычисляет обратную матрицу
<code>pinv</code>	Вычисляет псевдообратную матрицу Мура-Пенроуза для квадратной матрицы
<code>qr</code>	Вычисляет QR-разложение
<code>svd</code>	Вычисляет сингулярное разложение (SVD)
<code>solve</code>	Решает линейную систему $Ax = b$, где A – квадратная матрица
<code>lstsq</code>	Вычисляет решение уравнения $y = xb$ по методу наименьших квадратов

Генерация случайных 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 [208]: samples = np.random.normal(size=(4, 4))
```

```
In [209]: samples
```

```
Out[209]:
```

```
array([[ 0.1241,  0.3026,  0.5238,  0.0009],
       [ 1.3438, -0.7135, -0.8312, -2.3702],
      [-1.8608, -0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329, -2.3594]])
```

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 [210]: from random import normalvariate

In [211]: N = 1000000

In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop

In [213]: %timeit np.random.normal(size=N)
10 loops, best of 3: 57.7 ms per loop
```

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
<code>seed</code>	Задает начальное значение генератора случайных чисел
<code>permutation</code>	Возвращает случайную перестановку последовательности или диапазона
<code>shuffle</code>	Случайным образом переставляет последовательность на месте
<code>rand</code>	Случайная выборка с равномерным распределением
<code>randint</code>	Случайная выборка целого числа из заданного диапазона
<code>randn</code>	Случайная выборка с нормальным распределением со средним 0 и стандартным отклонением 1 (интерфейс похож на MATLAB)
<code>binomial</code>	Случайная выборка с биномиальным распределением
<code>normal</code>	Случайная выборка с нормальным (гауссовым) распределением
<code>beta</code>	Случайная выборка с бета-распределением
<code>chisquare</code>	Случайная выборка с распределением хи-квадрат
<code>gamma</code>	Случайная выборка с гамма-распределением
<code>uniform</code>	Случайная выборка с равномерным распределением на полуинтервале [0, 1)

Пример: случайное блуждание

Проиллюстрируем операции с массивами на примере случайного блуждания. Сначала рассмотрим случайное блуждание с начальной точкой 0 и шагами 1 и -1 , выбираемыми с одинаковой вероятностью. Вот реализация одного случайного блуждания с 1000 шагами на чистом Python с помощью встроенного модуля `random`:

```
import random
position = 0
walk = [position]
steps = 1000
for i in xrange(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.

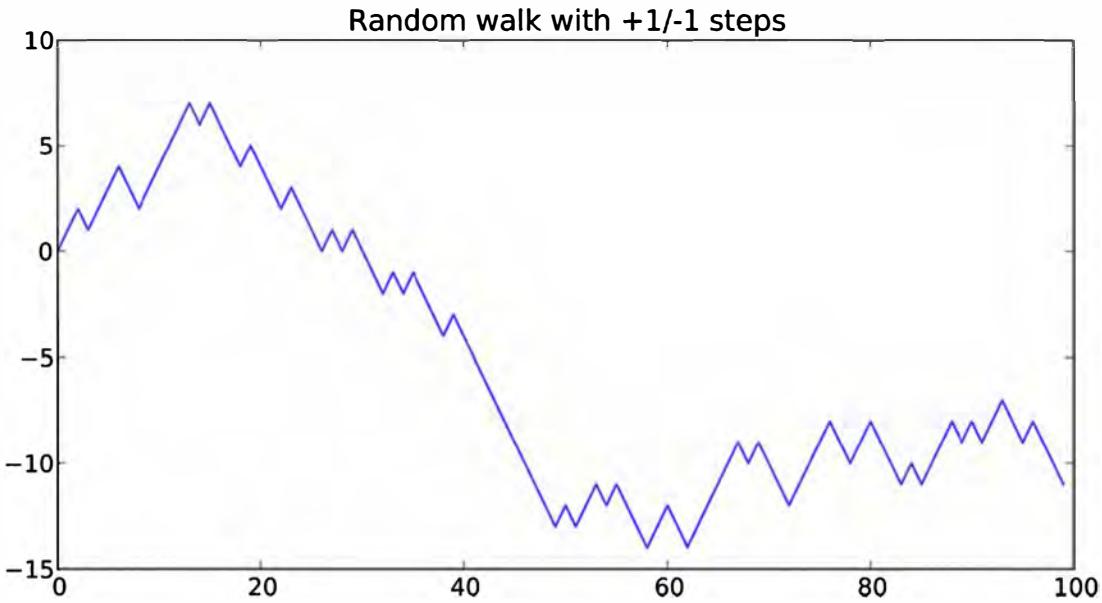


Рис. 4.4. Простое случайное блуждание

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 [215]: nsteps = 1000
In [216]: draws = np.random.randint(0, 2, size=nsteps)
In [217]: steps = np.where(draws > 0, 1, -1)
In [218]: walk = steps.cumsum()
```

Теперь можно приступить к вычислению статистики, например минимального и максимального значения на траектории блуждания:

In [219]: <code>walk.min()</code>	In [220]: <code>walk.max()</code>
Out[219]: -3	Out[220]: 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 [221]: (np.abs(walk) >= 10).argmax()  
Out[221]: 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 генерируют двумерный массив случайных чисел, и мы сможем вычислить нарастающие суммы по строкам, т. е. все 5000 случайных блужданий за одну операцию:

```
In [222]: nwalks = 5000  
  
In [223]: nsteps = 1000  
  
In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1  
  
In [225]: steps = np.where(draws > 0, 1, -1)  
  
In [226]: walks = steps.cumsum(1)  
  
In [227]: walks  
Out[227]:  
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 [228]: walks.max()           In [229]: walks.min()  
Out[228]: 138                 Out[229]: -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 [230]: hits30 = (np.abs(walks) >= 30).any(1)
```

```
In [231]: hits30
```

```
Out[231]: array([False, True, False, ..., False, True, False], dtype=bool)
```

```
In [232]: hits30.sum() # Сколько раз достигалось 30 или -30
```

```
Out[232]: 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 [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
```

```
In [234]: crossing_times.mean()
```

```
Out[234]: 498.88973607038122
```

Поэкспериментируйте с другими распределениями шагов, не ограничиваясь подбрасыванием правильной монеты. Всего-то и нужно, что взять другую функцию генерации случайных чисел, например, `normal` для генерации шагов с нормальным распределением с заданными средним и стандартным отклонением:

```
In [235]: steps = np.random.normal(loc=0, scale=0.25,  
.....: size=(nwalks, nsteps))
```



ГЛАВА 5.

Первое знакомство с pandas

Библиотека pandas будет основным предметом изучения в последующих главах. Она содержит высокоуровневые структуры данных и средства манипуляции ими, спроектированные так, чтобы обеспечить простоту и высокую скорость анализа данных на Python. Эта библиотека построена поверх NumPy, поэтому ей легко пользоваться в приложениях, ориентированных на NumPy.

Для сведения – я начал разрабатывать pandas в начале 2008 года, когда работал в компании AQR, занимающейся количественным анализом инвестиционных рисков. Тогда я столкнулся с требованиями, которые не мог полностью удовлетворить ни один из имеющихся в моем распоряжении инструментов:

- Структуры данных с помеченными осями, поддерживающие автоматическое или явное выравнивание. Это помогает избежать типичных ошибок, связанных с невыровненностью данных и работой с данными, поступившими из разных источников и по-разному проиндексированных.
- Встроенная функциональность для работы с временными рядами.
- Одни и те же структуры должны быть пригодны для обработки как временных рядов, так и данных иного характера.
- Арифметические операции и операции редуцирования (например, суммирование вдоль оси) должны «пробрасывать» метаданные (метки осей).
- Гибкая обработка отсутствующих данных.
- Объединение и другие реляционные операции, имеющиеся в популярных базах данных (например, на основе SQL).

Я хотел, чтобы все это было сосредоточено в одном месте и предпочтительно написано на языке, подходящем для разработки программ общего назначения. Python казался неплохим кандидатом, но в то время в нем не было встроенных структур данных и инструментов, поддерживающих нужную мне функциональность.

За прошедшие четыре года pandas превратилась в довольно обширную библиотеку, способную решать куда более широкий круг задач обработки данных, чем я планировал первоначально, но расширялась она, не принося в жертву простоту и удобство использования, к которым я стремился с самого начала. Надеюсь, что, прочитав эту книгу, вы так же, как и я, станете считать ее незаменимым инструментом.

В этой книге используются следующие соглашения об импорте для pandas:

```
In [1]: from pandas import Series, DataFrame
```

```
In [2]: import pandas as pd
```

Таким образом, увидев в коде строку `pd.`, знайте, что это ссылка на pandas. Объекты Series и DataFrame используются так часто, что я счел полезным импортировать их в локальное пространство имен.

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 массив данных (любого типа, поддерживаемого NumPy) и ассоциированный с ним массив меток, который называется *индексом*. Простейший объект Series состоит только из массива данных:

```
In [4]: obj = Series([4, 7, -5, 3])
```

```
In [5]: obj  
Out[5]:  
0    4  
1    7  
2   -5  
3    3
```

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 [6]: obj.values  
Out[6]: array([ 4,  7, -5,  3])
```

```
In [7]: obj.index  
Out[7]: Int64Index([0, 1, 2, 3])
```

Часто желательно создать объект Series с индексом, идентифицирующим каждый элемент данных:

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
In [9]: obj2
```

```
Out[9]:
```

```
d    4  
b    7  
a   -5  
c    3
```

```
In [10]: obj2.index
```

```
Out[10]: Index(['d', 'b', 'a', 'c'], dtype=object)
```

В отличие от обычного массива NumPy, для выборки одного или нескольких элементов из объекта Series можно использовать значения индекса:

```
In [11]: obj2['a']
```

```
Out[11]: -5
```

```
In [12]: obj2['d'] = 6
```

```
In [13]: obj2[['c', 'a', 'd']]
```

```
Out[13]:
```

```
c    3  
a   -5  
d    6
```

Операции с массивом NumPy, например фильтрация с помощью булева массива, скалярное умножение или применение математических функций, сохраняют связь между индексом и значением:

```
In [14]: obj2
```

```
Out[14]:
```

```
d    6  
b    7  
a   -5  
c    3
```

```
In [15]: obj2[obj2 > 0]
```

```
Out[15]:
```

```
d    6  
b    7  
c    3
```

```
In [16]: obj2 * 2
```

```
Out[16]:
```

```
d    12  
b    14  
a   -10  
c     6
```

```
In [17]: np.exp(obj2)
```

```
Out[17]:
```

```
d    403.428793  
b   1096.633158  
a     0.006738  
c   20.085537
```

Объект Series можно также представлять себе как упорядоченный словарь фиксированной длины, поскольку он отображает индекс на данные. Его можно передавать многим функциям, ожидающим получить словарь:

```
In [18]: 'b' in obj2
```

```
Out[18]: True
```

```
In [19]: 'e' in obj2
```

```
Out[19]: False
```

Если имеется словарь Python, содержащий данные, то из него можно создать объект Series:

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [21]: obj3 = Series(sdata)

In [22]: obj3
Out[22]:
Ohio    35000
Oregon  16000
Texas   71000
Utah    5000
```

Если передается только словарь, то в получившемся объекте Series ключи будут храниться в индексе по порядку:

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [24]: obj4 = Series(sdata, index=states)

In [25]: obj4
Out[25]:
California      NaN
Ohio            35000
Oregon          16000
Texas           71000
```

В данном случае 3 значения, найденные 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. Иногда, говоря об отсутствующих данных, я буду употреблять термин «NA». Для распознавания отсутствующих данных в pandas следует использовать функции isnull и notnull:

In [26]: pd.isnull(obj4)	In [27]: pd.notnull(obj4)
Out[26]:	Out[27]:
California True	California False
Ohio False	Ohio True
Oregon False	Oregon True
Texas False	Texas True

У объекта Series есть также методы экземпляра:

```
In [28]: obj4.isnull()
Out[28]:
California  True
Ohio        False
Oregon      False
Texas       False
```

Более подробно работа с отсутствующими данными будет обсуждаться ниже в этой главе. Для многих приложений особенно важно, что при выполнении ариф-

метических операций объект Series автоматически выравнивает данные, которые проиндексированы в разном порядке:

```
In [29]: obj3
Out[29]:
Ohio    35000
Oregon   16000
Texas    71000
Utah     5000

In [30]: obj4
Out[30]:
California      NaN
Ohio            35000
Oregon          16000
Texas           71000

In [31]: obj3 + obj4
Out[31]:
California      NaN
Ohio            70000
Oregon          32000
Texas           142000
Utah            NaN
```

Вопрос о выравнивании данных будет рассмотрен отдельно.

И у самого объекта Series, и у его индекса имеется атрибут name, тесно связанный с другими частями функциональности pandas:

```
In [32]: obj4.name = 'population'
In [33]: obj4.index.name = 'state'

In [34]: obj4
Out[34]:
state
California      NaN
Ohio            35000
Oregon          16000
Texas           71000
Name: population
```

Индекс объекта Series можно изменить на месте с помощью присваивания:

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj
Out[36]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
```

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. По сравнению с другими похожими на DataFrame структурами,

которые вам могли встречаться раньше (например, `data.frame` в языке R), операции со строками и столбцами в DataFrame в первом приближении симметричны. 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'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

Для получившегося DataFrame автоматически будет построен индекс, как и в случае Series, и столбцы расположатся по порядку:

```
In [38]: frame
Out[38]:
   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 [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
   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
```

Как и в случае Series, если запросить столбец, которого нет в data, то он будет заполнен значениями NaN:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                           ....:                  index=['one', 'two', 'three', 'four', 'five'])
In [41]: frame2
```

```
Out[41]:
   year  state    pop  debt
0  2000  Ohio    1.5  NaN
1  2001  Ohio    1.7  NaN
2  2002  Ohio    3.6  NaN
3  2001  Nevada  2.4  NaN
4  2002  Nevada  2.9  NaN
```

```
In [42]: frame2.columns
Out[42]: Index(['year', 'state', 'pop', 'debt'], dtype=object)
```

Столбец DataFrame можно извлечь как объект Series, воспользовавшись нотацией словарей, или с помощью атрибута:

<pre>In [43]: frame2['state'] Out[43]: one Ohio two Ohio three Ohio four Nevada five Nevada Name: state</pre>	<pre>In [44]: frame2.year Out[44]: one 2000 two 2001 three 2002 four 2001 five 2002</pre>
--	---

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, для чего есть два метода, один из них – ix с указанием индексного поля (подробнее об этом ниже):

```
In [45]: frame2.ix['three']
Out[45]:
year    2002
state   Ohio
pop     3.6
debt    NaN
Name: three
```

Столбцы можно модифицировать путем присваивания. Например, пустому столбцу 'debt' можно было бы присвоить скалярное значение или массив значений:

```
In [46]: frame2['debt'] = 16.5
In [47]: frame2
Out[47]:
   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
In [48]: frame2['debt'] = np.arange(5.)
In [49]: frame2
```

```
Out[49]:
      year  state    pop   debt
one   2000  Ohio    1.5     0
two   2001  Ohio    1.7     1
three 2002  Ohio    3.6     2
four  2001  Nevada  2.4     3
five  2002  Nevada  2.9     4
```

Когда столбцу присваивается список или массив, длина значения должна совпадать с длиной DataFrame. Если же присваивается объект Series, то он будет точно согласован с индексом DataFrame, а в «дырки» будут вставлены значения NA:

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [51]: frame2['debt'] = val

In [52]: frame2
Out[52]:
      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
```

Присваивание несуществующему столбцу приводит к созданию нового столбца. Для удаления столбцов служит ключевое слово `del`, как и в обычном словаре:

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'

In [54]: frame2
Out[54]:
      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

In [55]: del frame2['eastern']

In [56]: frame2.columns
Out[56]: 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 [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},  
....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Если передать его конструктору DataFrame, то ключи внешнего словаря будут интерпретированы как столбцы, а ключи внутреннего словаря – как индексы строк:

```
In [58]: frame3 = DataFrame(pop)
```

```
In [59]: frame3  
Out[59]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Разумеется, результат можно транспонировать:

```
In [60]: frame3.T  
Out[60]:
```

	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 [61]: DataFrame(pop, index=[2001, 2002, 2003])  
Out[61]:
```

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

Словари объектов Series интерпретируются очень похоже:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][:-1],  
....: 'Nevada': frame3['Nevada'][:2]}  
  
In [63]: DataFrame(pdata)  
Out[63]:
```

	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 [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'  
  
In [65]: frame3
```

```
Out[65]:
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 [66]: frame3.values
Out[66]:
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 [67]: frame2.values
Out[67]:
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]]], 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
Структурный 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

Тип	Примечания
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 [68]: obj = Series(range(3), index=['a', 'b', 'c'])

In [69]: index = obj.index

In [70]: index
Out[70]: Index([a, b, c], dtype=object)

In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Индексные объекты неизменяемы, т. е. пользователь не может их модифицировать:

```
In [72]: index[1] = 'd'
-----
Exception                                Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302     def __setitem__(self, key, value):
    303         """Disable the setting of values."""
--> 304     raise Exception(str(self.__class__) + ' object is immutable')
    305
    306     def __getitem__(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

Неизменяемость важна для того, чтобы несколько структур данных могли совместно использовать один и тот же индексный объект, не опасаясь его повредить:

```
In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index
Out[75]: True
```

В табл. 5.2 перечислены включенные в библиотеку индексные классы. При некотором усилии можно даже создать подклассы класса Index, если требуется реализовать специализированную функциональность индексирования вдоль оси.



Многим пользователям подробная информация об индексных объектах не нужна, но они, тем не менее, являются важной частью модели данных pandas.

Таблица 5.2. Основные индексные объекты в pandas

Тип	Примечания
Index	Наиболее общий индексный объект, представляющий оси в массиве NumPy, состоящем из объектов Python
Int64Index	Специализированный индекс для целых значений
MultiIndex	«Иерархический» индекс, представляющий несколько уровней индексирования по одной оси. Можно считать аналогом массива кортежей
DatetimeIndex	Хранит временные метки с наносекундной точностью (представлены типом данных NumPy datetime64)
PeriodIndex	Специализированный индекс для данных о периодах (временных промежутках)

Индексный объект не только похож на массив, но и ведет себя как множество фиксированного размера:

```
In [76]: frame3
Out[76]:
state    Nevada    Ohio
year
2000      NaN     1.5
2001      2.4     1.7
2002      2.9     3.6

In [77]: 'Ohio' in frame3.columns
Out[77]: True

In [78]: 2003 in frame3.index
Out[78]: False
```

У любого объекта Index есть ряд свойств и методов для ответа на типичные вопросы о хранящихся в нем данных. Они перечислены в табл. 5.3.

Таблица 5.3. 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

Метод	Описание
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>i</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>i</i>
is_monotonic	Возвращает True, если каждый элемент больше или равен предыдущему
is_unique	Возвращает True, если в индексе нет повторяющихся значений
unique	Вычисляет массив уникальных значений в индексе

Базовая функциональность

В этом разделе мы рассмотрим фундаментальные основы взаимодействия с данными, хранящимися в объектах Series и DataFrame. В последующих главах мы 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 читателю.

Переиндексация

Для объектов pandas критически важен метод `reindex`, т. е. возможность создания нового объекта, данные в котором *согласуются* с новым индексом. Рассмотрим простой пример:

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [80]: obj
Out[80]:
d    4.5
b    7.2
a   -5.3
c    3.6
```

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 [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [82]: obj2
Out[82]:
a    -5.3
b     7.2
c     3.6
d     4.5
e      NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[83]:
a    -5.3
b     7.2
c     3.6
d     4.5
e     0.0
```

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 [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')
Out[85]:
0     blue
1     blue
2   purple
3   purple
4  yellow
5  yellow
```

В табл. 5.4 перечислены возможные значения параметра `method`. В настоящее время интерполяцию, более сложную, чем прямое и обратное восполнение, нужно производить постфактум.

Таблица 5.4. Значение параметра `method` функции `reindex` (интерполяция)

Значение	Описание
<code>ffill</code> или <code>pad</code>	Восполнить (или перенести) значения в прямом направлении
<code>bfill</code> или <code>backfill</code>	Восполнить (или перенести) значения в обратном направлении

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 [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
...: columns=['Ohio', 'Texas', 'California'])

In [87]: frame
Out[87]:
```

```
Ohio  Texas  California
a      0      1          2
c      3      4          5
d      6      7          8
```

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

```
In [89]: fra
Ohio  Texas  California
a      0      1          2
b      NaN    NaN        NaN
c      3      4          5
d      6      7          8
```

Столбцы можно переиндексировать, задав ключевое слово `columns`:

In [90]: states = ['Texas', 'Utah', 'California']

```
In [91]: frame.reindex(columns=states)
Out[91]:
Texas  Utah  California
a      1      NaN        2
c      4      NaN        5
d      7      NaN        8
```

Строки и столбцы можно переиндексировать за одну операцию, хотя интерполяция будет применена только к строкам (к оси 0):

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
...: columns=states)
Out[92]:
Texas  Utah  California
a      1      NaN        2
b      1      NaN        2
c      4      NaN        5
d      7      NaN        8
```

Как мы скоро увидим, переиндексацию можно определить короче путем индексации меток с помощью поля `ix`:

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]
Out[93]:
Texas  Utah  California
a      1      NaN        2
b      NaN    NaN        NaN
c      4      NaN        5
d      7      NaN        8
```

Таблица 5.5. Аргументы функции `reindex`

Аргумент	Описание
index	Последовательность, которая должна стать новым индексом. Может быть экземпляром <code>Index</code> или любой другой структурой данных Python, похожей на последовательность. Экземпляр <code>Index</code> будет использован «как есть», без копирования

Аргумент	Описание
method	Метод интерполяции (восполнения), возможные значения приведены в табл. 5.4
fill_value	Значение, которой должно подставляться вместо отсутствующих значений, появляющихся в результате переиндексации
limit	При прямом или обратном восполнении максимальная длина восполняемой лакуны
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 [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [95]: new_obj = obj.drop('c')
```

```
In [96]: new_obj
```

```
Out[96]:
```

```
a    0  
b    1  
d    3  
e    4
```

```
In [97]: obj.drop(['d', 'c'])
```

```
Out[97]:
```

```
a    0  
b    1  
e    4
```

В случае DataFrame указанные в индексе значения можно удалить из любой оси:

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),  
.....:           index=['Ohio', 'Colorado', 'Utah', 'New York'],  
.....:           columns=['one', 'two', 'three', 'four'])
```

```
In [99]: data.drop(['Colorado', 'Ohio'])
```

```
Out[99]:
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
In [100]: data.drop('two', axis=1)    In [101]: data.drop(['two', 'four'], axis=1)
```

	one	three	four		one	three
Ohio	0	2	3	Ohio	0	2
Colorado	4	6	7	Colorado	4	6
Utah	8	10	11	Utah	8	10
New York	12	14	15	New York	12	14

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 любые значения из индекса объекта Series. Вот несколько примеров:

In [102]:	<code>obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])</code>		
In [103]:	<code>obj['b']</code>	In [104]:	<code>obj[1]</code>
Out [103]:	1.0	Out [104]:	1.0
In [105]:	<code>obj[2:4]</code>	In [106]:	<code>obj[['b', 'a', 'd']]</code>
Out [105]:		Out [106]:	
c	2	b	1
d	3	a	0
		d	3
In [107]:	<code>obj[[1, 3]]</code>	In [108]:	<code>obj[obj < 2]</code>
Out [107]:		Out [108]:	
b	1	a	0
d	3	b	1

Вырезание с помощью меток отличается от обычного вырезания в Python тем, что конечная точка включается:

In [109]:	<code>obj['b':'c']</code>
Out [109]:	
b	1
c	2

Установка с помощью этих методов работает ожидаемым образом:

In [110]:	<code>obj['b':'c'] = 5</code>
In [111]:	<code>obj</code>
Out [111]:	
a	0
b	5
c	5
d	3

Как мы уже видели, доступ по индексу к DataFrame применяется для извлечения одного или нескольких столбцов путем задания единственного значения или последовательности:

```
In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
....:                         index=['Ohio', 'Colorado', 'Utah', 'New York'],
....:                         columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
   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 [114]: data['two']
Out[114]:
Ohio      1
Colorado  5
Utah      9
New York 13
Name: two

In [115]: data[['three', 'one']]
Out[115]:
   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 строк с помощью вырезания или булева массива:

```
In [116]: data[:2]
Out[116]:
   one  two  three  four
Ohio     0     1     2     3
Colorado 4     5     6     7

In [117]: data[data['three'] > 5]
Out[117]:
   one  two  three  four
Colorado  4     5     6     7
Utah      8     9    10    11
New York 12    13    14    15
```

Некоторым читателям такой синтаксис может показаться непоследовательным, но он был выбран исключительно из практических соображений. Еще одна возможность – доступ по индексу с помощью булева DataFrame, например порожденного в результате скалярного сравнения:

```
In [118]: data < 5
Out[118]:
   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 [119]: data[data < 5] = 0
```

```
In [120]: data
Out[120]:
   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
```

Идея в том, чтобы сделать DataFrame синтаксически более похожим на ndarray в данном частном случае. Для доступа к строкам по индексу с помощью меток я ввел специальное индексное поле ix. Оно позволяет выбрать подмножество строк и столбцов DataFrame с применение нотации NumPy, дополненной метками осей. Как я уже говорил, это еще и более лаконичный способ выполнить переиндексацию:

```
In [121]: data.ix['Colorado', ['two', 'three']]
Out[121]:
two      5
three    6
Name: Colorado

In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
Out[122]:
        four  one  two
Colorado    7    0    5
Utah       11   8    9

In [123]: data.ix[2]
Out[123]:
one      8
two      9
three    10
four     11
Name: Utah

In [124]: data.ix[:'Utah', 'two']
Out[124]:
Ohio      0
Colorado  5
Utah     9
Name: two

In [125]: data.ix[data.three > 5, :3]
Out[125]:
        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 их них приведена в табл. 5.6. Позже мы увидим, что при работе с иерархическими индексами есть ряд дополнительных возможностей.

Таблица 5.6. Варианты доступа по индексу для объекта DataFrame

Вариант	Примечание
obj[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)
obj.ix[val]	Выбрать одну строку или подмножество строк из DataFrame
obj.ix[:, val]	Выбрать один столбец или подмножество столбцов

Вариант	Примечание
<code>obj.ix[val1, val2]</code>	Выбрать строки и столбцы
метод <code>reindex</code>	Привести одну или несколько осей в соответствие с новыми индексами
метод <code>xs</code>	Выбрать одну строку или столбец по метке и вернуть объект <code>Series</code>
методы <code>icol</code> , <code>irow</code>	Выбрать одну строку или столбец соответственно по целочисленному номеру и вернуть объект <code>Series</code>
методы <code>get_value</code> , <code>set_value</code>	Выбрать одно значение по меткам строки и столбца



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 более выразительные операции доступа с помощью индексов-меток в `ix`.

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 результатирующий индекс будет объединением индексов. Рассмотрим простой пример:

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

In [128]: s1
Out[128]:
a    7.3
c   -2.5
d    3.4
e    1.5

In [129]: s2
Out[129]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
```

Сложение этих объектов дает:

```
In [130]: s1 + s2
Out[130]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

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 [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
.....:                  index=['Ohio', 'Texas', 'Colorado'])

In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [133]: df1
Out[133]:
      b   c   d
Ohio    0   1   2
Texas   3   4   5
Colorado 6   7   8

In [134]: df2
Out[134]:
      b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11
```

При сложении этих объектов получается DataFrame, индекс и столбцы которого являются объединениями индексов и столбцов слагаемых:

```
In [135]: df1 + df2
Out[135]:
      b   c   d   e
Colorado  NaN  NaN  NaN  NaN
Ohio      3   NaN   6   NaN
Oregon   NaN  NaN  NaN  NaN
Texas     9   NaN  12   NaN
Utah     NaN  NaN  NaN  NaN
```

Восполнение значений в арифметических методах

При выполнении арифметических операций с объектами, проиндексированными по-разному, иногда желательно поместить специальное значение, например 0, в позиции операнда, которым в другом операнде соответствует отсутствующая позиция:

```
In [136]: df1=DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
In [137]: df2=DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

In [138]: df1
Out[138]:
      a   b   c   d
0    0   1   2   3
1    4   5   6   7
2    8   9  10  11

In [139]: df2
Out[139]:
      a   b   c   d   e
0    0   1   2   3   4
1    5   6   7   8   9
2   10  11  12  13  14
3   15  16  17  18  19
```

Сложение этих объектов порождает отсутствующие значения в позициях, которые имеются не в обоих операндах:

```
In [140]: df1 + df2
Out[140]:
   a    b    c    d    e
0  0    2    4    6  NaN
1  9   11   13   15  NaN
2 18   20   22   24  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 [141]: df1.add(df2, fill_value=0)
Out[141]:
   a    b    c    d    e
0  0    2    4    6    4
1  9   11   13   15    9
2 18   20   22   24   14
3 15   16   17   18   19
```

Точно так же, выполняя переиндексацию объекта Series или DataFrame, можно указать восполняемое значение:

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)
Out[142]:
   a    b    c    d    e
0  0    1    2    3    0
1  4    5    6    7    0
2  8    9   10   11    0
```

Таблица 5.7. Гибкие арифметические методы

Метод	Описание
add	Сложение (+)
sub	Вычитание (-)
div	Деление (/)
mul	Умножение (*)

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 [143]: arr = np.arange(12.).reshape((3, 4))

In [144]: arr
Out[144]:
array([[ 0.,  1.,  2.,  3.],
```

```
[ 4., 5., 6., 7.],  
[ 8., 9., 10., 11.])  
  
In [145]: arr[0]  
Out[145]: array([ 0., 1., 2., 3.])  
  
In [146]: arr - arr[0]  
Out[146]:  
array([[ 0., 0., 0., 0.],  
      [ 4., 4., 4., 4.],  
      [ 8., 8., 8., 8.]])
```

Это называется *укладыванием* и подробно объясняется в главе 12. Операции между DataFrame и Series аналогичны:

```
In [147]: frame=DataFrame(np.arange(12.).reshape((4,3)),columns=list('bde'),  
.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])  
  
In [148]: series = frame.ix[0]  
  
In [149]: frame  
Out[149]:  
      b   d   e  
Utah  0   1   2  
Ohio   3   4   5  
Texas  6   7   8  
Oregon 9  10  11  
  
In [150]: series  
Out[150]:  
      b  
Utah  0  
      d  
Ohio  1  
      e  
Texas 2  
Name: Utah
```

По умолчанию при выполнении арифметических операций между DataFrame и Series индекс Series сопоставляется со столбцами DataFrame, а укладываются строки:

```
In [151]: frame - series  
Out[151]:  
      b   d   e  
Utah  0   0   0  
Ohio   3   3   3  
Texas  6   6   6  
Oregon 9   9   9
```

Если какой-нибудь индекс не найден либо в столбцах DataFrame, либо в индексе Series, то объекты переиндексируются для образования объединения:

```
In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])  
  
In [153]: frame + series2  
Out[153]:  
      b     d     e     f  
Utah  0    NaN    3    NaN  
Ohio   3    NaN    6    NaN  
Texas  6    NaN    9    NaN  
Oregon 9    NaN   12    NaN
```

Если вы хотите вместо этого сопоставлять строки, а укладывать столбцы, то должны будете воспользоваться каким-нибудь арифметическим методом. Например:

```
In [154]: series3 = frame['d']

In [155]: frame
Out[155]:
      b   d   e
Utah  0   1   2
Ohio   3   4   5
Texas  6   7   8
Oregon 9  10  11

In [156]: series3
Out[156]:
Utah    1
Ohio    4
Texas   7
Oregon  10
Name: d

In [157]: frame.sub(series3, axis=0)
Out[157]:
      b   d   e
Utah -1   0   1
Ohio -1   0   1
Texas -1   0   1
Oregon -1   0   1
```

Передаваемый номер оси – это ось, *вдоль которой производится сопоставление*. В данном случае мы хотим сопоставлять с индексом строк DataFrame и укладывать поперек.

Применение функций и отображение

Универсальные функции NumPy (поэлементные методы массивов) отлично работают и с объектами pandas:

```
In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [159]: frame
Out[159]: Out[160]: np.abs(frame)
      b       d       e           b       d       e
Utah -0.204708 0.478943 -0.519439  Utah  0.204708 0.478943 0.519439
Ohio -0.555730 1.965781  1.393406  Ohio  0.555730 1.965781 1.393406
Texas  0.092908 0.281746  0.769023 Texas  0.092908 0.281746 0.769023
Oregon 1.246435 1.007189 -1.296221 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 [161]: f = lambda x: x.max() - x.min()

In [162]: frame.apply(f)
Out[162]:
b    1.802165
d    1.684034

In [163]: frame.apply(f, axis=1)
Out[163]:
Utah    0.998382
Ohio    2.521511
```

```
e    2.689627
Texas  0.676115
Oregon 2.542656
```

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 [164]: def f(x):
....:     return Series([x.min(), x.max()], index=['min', 'max'])

In [165]: frame.apply(f)
Out[165]:
      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 [166]: format = lambda x: '%.2f' % x

In [167]: frame.applymap(format)
Out[167]:
      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 [168]: frame['e'].map(format)
Out[168]:
Utah    -0.52
Ohio     1.39
Texas    0.77
Oregon   -1.30
Name: e
```

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 [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
```

```
a    1
b    2
c    3
d    0
```

В случае DataFrame можно сортировать по индексу, ассоцииированному с любой осью:

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
.....: columns=['d', 'a', 'b', 'c'])
```

```
In [172]: frame.sort_index()
```

```
Out[172]:
```

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

```
In [173]: frame.sort_index(axis=1)
```

```
Out[173]:
```

	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 [174]: frame.sort_index(axis=1, ascending=False)
```

```
Out[174]:
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

Для сортировки Series по значениям служит метод order:

```
In [175]: obj = Series([4, 7, -3, 2])
```

```
In [176]: obj.order()
```

```
Out[176]:
```

2	-3
3	2
0	4
1	7

Отсутствующие значения по умолчанию оказываются в конце Series:

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])
```

```
In [178]: obj.order()
```

```
Out[178]:
```

4	-3
5	2
0	4
2	7
1	NaN
3	NaN

Объект DataFrame можно сортировать по значениям в одном или нескольких столбцах. Для этого имена столбцов следует передать в качестве значения параметра by:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

```
In [180]: frame
```

```
Out[180]:
```

	a	b
0	0	4
1	1	7
2	0	-3
3	1	2

```
In [181]: frame.sort_index(by='b')
```

```
Out[181]:
```

	a	b
2	0	-3
3	1	2
0	0	4
1	1	7

Для сортировки по нескольким столбцам следует передать список имен:

```
In [182]: frame.sort_index(by=['a', 'b'])
```

```
Out[182]:
```

	a	b
2	0	-3
0	0	4
3	1	2
1	1	7

Ранжирование тесно связано с сортировкой, заключается оно в присваивании рангов – от единицы до числа присутствующих в массиве элементов. Это аналогично косвенным индексам, порождаемым методом `numpy.argsort`, с тем отличием, что существует правило обработки связанных рангов. Для ранжирования применяется метод `rank` объектов `Series` и `DataFrame`; по умолчанию `rank` обрабатывает связанные ранги, присваивая каждой группе средний ранг:

```
In [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [184]: obj.rank()
```

```
Out[184]:
```

0	6.5
1	1.0
2	6.5
3	4.5
4	3.0
5	2.0
6	4.5

Ранги можно также присваивать в соответствии с порядком появления в данных:

```
In [185]: obj.rank(method='first')
```

```
Out[185]:
```

0	6
1	1
2	7
3	4
4	3
5	2
6	5

Естественно, можно ранжировать и в порядке убывания:

```
In [186]: obj.rank(ascending=False, method='max')
Out[186]:
0      2
1      7
2      2
3      4
4      5
5      6
6      4
```

В табл. 5.8 приведен перечень способов обработки связанных рангов. DataFrame умеет вычислять ранги как по строкам, так и по столбцам:

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
.....: 'c': [-2, 5, 8, -2.5]})

In [188]: frame
Out[188]:
   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 [189]: frame.rank(axis=1)
Out[189]:
   a    b    c
0  2  3  1
1  1  3  2
2  2  1  3
3  2  3  1
```

Таблица 5.8. 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

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 [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [191]: obj
Out[191]:
a    0
a    1
b    2
b    3
c    4
```

О том, являются значения уникальными или нет, можно узнать, опросив свойство `is_unique`:

```
In [192]: obj.index.is_unique  
Out[192]: 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 [193]: obj['a']
Out[193]:
a    0
a    1
```

```
In [194]: obj['c']
Out[194]: 4
```

Такое же правило действует и для доступа к строкам в DataFrame:

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [196]: df
Out[196]:
          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 [197]: df.ix['b']
Out[197]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

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 [198]: 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 [199]: df
```

```
Out[199]:
   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 [200]: df.sum()
Out[200]:
one    9.25
two   -5.80
```

Если передать параметр `axis=1`, то суммирование будет производиться по строкам:

```
In [201]: df.sum(axis=1)
Out[201]:
a    1.40
b    2.60
c    NaN
d   -0.55
```

Отсутствующие значения исключаются, если только не является отсутствующим весь срез (в данном случае строка или столбец). Это можно подавить, задав параметр `skipna`:

```
In [202]: df.mean(axis=1, skipna=False)
Out[202]:
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
<code>axis</code>	Ось, по которой производится редуцирование. В случае <code>DataFrame</code> 0 означает строки, 1 – столбцы.
<code>skipna</code>	Исключать отсутствующие значения. По умолчанию <code>True</code>
<code>level</code>	Редуцировать с группировкой по уровням, если индекс по оси иерархический (<code>MultIndex</code>)

Некоторые методы, например `idxmin` и `idxmax`, возвращают косвенные статистики, скажем, индекс, при котором достигается минимум или максимум:

```
In [203]: df.idxmax()
Out[203]:
one    b
two    d
```

Есть также *аккумулирующие* методы:

```
In [204]: df.cumsum()
Out[204]:
      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 [205]: df.describe()
Out[205]:
      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 [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [207]: obj.describe()
Out[207]:
count     16
unique      3
top       a
freq      8
```

Полный список сводных статистик и родственных методов приведен в табл. 5.10.

Таблица 5.10. 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

Метод	Описание
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%-ый 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:

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

price = DataFrame({tic: data['Adj Close']
                  for tic, data in all_data.iteritems()})

volume = DataFrame({tic: data['Volume']
                    for tic, data in all_data.iteritems()})
```

Теперь вычислим процентные изменения цен:

```
In [209]: returns = price.pct_change()
```

```
In [210]: returns.tail()
```

Out[210]:

Date	AAPL	GOOG	IBM	MSFT
2009-12-24	0.034339	0.011117	0.004420	0.002747
2009-12-28	0.012294	0.007098	0.013282	0.005479
2009-12-29	-0.011861	-0.005571	-0.003474	0.006812
2009-12-30	0.012147	0.005376	0.005468	-0.013532
2009-12-31	-0.004300	-0.004416	-0.012609	-0.015432

Метод corr объекта Series вычисляет корреляцию перекрывающихся, отличных от NA, выровненных по индексу значений в двух объектах Series. Соответственно, метод cov вычисляет ковариацию:

```
In [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838
```

```
In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

С другой стороны, методы corr и cov объекта DataFrame возвращают соответственно полную корреляционную или ковариационную матрицу в виде DataFrame:

```
In [213]: returns.corr()
Out[213]:
          AAPL      GOOG      IBM      MSFT
AAPL    1.000000  0.470660  0.410648  0.424550
GOOG    0.470660  1.000000  0.390692  0.443334
IBM     0.410648  0.390692  1.000000  0.496093
MSFT    0.424550  0.443334  0.496093  1.000000
```

```
In [214]: returns.cov()
Out[214]:
          AAPL      GOOG      IBM      MSFT
AAPL    0.001028  0.000303  0.000252  0.000309
GOOG    0.000303  0.000580  0.000142  0.000205
IBM     0.000252  0.000142  0.000367  0.000216
MSFT    0.000309  0.000205  0.000216  0.000516
```

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 [215]: returns.corrwith(returns.IBM)
Out[215]:
AAPL    0.410648
GOOG    0.390692
IBM     1.000000
MSFT    0.496093
```

Если передать объект DataFrame, то будут вычислены корреляции столбцов с соответственными именами. Ниже я вычисляю корреляции процентных изменений с объемом сделок:

```
In [216]: returns.corrwith(volume)
Out[216]:
AAPL    -0.057461
GOOG     0.062644
IBM     -0.007900
MSFT    -0.014175
```

Если передать `axis=1`, 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 [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'b', 'c', 'c'])
```

Метод `unique` возвращает массив уникальных значений в Series:

```
In [218]: uniques = obj.unique()
```

```
In [219]: uniques
Out[219]: 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 [220]: obj.value_counts()
Out[220]:
c    3
a    3
b    2
d    1
```

Для удобства этот объект отсортирован по значениям в порядке убывания. Функция `value_counts` может быть также вызвана как метод pandas верхнего уровня и в таком случае применима к любому массиву или последовательности:

```
In [221]: pd.value_counts(obj.values, sort=False)
Out[221]:
a    3
b    2
c    3
d    1
```

Наконец, 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 [222]: mask = obj.isin(['b', 'c'])

In [223]: mask
Out[223]:
0    True
1   False
2   False
3   False
4   False
5    True
6    True
7    True
8    True

In [224]: obj[mask]
Out[224]:
0    c
5    b
6    b
7    c
8    c
```

Справочная информация по этим методам приведена в табл. 5.11.

Таблица 5.11. Уникальные значения, счетчики значений и методы «раскладывания»

Метод	Описание
isin	Вычисляет булев массив, показывающий, содержится ли каждое принадлежащее Series значение в переданной последовательности
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. Отсортирован в порядке убывания частот

Иногда требуется вычислить гистограмму нескольких взаимосвязанных столбцов в DataFrame. Приведем пример:

```
In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....: 'Qu2': [2, 3, 1, 2, 3],
.....: 'Qu3': [1, 5, 2, 4, 4]})

In [226]: data
Out[226]:
   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 [227]: result = data.apply(pd.value_counts).fillna(0)

In [228]: result
Out[228]:
   Qu1  Qu2  Qu3
1     1     1     1
2     0     2     1
3     2     2     0
```

4	2	0	2
5	0	0	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, выше мы видели, что при вычислении всех описательных статистик для объектов pandas отсутствующие данные не учитываются.

В pandas для представления отсутствующих данных в любых массивах – как чисел с плавающей точкой, так и иных – используется значение с плавающей точкой `NaN` (не число). Это просто *признак*, который легко распознать:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [230]: string_data
Out[230]:
0    aardvark
1    artichoke
2      NaN
3    avocado

In [231]: string_data.isnull()
Out[231]:
0    False
1    False
2     True
3    False
```

Встроено в Python значение `None` также рассматривается как отсутствующее в массивах объектов:

```
In [232]: string_data[0] = None

In [233]: string_data.isnull()
Out[233]:
0    True
1   False
2    True
3   False
```

Я не утверждаю, что представление отсутствующих значений в pandas оптимально, но оно простое и в разумной степени последовательное. Принимая во внимание характеристики производительности и простой API, это наилучшее решение, которое я смог придумать в отсутствие истинного типа данных `NA` или выделенной комбинации бит среди типов данных NumPy. Поскольку разработка NumPy продолжается, в будущем ситуация может измениться.

Таблица 5.12. Методы обработки отсутствующих данных

Аргумент	Описание
<code>dropna</code>	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

Аргумент	Описание
fillna	Восполняет отсутствующие данные указанным значением или использует какой-нибудь метод интерполяции, например 'ffill' или 'bfill'
isnull	Возвращает объект, содержащий булевые значения, которые показывают, какие значения отсутствуют
notnull	Логическое отрицание isnull

Фильтрация отсутствующих данных

Существует ряд средств для фильтрации отсутствующих данных. Конечно, можно сделать это вручную, но часто бывает полезен метод dropna. Для Series он возвращает другой объект Series, содержащий только данные и значения индекса, отличные от NA:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
0    1.0
2    3.5
4    7.0
```

Естественно, это можно было бы вычислить и самостоятельно с помощью булевой индексации:

```
In [237]: data[data.notnull()]
Out[237]:
0    1.0
2    3.5
4    7.0
```

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 [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
.....: [NA, NA, NA], [NA, 6.5, 3.]))

In [239]: cleaned = data.dropna()

In [240]: data
Out[240]:
   0    1    2
0  1  6.5  3
1  1  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3

In [241]: cleaned
Out[241]:
   0    1    2
0  1  6.5  3
```

Если передать параметр `how='all'`, то будут отброшены строки, которые целиком состоят из отсутствующих значений:

```
In [242]: data.dropna(how='all')
Out[242]:
   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 [243]: data[4] = NA
```

<pre>In [244]: data Out[244]: 0 1 2 4 0 1 6.5 3 NaN 1 1 NaN NaN NaN 2 NaN NaN NaN NaN 3 NaN 6.5 3 NaN</pre>	<pre>In [245]: data.dropna(axis=1, how='all') Out[245]: 0 1 2 0 1 6.5 3 1 1 NaN NaN 2 NaN NaN NaN 3 NaN 6.5 3</pre>
--	---

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 [246]: df = DataFrame(np.random.randn(7, 3))
```

```
In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```

<pre>In [248]: df Out[248]: 0 1 2 0 -0.577087 NaN NaN 1 0.523772 NaN NaN 2 -0.713544 NaN NaN 3 -1.860761 NaN 0.560145 4 -1.265934 NaN -1.063512 5 0.332883 -2.359419 -0.199543 6 -1.541996 -0.970736 -1.307030</pre>	<pre>In [249]: df.dropna(thresh=3) Out[249]: 0 1 2 5 0.332883 -2.359419 -0.199543 6 -1.541996 -0.970736 -1.307030</pre>
--	---

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 [250]: df.fillna(0)
Out[250]:
```

```

        0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
    
```

Если передать методу `fillna` словарь, то можно будет подставлять вместо отсутствующих данных значение, зависящее от столбца:

```

In [251]: df.fillna({1: 0.5, 3: -1})
Out[251]:
        0         1         2
0 -0.577087  0.500000  NaN
1  0.523772  0.500000  NaN
2 -0.713544  0.500000  NaN
3 -1.860761  0.500000  0.560145
4 -1.265934  0.500000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
    
```

Метод `fillna` возвращает новый объект, но можно также модифицировать существующий объект на месте:

```

# всегда возвращает ссылку на заполненный объект
In [252]: _ = df.fillna(0, inplace=True)

In [253]: df
Out[253]:
        0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
    
```

Те же методы интерполяции, что применяются для переиндексации, годятся и для `fillna`:

```

In [254]: df = DataFrame(np.random.randn(6, 3))

In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA

In [256]: df
Out[256]:
   0  1  2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674      NaN  1.004812
3  1.327195      NaN -1.549106
4  0.022185      NaN      NaN
    
```

```
5    0.862580      NaN      NaN
```

```
In [257]: df.fillna(method='ffill')
```

```
Out[257]:
```

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	1.349742	1.004812
3	1.327195	1.349742	-1.549106
4	0.022185	1.349742	-1.549106
5	0.862580	1.349742	-1.549106

```
In [258]: df.fillna(method='ffill', limit=2)
```

```
Out[258]:
```

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	1.349742	1.004812
3	1.327195	1.349742	-1.549106
4	0.022185	NaN	-1.549106
5	0.862580	NaN	-1.549106

При некоторой изобретательности можно использовать `fillna` и другими способами. Например, можно передать среднее или медиану объекта Series:

```
In [259]: data = Series([1., NA, 3.5, NA, 7])
```

```
In [260]: data.fillna(data.mean())
```

```
Out[260]:
```

0	1.000000
1	3.833333
2	3.500000
3	3.833333
4	7.000000

Справочная информация о методе `fillna` приведена в табл. 5.13.

Таблица 5.13. Аргументы метода `fillna`

Аргумент	Описание
<code>value</code>	Скалярное значение или похожий на словарь объект для восполнения отсутствующих значений
<code>method</code>	Метод интерполяции. По умолчанию, если не задано других аргументов, предполагается метод 'ffill'
<code>axis</code>	Ось, по которой производится восполнение. По умолчанию <code>axis=0</code>
<code>inplace</code>	Модифицировать исходный объект, не создавая копию
<code>limit</code>	Для прямого и обратного восполнения максимальное количество непрерывных заполняемых промежутков

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 [261]: data = Series(np.random.randn(10),
.....: index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
```

```
.....: [1, 2, 3, 1, 2, 3, 1, 2, 2, 3])  
  
In [262]: data  
Out[262]:  
a 1 0.670216  
2 0.852965  
3 -0.955869  
b 1 -0.023493  
2 -2.304234  
3 -0.652469  
c 1 -1.218302  
2 -1.332610  
d 2 1.074623  
3 0.723642
```

Здесь мы видим отформатированное представление Series с мультииндексом (MultiIndex). «Разрывы» в представлении индекса означают «взять значение вышестоящей метки».

```
In [263]: data.index  
Out[263]:  
MultiIndex  
[('a', 1) ('a', 2) ('a', 3) ('b', 1) ('b', 2) ('b', 3) ('c', 1)  
 ('c', 2) ('d', 2) ('d', 3)]
```

Для иерархически индексированного объекта возможен доступ по так называемому *частичному* индексу, что позволяет лаконично записывать выборку подмножества данных:

```
In [264]: data['b']  
Out[264]:  
1 -0.023493  
2 -2.304234  
3 -0.652469  
  
In [265]: data['b':'c']  
Out[265]:  
b 1 -0.023493  
2 -2.304234  
3 -0.652469  
c 1 -1.218302  
2 -1.332610  
  
In [266]: data.ix[['b', 'd']]  
Out[266]:  
b 1 -0.023493  
2 -2.304234  
3 -0.652469  
d 2 1.074623  
3 0.723642
```

В некоторых случаях возможна даже выборка с «внутреннего» уровня:

```
In [267]: data[:, 2]  
Out[267]:  
a 0.852965  
b -2.304234  
c -1.332610  
d 1.074623
```

Иерархическое индексирование играет важнейшую роль в изменении формы данных и групповых операциях, в том числе построении сводных таблиц. Напри-

мер, эти данные можно было бы преобразовать в DataFrame с помощью метода `unstack`:

```
In [268]: data.unstack()
Out[268]:
1 2 3
a  0.670216  0.852965 -0.955869
b -0.023493 -2.304234 -0.652469
c -1.218302 -1.332610      NaN
d      NaN    1.074623  0.723642
```

Обратной к `unstack` операцией является `stack`:

```
In [269]: data.unstack().stack()
Out[269]:
a 1  0.670216
   2  0.852965
   3 -0.955869
b 1 -0.023493
   2 -2.304234
   3 -0.652469
c 1 -1.218302
   2 -1.332610
d 2  1.074623
   3  0.723642
```

Методы `stack` и `unstack` будут подробно рассмотрены в главе 7.

В случае DataFrame иерархический индекс можно построить по любой оси:

```
In [270]: frame = DataFrame(np.arange(12).reshape((4, 3)),
.....:                               index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
.....:                               columns=[['Ohio', 'Ohio', 'Colorado'],
.....:                                     ['Green', 'Red', 'Green']])
In [271]: frame
Out[271]:
          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 осиях!):

```
In [272]: frame.index.names = ['key1', 'key2']

In [273]: frame.columns.names = ['state', 'color']

In [274]: frame
Out[274]:
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
    
```

Доступ по частичному индексу, как и раньше, позволяет выбирать группы столбцов:

```

In [275]: frame['Ohio']
Out[275]:
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:

```
tiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                    names=['state', 'color'])
```

Уровни переупорядочения и сортировки

Иногда требуется изменить порядок уровней на оси или отсортировать данные по значениям на одной 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 [276]: frame.swaplevel('key1', 'key2')
Out[276]:
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 `sortlevel` выполняет устойчивую сортировку данных, используя только значения на одном уровне. После перестановки уровней обычно вызывают также `sortlevel`, чтобы лексикографически отсортировать результат:

<pre>In [277]: frame.sortlevel(1) Out[277]:</pre>	<pre>In [278]: frame.swaplevel(0, 1).sortlevel(0) Out[278]:</pre>
state Ohio	state Ohio
Colorado	Colorado

color	Green	Red		Green	color	Green	Red		Green
key1	key2				key2	key1			
a	1	0	1		2	1	a	0	1
b	1	6	7		8		b	6	7
a	2	3	4		5	2	a	3	4
b	2	9	10		11		b	9	10



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_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 суммирование по конкретной оси. Рассмотрим тот же объект DataFrame, что и выше; мы можем суммировать по уровню для строк или для столбцов:

```
In [279]: frame.sum(level='key2')
Out[279]:
state      Ohio          Colorado
color    Green     Red        Green
key2
1            6      8       10
2           12     14       16
```

```
In [280]: frame.sum(level='color', axis=1)
Out[280]:
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.

Работа со столбцами 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 [281]: 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 [282]: frame
```

```
Out[282]:  
     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 [283]: frame2 = frame.set_index(['c', 'd'])
```

```
In [284]: frame2
```

```
Out[284]:
```

```
     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 [285]: frame.set_index(['c', 'd'], drop=False)
```

```
Out[285]:
```

```
     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 [286]: frame2.reset_index()
```

```
Out[286]:
```

```
     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
```

Другие возможности pandas

Ниже перечислено еще несколько возможностей, которые могут пригодиться вам в экспериментах с данными.

Доступ по целочисленному индексу

При работе с объектами pandas, проиндексированными целыми числами, начинающие часто попадают в ловушку из-за некоторых различий с семантикой доступа по индексу к встроенным в Python структурам данных, в частности, спискам и кортежам. Например, вряд ли вы ожидаете ошибки в таком коде:

```
ser = Series(np.arange(3.))
ser[-1]
```

В данном случае pandas могла бы прибегнуть к целочисленному индексированию, но я не знаю никакого общего и безопасного способа сделать это, не внося тонких ошибок. Здесь мы имеем индекс, содержащий значения 0, 1, 2, но автоматически понять, чего хочет пользователь (индекс по метке или по позиции) трудно:

```
In [288]: ser
Out[288]:
0    0
1    1
2    2
```

С другой стороны, когда индекс не является целым числом, неоднозначности не возникает:

```
In [289]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])
In [290]: ser2[-1]
Out[290]: 2.0
```

Чтобы не оставлять места разноречивым интерпретациям, принято решение: если индекс по оси содержит индексаторы, то доступ к данным по целочисленному индексу всегда трактуется как доступ по метке. Это относится и к полю `ix`:

```
In [291]: ser.ix[:1]
Out[291]:
0    0
1    1
```

В случае, когда требуется надежный доступ по номеру позиции вне зависимости от типа индекса, можно использовать метод `iget_value` объекта `Series` или методы `irow` и `icol` объекта `DataFrame`:

```
In [292]: ser3 = Series(range(3), index=[-5, 1, 3])
In [293]: ser3.iget_value(2)
```

```
Out[293]: 2

In [294]: frame = DataFrame(np.arange(6).reshape(3, 2)), index=[2, 0, 1])

In [295]: frame.irow(0)
Out[295]:
0    0
1    1
Name: 2
```

Структура данных Panel

Хотя структура данных Panel и не является основной темой этой книги, она существует в pandas и может рассматриваться как трехмерный аналог DataFrame. При разработке pandas основное внимание уделялось манипуляциям с табличными данными, поскольку о них проще рассуждать, а наличие иерархических индексов в большинстве случаев позволяет обойтись без настоящих N-мерных массивов.

Для создания Panel используется словарь объектов DataFrame или трехмерный массив ndarray:

```
import pandas.io.data as web

pdata = pd.Panel(dict((stk, web.get_data_yahoo(stk, '1/1/2009', '6/1/2012'))
                      for stk in ['AAPL', 'GOOG', 'MSFT', 'DELL']))
```

Каждый элемент Panel (аналог столбца в DataFrame) является объектом DataFrame:

```
In [297]: pdata
Out[297]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 861 (major) x 6 (minor)
Items: AAPL to MSFT
Major axis: 2009-01-02 00:00:00 to 2012-06-01 00:00:00
Minor axis: Open to Adj Close

In [298]: pdata = pdata.swapaxes('items', 'minor')

In [299]: pdata['Adj Close']
Out[299]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 861 entries, 2009-01-02 00:00:00 to 2012-06-01 00:00:00
Data columns:
AAPL    861    non-null values
DELL    861    non-null values
GOOG    861    non-null values
MSFT    861    non-null values
dtypes: float64(4)
```

Основанное на поле ix индексирование по меткам обобщается на три измерения, поэтому мы можем выбрать все данные за конкретную дату или диапазон дат следующим образом:

```
In [300]: pdata.ix[:, '6/1/2012', :]
Out[300]:
Open      High      Low     Close   Volume       Adj     Close
AAPL    569.16  572.65  560.52  560.99  18606700  560.99
DELL     12.15   12.30   12.05   12.07  19396700  12.07
GOOG    571.79  572.65  568.35  570.98  3057900   570.98
MSFT     28.76   28.96   28.44   28.45  56634300  28.45

In [301]: pdata.ix['Adj Close', '5/22/2012':, :]
Out[301]:
          AAPL     DELL     GOOG     MSFT
Date
2012-05-22  556.97  15.08  600.80  29.76
2012-05-23  570.56  12.49  609.46  29.11
2012-05-24  565.32  12.45  603.66  29.07
2012-05-25  562.29  12.46  591.53  29.06
2012-05-29  572.27  12.66  594.34  29.56
2012-05-30  579.17  12.56  588.23  29.34
2012-05-31  577.73  12.33  580.86  29.19
2012-06-01  560.99  12.07  570.98  28.45
```

Альтернативный способ представления панельных данных, особенно удобный для подгонки статистических моделей, – в виде «стопки» объектов DataFrame:

```
In [302]: stacked = pdata.ix[:, '5/30/2012':, :].to_frame()

In [303]: stacked
Out[303]:
Open High Low Close Volume Adj Close
major minor
2012-05-30 AAPL  569.20  579.99  566.56  579.17  18908200  579.17
              DELL  12.59   12.70   12.46   12.56  19787800  12.56
              GOOG  588.16  591.90  583.53  588.23  1906700   588.23
              MSFT  29.35  29.48   29.12   29.34  41585500  29.34
2012-05-31 AAPL  580.74  581.50  571.46  577.73  17559800  577.73
              DELL  12.53  12.54   12.33   12.33  19955500  12.33
              GOOG  588.72  590.00  579.00  580.86  2968300   580.86
              MSFT  29.30  29.42   28.94   29.19  39134000  29.19
2012-06-01 AAPL  569.16  572.65  560.52  560.99  18606700  560.99
              DELL  12.15  12.30   12.05   12.07  19396700  12.07
              GOOG  571.79  572.65  568.35  570.98  3057900   570.98
              MSFT  28.76  28.96   28.44   28.45  56634300  28.45
```

У объекта DataFrame есть метод `to_panel` – обращение `to_frame`:

```
In [304]: stacked.to_panel()
Out[304]:
<class 'pandas.core.panel.Panel'>
Dimensions: 6 (items) x 3 (major) x 4 (minor)
Items: Open to Adj Close
Major axis: 2012-05-30 00:00:00 to 2012-06-01 00:00:00
Minor axis: AAPL to MSFT
```

ГЛАВА 6.

Чтение и запись данных, форматы файлов

Описанные в этой книге инструменты были бы бесполезны, если бы в программе на Python не было возможности легко импортировать и экспортировать данные. Я буду рассматривать в основном ввод и вывод с помощью объектов pandas, хотя, разумеется, в других библиотеках нет недостатка в соответствующих средствах. Например, в NumPy имеются низкоуровневые, но очень быстрые функции для загрузки и сохранения данных, включая и поддержку файлов, спроектированных на память. Подробнее об этом см. главу 12.

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 веб.

Чтение и запись данных в текстовом формате

Python превратился в излюбленный язык манипулирования текстом и файлами благодаря простому синтаксису взаимодействия с файлами, интуитивно понятным структурам данных и таким удобным средствам, как упаковка и распаковка кортежей.

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`.

Таблица 6.1. Функции чтения в pandas

Функция	Описание
<code>read_csv</code>	Загружает данные с разделителями из файла, URL-адреса или похожего на файл объекта. По умолчанию разделителем является запятая
<code>read_table</code>	Загружает данные с разделителями из файла, URL-адреса или похожего на файл объекта. По умолчанию разделителем является символ табуляции ('\\t')

Функция	Описание
read_fwf	Читает данные в формате с фиксированной шириной столбцов (без разделителей)
read_clipboard	Вариант <code>read_table</code> , который читает данные из буфера обмена. Полезно для преобразования в таблицу данных на веб-странице

Я дам краткий обзор этих функций, которые служат для преобразования текстовых данных в объект `DataFrame`. Их 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.

Выведение типа – одна из самых важных черт этих функций; это означает, что пользователю необязательно явно задавать, содержат столбцы данные с плавающей точкой, целочисленные, булевые или строковые. Правда, для обработки дат и нестандартных типов требуется больше усилий. Начнем с текстового файла, содержащего короткий список данных через запятую (формат CSV):

```
In [846]: !cat ch06/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Поскольку данные разделены запятыми, мы можем прочитать их в `DataFrame` с помощью функции `read_csv`:

```
In [847]: df = pd.read_csv('ch06/ex1.csv')
```

```
In [848]: df
Out[848]:
   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 [849]: pd.read_table('ch06/ex1.csv', sep=',')
Out[849]:
   a   b   c   d      message
0  1   2   3   4      hello
1  5   6   7   8     world
2  9  10  11  12      foo
```



Я здесь пользуюсь 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`.

В файле не всегда есть строка-заголовок. Рассмотрим такой файл:

```
In [850]: !cat ch06/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 выбор имен столбцов по умолчанию, а можно задать их самостоятельно:

```
In [851]: pd.read_csv('ch06/ex2.csv', header=None)
Out[851]:
   X.1   X.2   X.3   X.4   X.5
0     1     2     3     4  hello
1     5     6     7     8  world
2     9    10    11    12    foo
```

```
In [852]: pd.read_csv('ch06/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[852]:
   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 [853]: names = ['a', 'b', 'c', 'd', 'message']
```

```
In [854]: pd.read_csv('ch06/ex2.csv', names=names, index_col='message')
Out[854]:
      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 [855]: !cat ch06/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 [856]: parsed = pd.read_csv('ch06/csv_mindex.csv', index_col=['key1', 'key2'])
In [857]: parsed
Out[857]:
      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 [858]: list(open('ch06/ex3.txt'))
Out[858]:
['          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 регулярное выражение `\s+`:

```
In [859]: result = pd.read_table('ch06/ex3.txt', sep='\s+')
In [860]: result
Out[860]:
      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 число 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 строку файла:

```
In [861]: !cat ch06/ex4.csv
# привет!
a,b,c,d,message
# хотелось немного усложнить тебе жизнь
# а нечего читать CSV-файла на компьютере
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [862]: pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])
Out[862]:
   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 [863]: !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 [864]: result = pd.read_csv('ch06/ex5.csv')

In [865]: result
Out[865]:
   something   a   b   c   d   message
0        one   1   2   3   4      NaN
1       two   5   6  NaN   8      world
2     three   9  10  11  12      foo

In [866]: pd.isnull(result)
Out[866]:
   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 [867]: result = pd.read_csv('ch06/ex5.csv', na_values=['NULL'])
```

```
In [868]: result
```

```
Out[868]:
   something    a    b    c    d    message
0      one     1     2     3     4      NaN
1      two     5     6    NaN     8    world
2    three     9    10    11    12      foo
```

Если в разных столбцах применяются разные маркеры, то их можно задать с помощью словаря:

```
In [869]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}
```

```
In [870]: pd.read_csv('ch06/ex5.csv', na_values=sentinels)
```

```
Out[870]:
   something    a    b    c    d    message
0      one     1     2     3     4      NaN
1      NaN     5     6    NaN     8    world
2    three     9    10    11    12      NaN
```

Таблица 6.2. Аргументы функций `read_csv` и `read_table`

Аргумент	Описание
<code>path</code>	Строка, обозначающая путь в файловой системе, URL-адрес или похожий на файл объект
<code>sep</code> или <code>delimiter</code>	Последовательность символов или регулярное выражение, служащее для разделения полей в строке
<code>header</code>	Номер строки, содержащей имена столбцов. По умолчанию равен 0 (первая строка). Если строки-заголовка нет, должен быть равен <code>None</code>
<code>index_col</code>	Номера или имена столбцов, трактуемых как индекс строк в результирующем объекте. Может быть задан один номер (имя) или список номеров (имен), определяющий иерархический индекс
<code>names</code>	Список имен столбцов результирующего объекта, задается, если <code>header=None</code>
<code>skiprows</code>	Количество игнорируемых начальных строк или список номеров игнорируемых строк (нумерация начинается с 0)
<code>na_values</code>	Последовательность значений, интерпретируемых как маркеры отсутствующих данных
<code>comment</code>	Один или несколько символов, начинающих комментарий, который продолжается до конца строки
<code>parse_dates</code>	Пытаться разобрать данные как дату и время; по умолчанию <code>False</code> . Если равен <code>True</code> , 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)
<code>keep_date_col</code>	В случае, когда для разбора данных столбцы объединяются, следует ли отбрасывать объединенные столбцы. По умолчанию <code>True</code>

Аргумент	Описание
converters	Словарь, содержащий отображение номеров или имен столбцов на функции. Например, {'foo': f} означает, что нужно применить функцию f ко всем значением в столбце foo
dayfirst	При разборе потенциально неоднозначных дат предполагать международный формат (т. е. 7/6/2012 означает «7 июня 2012»). 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 .

```
In [871]: result = pd.read_csv('ch06/ex6.csv')

In [872]: result
Out[872]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns:
one      10000 non-null values
two      10000 non-null values
three    10000 non-null values
four     10000 non-null values
key      10000 non-null values
dtypes: float64(4), object(1)
```

Чтобы прочитать только небольшое число строк (а не весь файл), нужно задать это число в параметре nrows:

```
In [873]: pd.read_csv('ch06/ex6.csv', nrows=5)
Out[873]:
```

```

      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 порций в строках:

```
In [874]: chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)
```

```
In [875]: chunker
```

```
Out[875]: <pandas.io.parsers.TextParser at 0x8398150>
```

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('ch06/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 [877]: tot[:10]
```

```
Out[877]:
```

E	368
X	364
L	346
O	343
Q	340
M	338
J	337
F	335
K	334
H	330

У объекта `TextParser` имеется также метод `get_chunk`, который позволяет читать куски произвольного размера.

Вывод данных в текстовом формате

Данные можно экспортить в формате с разделителями. Рассмотрим одну из приведенных выше операций чтения CSV-файла:

```
In [878]: data = pd.read_csv('ch06/ex5.csv')
```

```
In [879]: data
```

```
Out[879]:
   something    a    b    c    d    message
0      one     1    2    3    4      NaN
1      two     5    6  NaN    8    world
2    three     9   10   11   12      foo
```

С помощью метода `to_csv` объекта `DataFrame` мы можем вывести данные в файл через запятую:

```
In [880]: data.to_csv('ch06/out.csv')
```

```
In [881]: !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 [882]: 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 [883]: 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 [884]: 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 [885]: 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 [886]: dates = pd.date_range('1/1/2000', periods=7)
```

```
In [887]: ts = Series(np.arange(7), index=dates)
```

```
In [888]: ts.to_csv('ch06/tseries.csv')
```

```
In [889]: !cat ch06/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
```

Подходящим образом задав параметры (заголовок отсутствует, первый столбец считается индексом), CSV-представление объекта Series можно прочитать методом `read_csv`, но существует вспомогательный метод `from_csv`, который позволяет сделать это немного проще:

```
In [890]: Series.from_csv('ch06/tseries.csv', parse_dates=True)
```

```
Out[890]:
```

```
2000-01-01    0  
2000-01-02    1  
2000-01-03    2  
2000-01-04    3  
2000-01-05    4  
2000-01-06    5  
2000-01-07    6
```

Дополнительные сведения о методах `to_csv` и `from_csv` смотрите в строках документации в IPython.

Ручная 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 ch06/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('ch06/ex7.csv')

reader = csv.reader(f)
```

Итерирование файла с помощью объекта reader дает кортежи значений в каждой строке после удаления кавычек:

```
In [893]: 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. Например:

```
In [894]: lines = list(csv.reader(open('ch06/ex7.csv')))

In [895]: header, values = lines[0], lines[1:]

In [896]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [897]: data_dict
Out[897]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

Встречаются различные вариации CSV-файлов. Для определения 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 = '"'

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

Аргумент	Описание
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": 25, "pet": "Zuko"}, {"name": "Katie", "age": 33, "pet": "Cisco"}]
```

```
}\n\n\n
```

Данные в формате JSON очень напоминают код на Python с тем отличием, что отсутствующее значение обозначается `null`, и еще некоторыми нюансами (например, запрещается ставить запятую после последнего элемента списка). Базовыми типами являются объекты (словари), массивы (списки), строки, числа, булевые 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 [899]: import json\n\nIn [900]: result = json.loads(obj)\n\nIn [901]: result\nOut[901]:\n{u'name': u'Wes',\n u'pet': None,\n u'places_lived': [u'United States', u'Spain', u'Germany'],\n u'siblings': [{u'age': 25, u'name': u'Scott', u'pet': u'Zuko'},\n               {u'age': 33, u'name': u'Katie', u'pet': u'Cisco'}]}
```

Напротив, метод `json.dumps` преобразует объект Python в формат JSON:

```
In [902]: 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 объектов JSON конструктору DataFrame и выбрать подмножество полей данных:

```
In [903]: siblings = DataFrame(result['siblings'], columns=['name', 'age'])\n\nIn [904]: siblings\nOut[904]:\n   name  age\n0  Scott   25\n1  Katie   33
```

Более полный пример чтения и манипулирования данными в формате JSON (включая и вложенные записи) приведен при рассмотрении базы данных о продуктах питания USDA в следующей главе.



Сейчас идет работа по добавлению в pandas быстрых написанных на C средств для экспорта в формате JSON (`to_json`) и декодирования данных в этом формате (`from_json`). На момент написания этой книги они еще не были готовы.

XML и HTML: разбор веб-страниц

На Python написано много библиотек для чтения и записи данных в вездесущих форматах HTML и XML. В частности, библиотека lxml (<http://lxml.de>) известна высокой производительностью при разборе очень больших файлов. Для lxml имеется несколько программных интерфейсов; сначала я продемонстрирую интерфейс lxml.html для работы с HTML, а затем разберу XML-документ с помощью lxml.objectify.

Многие сайты показывают данные в виде HTML-таблиц, удобных для просмотра в браузере, но не предлагают их в таких машиночитаемых форматах, как JSON или XML. Так, например, обстоит дело с данными об биржевых опционах на сайте Yahoo! Finance. Для тех, кто не в курсе, скажу, что опцион – это производный финансовый инструмент (дериватив), который дает право покупать (опцион на покупку, или *колл-опцион*) или продавать (опцион на продажу, или *пут-опцион*) акции компании по некоторой цене (*цене исполнения*) в промежутке времени между текущим моментом и некоторым фиксированным моментом в будущем (*конечной датой*). Колл- и пут-опционы торгуются с разными ценами исполнения и конечными датами; эти данные можно найти в таблицах на сайте Yahoo! Finance.

Для начала решите, с какого URL-адреса вы хотите загружать данные, затем откройте его с помощью средств из библиотеки urllib2 и разберите поток, пользуясь lxml:

```
from lxml.html import parse
from urllib2 import urlopen

parsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL+Options'))

doc = parsed.getroot()
```

Имея этот объект, мы можем выбрать все HTML-теги указанного типа, например, теги table, внутри которых находятся интересующие нас данные. Для примера получим список всех гиперссылок в документе, они представляются в HTML тегом a. Вызовем метод.findall корневого элемента документа, передав ему выражение XPath (это язык, на котором записываются «запросы» к документу):

```
In [906]: links = doc.findall('.//a')

In [907]: links[15:20]
Out[907]:
[<Element a at 0x6c488f0>,
 <Element a at 0x6c48950>,
 <Element a at 0x6c489b0>,
 <Element a at 0x6c48a10>,
 <Element a at 0x6c48a70>]
```

Но это объекты, представляющие HTML-элементы; чтобы получить URL и текст ссылки, нам нужно воспользоваться методом get элемента (для получения URL) или методом text_content (для получения текста):

```
In [908]: lnk = links[28]

In [909]: lnk
Out[909]: <Element a at 0x6c48dd0>

In [910]: lnk.get('href')
Out[910]: 'http://biz.yahoo.com/special.html'

In [911]: lnk.text_content()
Out[911]: 'Special Editions'
```

Таким образом, получение всех гиперссылок в документе сводится к списковому включению:

```
In [912]: urls = [lnk.get('href') for lnk in doc.findall('.//a')]

In [913]: urls[-10:]
Out[913]:
['http://info.yahoo.com/privacy/us/yahoo/finance/details.html',
 'http://info.yahoo.com/relevantads/',
 'http://docs.yahoo.com/info/terms/',
 'http://docs.yahoo.com/info/copyright/copyright.html',
 'http://help.yahoo.com/l/us/yahoo/finance/forms_index.html',
 'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
 'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
 'http://www.capitaliq.com',
 'http://www.csidata.com',
 'http://www.morningstar.com/']
```

Что касается отыскания нужных таблиц в документе, то это делается методом проб и ошибок; на некоторых сайтах решение этой задачи упрощается, потому что таблица имеет атрибут id. Я нашел, какие таблицы содержат данные о колл- и пут-опциях:

```
tables = doc.findall('.//table')
calls = tables[9]
puts = tables[13]
```

В каждой таблице имеется строка-заголовок, а за ней идут строки с данными:

```
In [915]: rows = calls.findall('.//tr')
```

Для всех строк, включая заголовок, мы хотим извлечь текст из каждой ячейки; в случае заголовка ячейками являются элементы th, а для строк данных – элементы td:

```
def _unpack(row, kind='td'):
    elts = row.findall('.//%s' % kind)
    return [val.text_content() for val in elts]
```

Таким образом, получаем:

```
In [917]: _unpack(rows[0], kind='th')
Out[917]: ['Strike', 'Symbol', 'Last', 'Chg', 'Bid', 'Ask', 'Vol', 'Open Int']

In [918]: _unpack(rows[1], kind='td')
```

```
Out[918]:
['295.00',
 'AAPL120818C00295000',
 '310.40',
 ' 0.00',
 '289.80',
 '290.80',
 '1',
 '169']
```

Теперь для преобразования данных в объект DataFrame осталось объединить все описанные шаги вместе. Поскольку числовые данные по-прежнему записаны в виде строк, возможно, потребуется преобразовать некоторые, но не все столбцы в формат с плавающей точкой. Это можно сделать и вручную, но, по счастью, в библиотеке pandas есть класс `TextParser`, который используется функцией `read_csv` и другими функциями разбора для автоматического преобразования типов:

```
from pandas.io.parsers import TextParser

def parse_options_data(table):
    rows = table.findall('.//tr')
    header = _unpack(rows[0], kind='th')
    data = [_unpack(r) for r in rows[1:]]
    return TextParser(data, names=header).get_chunk()
```

Наконец, вызываем эту функцию разбора для табличных объектов `lxml` и получаем результат в виде DataFrame:

```
In [920]: call_data = parse_options_data(calls)
```

```
In [921]: put_data = parse_options_data(puts)
```

```
In [922]: call_data[:10]
```

```
Out[922]:
```

	Strike	Symbol	Last	Chg	Bid	Ask	Vol	Open	Int
0	295	AAPL120818C00295000	310.40	0.0	289.80	290.80	1	169	
1	300	AAPL120818C00300000	277.10	1.7	284.80	285.60	2		478
2	305	AAPL120818C00305000	300.97	0.0	279.80	280.80	10		316
3	310	AAPL120818C00310000	267.05	0.0	274.80	275.65	6		239
4	315	AAPL120818C00315000	296.54	0.0	269.80	270.80	22		88
5	320	AAPL120818C00320000	291.63	0.0	264.80	265.80	96		173
6	325	AAPL120818C00325000	261.34	0.0	259.80	260.80	N/A		108
7	330	AAPL120818C00330000	230.25	0.0	254.80	255.80	N/A		21
8	335	AAPL120818C00335000	266.03	0.0	249.80	250.65	4		46
9	340	AAPL120818C00340000	272.58	0.0	244.80	245.80	4		30

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 применение библиотеки `lxml` и ее интерфейса `lxml.html`. А сейчас покажу альтернативный интерфейс, удобный для работы с XML-данными, — `lxml.objectify`.

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 компаний существует свой файл (например, `Performance_MNR.xml` для компании MetroNorth Railroad), содержащий данные за один 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 = '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 [927]: perf = DataFrame(data)
```

```
In [928]: perf
Out[928]:
Empty DataFrame
Columns: array([], dtype=int64)
Index: array([], dtype=int64)
```

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 StringIO 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 [930]: root
Out[930]: <Element a at 0x88bd4b0>

In [931]: root.get('href')
Out[931]: 'http://www.google.com'

In [932]: root.text
Out[932]: 'Google'
```

Двоичные форматы данных

Один из самых простых способов эффективного хранения данных в двоичном формате – воспользоваться встроенным в Python методом сериализации pickle. Поэтому у всех объектов pandas есть метод save, который сохраняет данные на диске в виде pickle-файла:

```
In [933]: frame = pd.read_csv('ch06/ex1.csv')

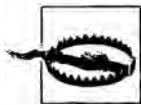
In [934]: frame
Out[934]:
   a   b   c   d  message
0  1  2  3  4  Hello
1  2  3  4  5  World
```

```
0  1   2   3   4      hello
1  5   6   7   8      world
2  9  10  11  12      foo
```

```
In [935]: frame.save('ch06/frame_pickle')
```

Прочитать данные с диска позволяет метод `pandas.load`, также упрощающий интерфейс с `pickle`:

```
In [936]: pd.load('ch06/frame_pickle')
Out[936]:
   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`.

Формат HDF5

Существует немало инструментов, предназначенных для эффективного чтения и записи больших объемов научных данных в двоичном формате. Популярна, в частности, библиотека промышленного качества `HDF5`, написанная на С и имеющая интерфейсы ко многим языкам, в том числе Java, 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`, в которых приняты совершенно различные подходы. `h5py` – прямой, хотя и высокоуровневый интерфейс к `HDF5 API`, тогда как `PyTables` абстрагирует многие детали `HDF5` с целью предоставления нескольких гибких контейнеров данных, средств индексирования таблиц, средств запроса и поддержки некоторых вычислений, отсутствующих в исходной библиотеке.

В `pandas` имеется минимальный похожий на словарь класс `HDFStore`, в котором для сохранения объектов `pandas` используется интерфейс `PyTables`:

```
In [937]: store = pd.HDFStore('mydata.h5')

In [938]: store['obj1'] = frame

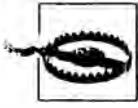
In [939]: store['obj1_col'] = frame['a']

In [940]: store
Out[940]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
obj1      DataFrame
obj1_col   Series
```

Объекты из HDF5-файла можно извлекать, как из словаря:

```
In [941]: store['obj1']
Out[941]:
   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 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 делают 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`, поэтому их нужно предварительно установить. Для работы с `ExcelFile` создайте его экземпляр, передав конструктору путь к файлу с расширением `xls` или `xlsx`:

```
xls_file = pd.ExcelFile('data.xls')
```

Прочитать данные из рабочего листа в объект `DataFrame` позволяет метод `parse`:

```
table = xls_file.parse('Sheet1')
```

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 разны-

ми способами; я рекомендую простой пакет `requests` (<http://docs.python-requests.org>). Для поиска по словам «python pandas» в Твиттере мы можем отправить такой HTTP-запрос GET:

```
In [944]: import requests  
In [945]: url = 'http://search.twitter.com/search.json?q=python%20pandas'  
In [946]: resp = requests.get(url)  
In [947]: resp  
Out[947]: <Response [200]>
```

У объекта `Response` имеет атрибут `text`, в котором хранится содержимое ответа на запрос GET. Многие API в веб возвращают JSON-строку, которую следует загрузить в объект Python:

```
In [948]: import json  
In [949]: data = json.loads(resp.text)  
In [950]: data.keys()  
Out[950]:  
[u'next_page',  
 u'completed_in',  
 u'max_id_str',  
 u'since_id_str',  
 u'refresh_url',  
 u'results',  
 u'since_id',  
 u'results_per_page',  
 u'query',  
 u'max_id',  
 u'page']
```

Поле ответа `results` содержит список твитов, каждый из которых представлен таким словарем Python:

```
{u'created_at': u'Mon, 25 Jun 2012 17:50:33 +0000',  
 u'from_user': u'wesmckinn',  
 u'from_user_id': 115494880,  
 u'from_user_id_str': u'115494880',  
 u'from_user_name': u'Wes McKinney',  
 u'geo': None,  
 u'id': 217313849177686018,  
 u'id_str': u'217313849177686018',  
 u'iso_language_code': u'pt',  
 u'metadata': {u'result_type': u'recent'},  
 u'source': u'<a href="http://twitter.com/">web</a>',  
 u'text': u'Lunchtime pandas-fu http://t.co/SI70xZZQ #pydata',  
 u'to_user': None,  
 u'to_user_id': 0,  
 u'to_user_id_str': u'0',  
 u'to_user_name': None}
```

Далее мы можем построить список интересующих нас полей твита и передать его конструктору DataFrame:

```
In [951]: tweet_fields = ['created_at', 'from_user', 'id', 'text']

In [952]: tweets = DataFrame(data['results'], columns=tweet_fields)

In [953]: tweets
Out[953]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15 entries, 0 to 14
Data columns:
created_at    15 non-null values
from_user     15 non-null values
id            15 non-null values
text          15 non-null values
dtypes: int64(1), object(3)
```

Теперь в каждой строке DataFrame находятся данные, извлеченные из одного твита:

```
In [121]: tweets.ix[7]
Out[121]:
created_at           Thu, 23 Jul 2012 09:54:00 +0000
from_user            deblike
id                  227419585803059201
text    pandas: powerful Python data analysis toolkit
Name: 7
```

Приложив толику усилий, вы сможете создать высокоуровневые интерфейсы к популярным в веб API, которые будут возвращать объекты DataFrame, легко поддающиеся анализу.

Взаимодействие с базами данных

Во многие приложения данные поступают не из файлов, потому что для хранения больших объемов данных текстовые файлы неэффективны. Широко используются реляционные базы данных на основе SQL (например, SQL Server, PostgreSQL и MySQL), а равно так называемые базы данных *NoSQL*, 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 я возьму базу данных SQLite, целиком размещающуюся в памяти, и драйвер sqlite3, включенный в стандартную библиотеку Python:

```
import sqlite3

query = """
```

```
CREATE TABLE test
(a VARCHAR(20), b VARCHAR(20),
c REAL, d INTEGER
);"""

con = sqlite3.connect(':memory:')
con.execute(query)
con.commit()
```

Затем вставлю несколько строк в таблицу:

```
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)
con.commit()
```

Большинство драйверов SQL, имеющихся в Python (PyODBC, psycopg2, MySQLdb, pymssql и т. д.), при выборе данных из таблицы возвращают список кортежей:

```
In [956]: cursor = con.execute('select * from test')

In [957]: rows = cursor.fetchall()

In [958]: rows
Out[958]:
[(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 необходимо еще имена столбцов, содержащиеся в атрибуте курсора description:

```
In [959]: cursor.description
Out[959]:
([('a', None, None, None, None, None),
 ('b', None, None, None, None, None),
 ('c', None, None, None, None, None),
 ('d', None, None, None, None, None))

In [960]: DataFrame(rows, columns=zip(*cursor.description)[0])
Out[960]:
      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. В pandas, точнее в модуле pandas.io.sql, имеется функция read_frame, которая упрощает эту процедуру. Нужно просто передать команду select и объект соединения:

```
In [961]: import pandas.io.sql as sql
In [962]: sql.read_frame('select * from test', con)
Out[962]:
      a          b    c   d
0  Atlanta  Georgia  1.25  6
1 Tallahassee  Florida  2.60  3
2 Sacramento California  1.70  5
```

Чтение и сохранение данных в MongoDB

Базы данных NoSQL весьма многообразны. Есть простые хранилища ключей и значений, напоминающие словарь, например BerkeleyDB или Tokyo Cabinet, а есть и документо-ориентированные базы, в которых похожий на словарь объект является основной единицей хранения. Я решил взять в качестве примера MongoDB (<http://mongodb.org>). Я запустил локальный экземпляр MongoDB на своей машине и подключился к ее порту по умолчанию с помощью pymongo, официального драйвера для MongoDB:

```
import pymongo
con = pymongo.Connection('localhost', port=27017)
```

В MongoDB документы хранятся в коллекциях в базе данных. Каждый экземпляр сервера MongoDB может обслуживать несколько баз данных, а в каждой базе может быть несколько коллекций. Допустим, я хочу сохранить данные, полученные ранее из Твиттера. Прежде всего, получаю доступ к коллекции твитов (пока пустой):

```
tweets = con.db.tweets
```

Затем запрашиваю список твитов и записываю каждый из них в коллекцию методом `tweets.save` (который сохраняет словарь Python в базе MongoDB):

```
import requests, json
url = 'http://search.twitter.com/search.json?q=python%20pandas'
data = json.loads(requests.get(url).text)

for tweet in data['results']:
    tweets.save(tweet)
```

Если затем я захочу получить все мои твиты из коллекции, то должен буду опросить ее, как показано ниже:

```
cursor = tweets.find({'from_user': 'wesmckinn'})
```

Возвращенный курсор – это итератор, который отдает каждый документ в виде словаря. Как и раньше, я могу преобразовать этот документ в DataFrame, возможно, ограничившись некоторым подмножеством полей данных:

```
tweet_fields = ['created_at', 'from_user', 'id', 'text']
result = DataFrame(list(cursor), columns=tweet_fields)
```



ГЛАВА 7.

Переформатирование данных: очистка, преобразование, слияние, изменение формы

Значительная часть времени программиста, занимающегося анализом и моделированием данных, уходит на подготовку данных: загрузку, очистку, преобразование и реорганизацию. 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 высокоДировневыми, 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.

Комбинирование и слияние наборов данных

Данные, хранящиеся в объектах pandas, можно комбинировать различными готовыми 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 стопку 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 [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
....:                  'data1': range(7)})

In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'],
....:                  'data2': range(3)})

In [17]: df1
Out[17]:
   data1  key
0      0    b
1      1    b
2      2    a
3      3    c
4      4    a
5      5    a
6      6    b

In [18]: df2
Out[18]:
   data2  key
0      0    a
1      1    b
2      2    d
```

Это пример слияния 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 [19]: pd.merge(df1, df2)
Out[19]:
   data1  key  data2
0      2    a      0
1      4    a      0
2      5    a      0
3      0    b      1
4      1    b      1
5      6    b      1
```

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 [20]: pd.merge(df1, df2, on='key')
Out[20]:
   data1  key  data2
0      2    a      0
1      4    a      0
2      5    a      0
3      0    b      1
```

```
4      1     b      1
5      6     b      1
```

Если имена столбцов в объектах различаются, то можно задать их порознь:

```
In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
...:                      'data1': range(7)})
In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'],
...:                      'data2': range(3)})
In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[23]:
   data1  lkey  data2  rkey
0      2     a      0     a
1      4     a      0     a
2      5     a      0     a
3      0     b      1     b
4      1     b      1     b
5      6     b      1     b
```

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 соединения:

```
In [24]: pd.merge(df1, df2, how='outer')
Out[24]:
   data1  key  data2
0      2     a      0
1      4     a      0
2      5     a      0
3      0     b      1
4      1     b      1
5      6     b      1
6      3     c    NaN
7     NaN     d      2
```

Для слияния типа *многие-ко-многим* поведение корректно определено, хотя на первый взгляд неочевидно. Вот пример:

```
In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
...:                      'data1': range(6)})
In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
...:                      'data2': range(5)})
In [27]: df1
Out[27]:
   data1  key
0      0     b
1      1     b
2      2     a
3      3     c
4      4     a
5      5     b
In [28]: df2
Out[28]:
   data2  key
0      0     a
1      1     b
2      2     a
3      3     b
4      4     d
```

1	1	b	1	1	b
2	2	a	2	2	a
3	3	c	3	3	b
4	4	a	4	4	d
5	5	b			

```
In [29]: pd.merge(df1, df2, on='key', how='left')
Out[29]:
```

	data1	key	data2
0	2	a	0
1	2	a	2
2	4	a	0
3	4	a	2
4	0	b	1
5	0	b	3
6	1	b	1
7	1	b	3
8	5	b	1
9	5	b	3
10	3	c	NaN

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 [30]: pd.merge(df1, df2, how='inner')
```

```
Out[30]:
```

	data1	key	data2
0	2	a	0
1	2	a	2
2	4	a	0
3	4	a	2
4	0	b	1
5	0	b	3
6	1	b	1
7	1	b	3
8	5	b	1
9	5	b	3

Для слияния по нескольким ключам следует передать список имен столбцов:

```
In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
...:                         'key2': ['one', 'two', 'one'],
...:                         'lval': [1, 2, 3]})
```

```
In [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
...:                         'key2': ['one', 'one', 'one', 'two'],
...:                         'rval': [4, 5, 6, 7]})
```

```
In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
Out[33]:
```

	key1	key2	lval	rval
0	bar	one	3	6

```

1   bar    two    NaN      7
2   foo    one     1      4
3   foo    one     1      5
4   foo    two     2    NaN

```

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 осах ниже), у функции `merge` имеется параметр `suffixes`, позволяющий задать строки, которые должны дописываться в конец одинаковых имен в левом и правом объектах DataFrame:

```
In [34]: pd.merge(left, right, on='key1')
```

```
Out[34]:
```

	key1	key2_x	lval	key2_y	rval
0	bar	one	3	one	6
1	bar	one	3	two	7
2	foo	one	1	one	4
3	foo	one	1	one	5
4	foo	two	2	one	4
5	foo	two	2	one	5

```
In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

```
Out[35]:
```

	key1	key2_left	lval	key2_right	rval
0	bar	one	3	one	6
1	bar	one	3	two	7
2	foo	one	1	one	4
3	foo	one	1	one	5
4	foo	two	2	one	4
5	foo	two	2	one	5

В табл. 7.1 приведена справка по аргументам функции `merge`. Соединение по индексу – тема следующего раздела.

Таблица 7.1. Аргументы функции `merge`

Аргумент	Описание
<code>left</code>	Объект DataFrame в левой части операции слияния
<code>right</code>	Объект DataFrame в правой части операции слияния
<code>how</code>	Допустимые значения: 'inner', 'outer', 'left', 'right'.

Аргумент	Описание
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

Слияние по индексу

Иногда ключ (или ключи) слияния 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 слияния следует использовать индекс:

```
In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
...:                      'value': range(6)})

In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [38]: left1
Out[38]:
   key  value
0    a      0
1    b      1
2    a      2
3    a      3
4    b      4
5    c      5

In [39]: right1
Out[39]:
   group_val
a        3.5
b        7.0

In [40]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[40]:
   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 слияние производится по пересекающимся ключам, но можно вместо пересечения выполнить объединение, указав внешнее соединение:

```
In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[41]:
   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 [42]: lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
....:                   'key2': [2000, 2001, 2002, 2001, 2002],
....:                   'data': np.arange(5.)})

In [43]: righth = DataFrame(np.arange(12).reshape((6, 2)),
....:                      index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
....:                             [2001, 2000, 2000, 2000, 2001, 2002]],
....:                      columns=['event1', 'event2'])

In [44]: lefth
Out[44]:
   data  key1  key2
0     0  Ohio  2000
1     1  Ohio  2001
2     2  Ohio  2002
3     3  Nevada 2001
4     4  Nevada 2002

In [45]: righth
Out[45]:
          event1  event2
Nevada  2001      0      1
              2000      2      3
Ohio    2000      4      5
              2000      6      7
              2001      8      9
              2002     10     11
```

В данном случае необходимо перечислить столбцы, по которым производится слияние, в виде списка (обращайте внимание на обработку повторяющихся значений в индексе):

```
In [46]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[46]:
   data  key1  key2  event1  event2
3     3  Nevada 2001      0      1
0     0  Ohio   2000      4      5
0     0  Ohio   2000      6      7
1     1  Ohio   2001      8      9
2     2  Ohio   2002     10     11

In [47]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
```

```
....: right_index=True, how='outer')
Out[47]:
   data    key1  key2  event1  event2
4   NaN  Nevada  2000      2      3
3     3  Nevada  2001      0      1
4     4  Nevada  2002      NaN      NaN
0     0    Ohio  2000      4      5
0     0    Ohio  2000      6      7
1     1    Ohio  2001      8      9
2     2    Ohio  2002     10     11
```

Употребление индексов в обеих частях слияния тоже не проблема:

```
In [48]: left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],
....: columns=['Ohio', 'Nevada'])
```

```
In [49]: right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14.]],
....: index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])
```

```
In [50]: left2
```

```
Out[50]:
   Ohio  Nevada
a     1      2
c     3      4
e     5      6
```

```
In [51]: right2
```

```
Out[51]:
   Missouri  Alabama
b          7      8
c          9     10
d         11     12
e         13     14
```

```
In [52]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

```
Out[52]:
   Ohio  Nevada  Missouri  Alabama
a     1      2      NaN      NaN
b    NaN      NaN      7      8
c     3      4      9     10
d    NaN      NaN     11     12
e     5      6     13     14
```

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 не пересекающимися столбцами. В предыдущем примере можно было бы написать:

```
In [53]: left2.join(right2, how='outer')
```

```
Out[53]:
   Ohio  Nevada  Missouri  Alabama
a     1      2      NaN      NaN
b    NaN      NaN      7      8
c     3      4      9     10
d    NaN      NaN     11     12
e     5      6     13     14
```

Отчасти из-за необходимости поддерживать совместимость (с очень старыми версиями pandas), метод `join` объекта DataFrame выполняет левое внешнее соединение. Он также поддерживает соединение с индексом переданного DataFrame по одному из столбцов вызывающего:

```
In [54]: left1.join(right1, on='key')
Out[54]:
   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 [55]: another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
....:                  index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])

In [56]: left2.join([right2, another])
Out[56]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1        2       NaN     NaN       7       8
c      3        4       9       10       9      10
e      5        6      13      14      11      12

In [57]: left2.join([right2, another], how='outer')
Out[57]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1        2       NaN     NaN       7       8
b    NaN      NaN       7       8     NaN     NaN
c      3        4       9       10       9      10
d    NaN      NaN      11      12     NaN     NaN
e      5        6      13      14      11      12
f    NaN      NaN     NaN     NaN      16      17
```

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 [58]: arr = np.arange(12).reshape((3, 4))

In [59]: arr
Out[59]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [60]: np.concatenate([arr, arr], axis=1)
Out[60]:
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, то как поступать с коллекцией осей: объединять или пересекать?
- Нужно ли иметь возможность идентифицировать группы в результирующем объекте?
- Имеет ли вообще какое-то значение ось конкатенации?

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 [61]: s1 = Series([0, 1], index=['a', 'b'])
In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])
In [63]: s3 = Series([5, 6], index=['f', 'g'])
```

Если передать их функции concat списком, то она «склеит» данные и индексы:

```
In [64]: pd.concat([s1, s2, s3])
Out[64]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
```

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 [65]: pd.concat([s1, s2, s3], axis=1)
Out[65]:
      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 [66]: s4 = pd.concat([s1 * 5, s3])
```

```
In [67]: pd.concat([s1, s4], axis=1) In [68]: pd.concat([s1, s4], axis=1, join='inner')
```

```
Out[67]:
```

	0	1
a	0	0
b	1	5
f	NaN	5
g	NaN	6

```
Out[68]:
```

	0	1
a	0	0
b	1	5

Можно даже задать, какие метки будут использоваться на других осях – с помощью параметра `join_axes`:

```
In [69]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

```
Out[69]:
```

	0	1
a	0	0
c	NaN	NaN
b	1	5
e	NaN	NaN

Проблема возникает из-за того, что в результирующем объекте не видно, конкатенацией каких объектов он получен. Допустим, что вместо этого требуется построить иерархический индекс на оси конкатенации. Для этого используется аргумент `keys`:

```
In [70]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
```

```
In [71]: result
```

```
Out[71]:
```

	one	two	three
a	0	0	0
b	1	1	1
f	5	5	5
g	6	6	6

О функции `unstack` будет рассказано гораздо подробнее ниже

```
In [72]: result.unstack()
```

```
Out[72]:
```

	one	two	three	a	b	f	g
a	0	0	0	0	1	NaN	NaN
b	1	1	1	1	1	NaN	NaN
f	5	5	5	5	5	5	5
g	6	6	6	6	6	6	6

При комбинировании Series вдоль оси `axis=1` элементы списка `keys` становятся заголовками столбцов объекта DataFrame:

```
In [73]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

```
Out[73]:
```

	one	two	three
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
```

Эта логика обобщается и на объекты DataFrame:

```
In [74]: df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
....:                  columns=['one', 'two'])
In [75]: df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
....:                  columns=['three', 'four'])
In [76]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[76]:
      level1      level2
      one  two  three  four
a      0   1      5   6
b      2   3     NaN  NaN
c      4   5      7   8
```

Если передать не список, а словарь объектов, то роль аргумента keys будут играть ключи словаря:

```
In [77]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[77]:
      level1      level2
      one  two  three  four
a      0   1      5   6
b      2   3     NaN  NaN
c      4   5      7   8
```

Есть еще два аргумента, управляющие созданием иерархического индекса (см. табл. 7.2):

```
In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
....:                 names=['upper', 'lower'])
Out[78]:
      upper      level1      level2
      one  two  three  four
a      0   1      5   6
b      2   3     NaN  NaN
c      4   5      7   8
```

Последнее замечание касается объектов DataFrame, в которых индекс строк не имеет смысла в контексте анализа:

```
In [79]: df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
In [80]: df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
In [81]: df1
Out[81]:
In [82]: df2
Out[82]:
```

	a	b	c	d		b	d	a
0	-0.204708	0.478943	-0.519439	-0.555730	0	0.274992	0.228913	1.352917
1	1.965781	1.393406	0.092908	0.281746	1	0.886429	-2.001637	-0.371843
2	0.769023	1.246435	1.007189	-1.296221				

В таком случае можно передать параметр `ignore_index=True`:

```
In [83]: pd.concat([df1, df2], ignore_index=True)
Out[83]:
```

	a	b	c	d
0	-0.204708	0.478943	-0.519439	-0.555730
1	1.965781	1.393406	0.092908	0.281746
2	0.769023	1.246435	1.007189	-1.296221
3	1.352917	0.274992	NaN	0.228913
4	-0.371843	0.886429	NaN	-2.001637

Таблица 7.2. 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 [84]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
....: index=['f', 'e', 'd', 'c', 'b', 'a'])

In [85]: b = Series(np.arange(len(a), dtype=np.float64),
....: index=['f', 'e', 'd', 'c', 'b', 'a'])

In [86]: b[-1] = np.nan

In [87]: a           In [88]: b           In [89]: np.where(pd.isnull(a), b, a)
Out[87]:          Out[88]:          Out[89]:
f      NaN        f      0        f      0.0
e      2.5        e      1        e      2.5
d      NaN        d      2        d      2.0
c      3.5        c      3        c      3.5
b      4.5        b      4        b      4.5
a      NaN        a      NaN     a      NaN
```

У объекта Series имеется метод `combine_first`, который выполняет эквивалент этой операции плюс выравнивание данных:

```
In [90]: b[:-2].combine_first(a[2:])
Out[90]:
a      NaN
b      4.5
c      3.0
d      2.0
e      1.0
f      0.0
```

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 [91]: df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
....:                      'b': [np.nan, 2., np.nan, 6.],
....:                      'c': range(2, 18, 4)})

In [92]: df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
....:                      'b': [np.nan, 3., 4., 6., 8.]})

In [93]: df1.combine_first(df2)
Out[93]:
   a    b    c
0  1  NaN    2
1  4    2    6
2  5    4   10
3  3    6   14
4  7    8  NaN
```

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 на примерах. Рассмотрим небольшой DataFrame, в котором индексы строк и столбцов – массивы строк.

```
In [94]: data = DataFrame(np.arange(6).reshape((2, 3)),
....:                      index=pd.Index(['Ohio', 'Colorado'], name='state'),
....:                      columns=pd.Index(['one', 'two', 'three'], name='number'))
```

```
In [95]: data
Out[95]:
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 [96]: result = data.stack()
```

```
In [97]: result
Out[97]:
state    number
Ohio      one      0
          two      1
          three    2
Colorado  one      3
          two      4
          three    5
```

Имея иерархически проиндексированный объект Series, мы можем восстановить DataFrame методом `unstack`:

```
In [98]: result.unstack()
Out[98]:
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 [99]: result.unstack(0)
Out[99]:
```

```
In [100]: result.unstack('state')
Out[100]:
```

```
state    Ohio  Colorado
number
one        0       3
two        1       4
three      2       5
```

```
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 [101]: s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
```

```
In [102]: s2 = Series([4, 5, 6], index=['c', 'd', 'e'])
```

```
In [103]: data2 = pd.concat([s1, s2], keys=['one', 'two'])
```

```
In [104]: data2.unstack()
```

```
Out[104]:
```

	a	b	c	d	e
one	0	1	2	3	NaN
two	NaN	NaN	4	5	6

При выполнении поворота отсутствующие данные по умолчанию отфильтровываются, поэтому операция обратима:

```
In [105]: data2.unstack().stack()
```

```
Out[105]:
```

	one	a	0
	b	1	
	c	2	
	d	3	
two	c	4	
	d	5	
	e	6	

```
In [106]: data2.unstack().stack(dropna=False)
```

```
Out[106]:
```

	one	a	0
	b	1	
	c	2	
	d	3	
two	a	NaN	e
	b	NaN	NaN
	c	4	
	d	5	
	e	6	

В случае обратного поворота DataFrame поворачиваемый уровень становится самым нижним уровнем результирующего объекта:

```
In [107]: df = DataFrame({'left': result, 'right': result + 5},
.....:                 columns=pd.Index(['left', 'right'], name='side'))
```

```
In [108]: df
```

```
Out[108]:
```

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 [109]: df.unstack('state')
```

```
In [110]: df.unstack('state').stack('side')
```

Out[109]:				Out[110]:			
side	left	right		state	Ohio	Colorado	
state	Ohio	Colorado	Ohio	Colorado	number	side	
number					one	left	0
one	0	3	5	8		right	5
two	1	4	6	9	two	left	1
three	2	5	7	10		right	6
					three	left	2
						right	7
							10

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*):

```
In [116]: ldata[:10]
Out[116]:
      date      item    value
0 1959-03-31 00:00:00  realgdp  2710.349
1 1959-03-31 00:00:00     infl    0.000
2 1959-03-31 00:00:00    unemp   5.800
3 1959-06-30 00:00:00  realgdp  2778.801
4 1959-06-30 00:00:00     infl    2.340
5 1959-06-30 00:00:00    unemp   5.100
6 1959-09-30 00:00:00  realgdp  2775.488
7 1959-09-30 00:00:00     infl    2.740
8 1959-09-30 00:00:00    unemp   5.300
9 1959-12-31 00:00:00  realgdp  2785.204
```

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 и запросы. Но есть и минус: с данными в длинном формате трудно работать; иногда 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 [117]: pivoted = ldata.pivot('date', 'item', 'value')
In [118]: pivoted.head()
Out[118]:
      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
```

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 [119]: ldata['value2'] = np.random.randn(len(ldata))
```

```
In [120]: ldata[:10]
```

```
Out[120]:
```

	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, мы получим DataFrame с иерархическими столбцами:

```
In [121]: pivoted = ldata.pivot('date', 'item')
```

```
In [122]: pivoted[:5]
```

```
Out[122]:
```

item	value			value2		
	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 [123]: pivoted['value'][:5]
```

```
Out[123]:
```

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 изменения формы с помощью unstack:

```
In [124]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [125]: unstacked[:7]
Out[125]:
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
```

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 [126]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
.....:                  'k2': [1, 1, 2, 3, 3, 4, 4]})

In [127]: data
Out[127]:
      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` возвращает булев объект Series, который для каждой строки показывает, есть в ней дубликаты или нет:

```
In [128]: data.duplicated()
Out[128]:
0    False
1     True
2    False
3    False
4     True
5    False
6     True
```

А метод `drop_duplicates` возвращает DataFrame, для которого массив `duplicated` будет содержать только значения `True`:

```
In [129]: data.drop_duplicates()
Out[129]:
   k1  k2
0  one  1
2  one  2
3  two  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 [130]: data['v1'] = range(7)
In [131]: data.drop_duplicates(['k1'])
Out[131]:
   k1  k2  v1
0  one  1    0
3  two  3    3
```

По умолчанию методы `duplicated` и `drop_duplicates` оставляют первую встретившуюся строку с данной комбинацией значений. Но если задать параметр `take_last=True`, то будет оставлена последняя строка:

```
In [132]: data.drop_duplicates(['k1', 'k2'], take_last=True)
Out[132]:
   k1  k2  v1
1  one  1    1
2  one  2    2
4  two  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 [133]: 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 [134]: data
Out[134]:
      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, что 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 отображений, но в данном случае возникает мелкая проблема: у нас названия некоторых сортов мяса начинаются с заглавной буквы, а остальные – со строчной. Поэтому нужно привести все строки к нижнему регистру:

```
In [136]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
```

```
In [137]: data
```

```
Out[137]:
```

	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 [138]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

```
Out[138]:
```

0	pig
1	pig
2	pig
3	cow
4	cow
5	pig
6	cow
7	pig
8	salmon

```
Name: food
```

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 [139]: data = Series([1., -999., 2., -999., -1000., 3.])  
  
In [140]: data  
Out[140]:  
0      1  
1    -999  
2      2  
3    -999  
4   -1000  
5      3
```

Значение `-999` могло бы быть маркером отсутствующих данных. Чтобы заменить все такие значения теми, которые понимает `pandas`, воспользуемся методом `replace`, порождающим новый объект `Series`:

```
In [141]: data.replace(-999, np.nan)  
Out[141]:  
0      1  
1    NaN  
2      2  
3    NaN  
4   -1000  
5      3
```

Чтобы заменить сразу несколько значений, нужно передать их список и заменяющее значение:

```
In [142]: data.replace([-999, -1000], np.nan)  
Out[142]:  
0      1  
1    NaN  
2      2  
3    NaN  
4    NaN  
5      3
```

Если для каждого заменяемого значения нужно свое заменяющее, передаем список замен:

```
In [143]: data.replace([-999, -1000], [np.nan, 0])  
Out[143]:
```

```
0      1  
1    NaN  
2      2  
3    NaN  
4      0  
5      3
```

В аргументе можно передавать также словарь:

```
In [144]: data.replace({-999: np.nan, -1000: 0})  
Out[144]:  
0      1  
1    NaN  
2      2  
3    NaN  
4      0  
5      3
```

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 [145]: 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 [146]: data.index.map(str.upper)  
Out[146]: array([OHIO, COLORADO, NEW YORK], dtype=object)
```

Индексу можно присваивать значение, т. е. модифицировать DataFrame на месте:

```
In [147]: data.index = data.index.map(str.upper)  
  
In [148]: data  
Out[148]:  
          one   two   three   four  
OHIO       0     1      2      3  
COLORADO   4     5      6      7  
NEW YORK  8     9     10     11
```

Если требуется создать преобразованный вариант набора данных, не меняя оригинал, то будет полезен метод rename:

```
In [149]: data.rename(index=str.title, columns=str.upper)  
Out[149]:  
          ONE   TWO   THREE   FOUR  
Ohio       0     1      2      3
```

```
Colorado    4    5    6    7
New York   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 [150]: data.rename(index={'OHIO': 'INDIANA'},
....:         columns={'three': 'peekaboo'})
Out[150]:
      one  two  peekaboo  four
INDIANA    0    1        2    3
COLORADO   4    5        6    7
NEW YORK   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`:

```
# Всегда возвращает ссылку на DataFrame
In [151]: _ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [152]: data
Out[152]:
      one  two  three  four
INDIANA    0    1        2    3
COLORADO   4    5        6    7
NEW YORK   8    9       10   11
```

Дискретизация и раскладывание

Непрерывные данные часто дискретизируются или как-то иначе раскладываются по интервалам – «ящикам» – для анализа. Предположим, что имеются данные о группе лиц в каком-то исследовании, и требуется разложить их ящикам, соответствующим возрасту – дискретной величине:

```
In [153]: 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 от 60 и старше. Для этой цели в `pandas` есть функция `cut`:

```
In [154]: bins = [18, 25, 35, 60, 100]
```

```
In [155]: cats = pd.cut(ages, bins)
```

```
In [156]: cats
```

```
Out[156]:
```

```
Categorical:
```

```
array([(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], (18, 25],
      (35, 60], (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]], dtype=object)
Levels (4): Index([(18, 25], (25, 35], (35, 60], (60, 100)], dtype=object)
```

`pandas` возвращает специальный объект `Categorical`. Его можно рассматривать как массив строк с именами ящика; на самом деле он содержит массив `levels`, в

котором хранятся неповторяющиеся имена категорий, а также метки данных `ages` в атрибуте `labels`:

```
In [157]: cats.labels
Out[157]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1])

In [158]: cats.levels
Out[158]: Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)

In [159]: pd.value_counts(cats)
Out[159]:
(18, 25)    5
(35, 60)    3
(25, 35)    3
(60, 100)   1
```

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 [160]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[160]:
Categorical:
array([[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), [18, 26),
       [36, 61), [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)], dtype=object)
Levels (4): Index([18, 26), [26, 36), [36, 61), [61, 100)], dtype=object)
```

Можно также самостоятельно задать имена ящиков, передав список или массив в параметре `labels`:

```
In [161]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [162]: pd.cut(ages, bins, labels=group_names)
Out[162]:
Categorical:
array([Youth, Youth, Youth, YoungAdult, Youth, Youth, MiddleAged,
       YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult], dtype=object)
Levels (4): Index([Youth, YoungAdult, MiddleAged, Senior], dtype=object)
```

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 значения. Рассмотрим раскладывание равномерно распределенных данных по четырем ящикам:

```
In [163]: data = np.random.rand(20)

In [164]: pd.cut(data, 4, precision=2)
Out[164]:
Categorical:
array([(0.45, 0.67], (0.23, 0.45], (0.0037, 0.23], (0.45, 0.67],
      (0.67, 0.9], (0.45, 0.67], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],
      (0.67, 0.9], (0.67, 0.9], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],
```

```
(0.23, 0.45], (0.67, 0.9], (0.0037, 0.23], (0.0037, 0.23],
(0.23, 0.45], (0.23, 0.45]], dtype=object)
Levels (4): Index([(0.0037, 0.23], (0.23, 0.45], (0.45, 0.67],
(0.67, 0.9]], dtype=object)
```

Родственный метод `qcut` раскладывает данные, исходя из выборочных квантилей. Метод `cut` обычно создает ящики, содержащие разное число точек, – это все-цело определяется распределением данных. Но поскольку `qcut` пользуется выборочными квантилями, то по определению получаются ящики равного размера:

```
In [165]: data = np.random.randn(1000) # Normally distributed
In [166]: cats = pd.qcut(data, 4) # Cut into quartiles
In [167]: cats
Out[167]:
Categorical:
array([(-0.022, 0.641], [-3.745, -0.635], (0.641, 3.26], ...,
(-0.635, -0.022], (0.641, 3.26], (-0.635, -0.022]], dtype=object)
Levels (4): Index([-3.745, -0.635], (-0.635, -0.022], (-0.022, 0.641],
(0.641, 3.26]], dtype=object)

In [168]: pd.value_counts(cats)
Out[168]:
[-3.745, -0.635]    250
(0.641, 3.26]      250
(-0.635, -0.022]   250
(-0.022, 0.641]    250
```

Как и в случае `cut`, можно задать величины квантилей (числа от 0 до 1 включительно) самостоятельно:

```
In [169]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[169]:
Categorical:
array([(-0.022, 1.302], (-1.266, -0.022], (-0.022, 1.302], ...,
(-1.266, -0.022], (-0.022, 1.302], (-1.266, -0.022]], dtype=object)
Levels (4): Index([-3.745, -1.266], (-1.266, -0.022], (-0.022, 1.302],
(1.302, 3.26]], dtype=object)
```

Мы еще вернемся к методам `cut` и `qcut` в главе об агрегировании и групповых операциях, 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 [170]: np.random.seed(12345)
In [171]: data = DataFrame(np.random.randn(1000, 4))
In [172]: data.describe()
```

Out[172]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067684	0.067924	0.025598	-0.002298
std	0.998035	0.992106	1.006835	0.996794
min	-3.428254	-3.548824	-3.184377	-3.745356
25%	-0.774890	-0.591841	-0.641675	-0.644144
50%	-0.116401	0.101143	0.002073	-0.013611
75%	0.616366	0.780282	0.680391	0.654328
max	3.366626	2.653656	3.260383	3.927528

Допустим, что мы хотим найти в одном из столбцов значения, превышающие 3 по абсолютной величине:

In [173]: col = data[3]

In [174]: col[np.abs(col) > 3]

Out[174]:
97 3.927528
305 -3.399312
400 -3.745356
Name: 3

Чтобы выбрать все строки, в которых встречаются значения, по абсолютной величине превышающие 3, мы можем воспользоваться методом any для булева объекта DataFrame:

In [175]: data[(np.abs(data) > 3).any(1)]

Out[175]:

	0	1	2	3
5	-0.539741	0.476985	3.248944	-1.021228
97	-0.774363	0.552936	0.106061	3.927528
102	-0.655054	-0.565230	3.176873	0.959533
305	-2.315555	0.457246	-0.025907	-3.399312
324	0.050188	1.951312	3.260383	0.963301
400	0.146326	0.508391	-0.196713	-3.745356
499	-0.293333	-0.242459	-3.056990	1.918403
523	-3.428254	-0.296336	-0.439938	-0.867165
586	0.275144	1.179227	-3.184377	1.369891
808	-0.362528	-3.548824	1.553205	-2.186301
900	3.366626	-2.372214	0.851010	1.332846

Можно также присваивать значения данным, удовлетворяющим этому критерию: Следующий код срезает значения, выходящие за границы интервала от -3 до 3:

In [176]: data[np.abs(data) > 3] = np.sign(data) * 3

In [177]: data.describe()

Out[177]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067684	0.067924	0.025598	-0.002298

```
std      0.998035      0.992106      1.006835      0.996794
min     -3.000000     -3.000000     -3.000000     -3.000000
25%    -0.774890    -0.591841    -0.641675    -0.644144
50%    -0.116401     0.101143     0.002073    -0.013611
75%     0.616366     0.780282     0.680391     0.654328
max      3.000000     2.653656     3.000000     3.000000
```

У-функция `np.sign` возвращает массив, содержащий 1 и -1 в зависимости от знака исходного значения.

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 [178]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))
```

```
In [179]: sampler = np.random.permutation(5)
```

```
In [180]: sampler
Out[180]: array([1, 0, 2, 3, 4])
```

Этот массив затем можно использовать для индексирования на основе поля `ix` или передать функции `take`:

<pre>In [181]: df Out[181]:</pre> <table border="0"> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>1</td><td>4</td><td>5</td><td>6</td></tr> <tr><td>2</td><td>8</td><td>9</td><td>10</td></tr> <tr><td>3</td><td>12</td><td>13</td><td>14</td></tr> <tr><td>4</td><td>16</td><td>17</td><td>18</td></tr> </table>	0	1	2	3	0	0	1	2	1	4	5	6	2	8	9	10	3	12	13	14	4	16	17	18	<pre>In [182]: df.take(sampler) Out[182]:</pre> <table border="0"> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>1</td><td>4</td><td>5</td><td>6</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>2</td><td>8</td><td>9</td><td>10</td></tr> <tr><td>3</td><td>12</td><td>13</td><td>14</td></tr> <tr><td>4</td><td>16</td><td>17</td><td>18</td></tr> </table>	0	1	2	3	1	4	5	6	0	0	1	2	2	8	9	10	3	12	13	14	4	16	17	18
0	1	2	3																																														
0	0	1	2																																														
1	4	5	6																																														
2	8	9	10																																														
3	12	13	14																																														
4	16	17	18																																														
0	1	2	3																																														
1	4	5	6																																														
0	0	1	2																																														
2	8	9	10																																														
3	12	13	14																																														
4	16	17	18																																														

Чтобы выбрать случайное подмножество без замены, можно, например, вырезать первые `k` элементов массива, возвращенного функцией `permutation`, где `k` – размер требуемого подмножества. Существуют гораздо более эффективные алгоритмы выборки без замены, но это простая стратегия, в которой применяются только уже имеющиеся инструменты:

```
In [183]: df.take(np.random.permutation(len(df))[:3])
```

```
Out[183]:
0   1   2   3
1   4   5   6   7
2   8   9   10  11
3  12  13  14  15
4  16  17  18  19
```

Самый быстрый способ сгенерировать выборку с заменой – воспользоваться функцией `np.random.randint`, которая возвращает множество случайных целых чисел:

```
In [184]: bag = np.array([5, 7, -1, 6, 4])  
  
In [185]: sampler = np.random.randint(0, len(bag), size=10)  
  
In [186]: sampler  
Out[186]: array([4, 4, 2, 2, 2, 0, 3, 0, 4, 1])  
  
In [187]: draws = bag.take(sampler)  
  
In [188]: draws  
Out[188]: array([ 4, 4, -1, -1, -1, 5, 6, 5, 4, 7])
```

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 к различных значений, то можно построить матрицу или объект DataFrame с к 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 [189]: df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],  
.....: 'data1': range(6)})  
  
In [190]: pd.get_dummies(df['key'])  
Out[190]:  
   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 [191]: dummies = pd.get_dummies(df['key'], prefix='key')  
  
In [192]: df_with_dummy = df[['data1']].join(dummies)  
  
In [193]: df_with_dummy  
Out[193]:  
   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. Вернемся к рассмотренному ранее набору данных MovieLens 1M:

```
In [194]: mnames = ['movie_id', 'title', 'genres']

In [195]: movies = pd.read_table('ch07/movies.dat', sep='::', header=None,
.....:             names=mnames)

In [196]: movies[:10]
Out[196]:
   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 (с помощью элегантного использования `set.union`):

```
In [197]: genre_iter = (set(x.split('|')) for x in movies.genres)

In [198]: genres = sorted(set.union(*genre_iter))
```

Теперь для построения индикаторного DataFrame можно, например, начать с объекта DataFrame, содержащего только нули:

```
In [199]: dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=genres)
```

Затем перебираем все фильмы и присваиваем элементам в каждой строке объекта `dummies` значение 1:

```
In [200]: for i, gen in enumerate(movies.genres):
.....:     dummies.ix[i, gen.split('|')] = 1
```

После этого можно, как и раньше, соединить с `movies`:

```
In [201]: movies_windic = movies.join(dummies.add_prefix('Genre_'))
```

```
In [202]: movies_windic.ix[0]
Out[202]:
   movie_id 1
   title          Toy Story (1995)
   genres         Animation|Children's|Comedy
   Genre_Action          0
   Genre_Adventure        0
```



Genre_Animation	1
Genre_Children's	1
Genre_Comedy	1
Genre_Crime	0
Genre_Documentary	0
Genre_Drama	0
Genre_Fantasy	0
Genre_Film-Noir	0
Genre_Horror	0
Genre_Musical	0
Genre_Mystery	0
Genre_Romance	0
Genre_Sci-Fi	0
Genre_Thriller	0
Genre_War	0
Genre_Western	0
Name: 0	0



Для очень больших наборов данных такой способ построения индикаторных переменных для нескольких категорий быстрым не назовешь. Но, безусловно, можно написать низкоуровневую функцию, в которой будут использованы знания о внутренней структуре объекта DataFrame.

В статистических приложениях бывает полезно сочетать функцию `get_dummies` с той или иной функцией дискретизации, например `cut`:

```
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

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 брать 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 по полям с помощью метода `split`:

```
In [208]: val = 'a,b, guido'  
In [209]: val.split(',')  
Out[209]: ['a', 'b', ' guido']
```

Метод `split` часто употребляется вместе с методом `strip`, чтобы убрать пробельные символы (в том числе перехода на новую строку):

```
In [210]: pieces = [x.strip() for x in val.split(',')]  
In [211]: pieces  
Out[211]: ['a', 'b', 'guido']
```

Чтобы конкатенировать строки, применяя в качестве разделителя двойное двоеточие, можно использовать оператор сложения:

```
In [212]: first, second, third = pieces  
In [213]: first + '::' + second + '::' + third  
Out[213]: 'a::b::guido'
```

Но это недостаточно общий метод. Быстрее и лучше соответствует духу Python другой способ: передать список или кортеж методу `join` строки '::':

```
In [214]: '::'.join(pieces)  
Out[214]: '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 [215]: 'guido' in val  
Out[215]: True  
  
In [216]: val.index(',') In [217]: val.find(':')  
Out[216]: 1 Out[217]: -1
```

Разница между `find` и `index` состоит в том, что `index` возбуждает исключение, если строка не найдена (вместо того чтобы возвращать `-1`):

```
In [218]: 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:

<pre>In [220]: val.replace(',', '::') Out[220]: 'a::b:: guido'</pre>	<pre>In [221]: val.replace(',', '') Out[221]: 'ab guido'</pre>
--	--

Как мы вскоре увидим, во многих таких операциях можно использовать также регулярные выражения.

Таблица 7.3. Встроенные в Python методы строковых объектов

Метод	Описание
<code>count</code>	Возвращает количество неперекрывающихся вхождений подстроки в строку
<code>endswith</code> , <code>startswith</code>	Возвращает <code>True</code> , если строка оканчивается (начинается) указанной подстрокой
<code>join</code>	Использовать данную строку как разделитель при конкатенации последовательности других строк
<code>index</code>	Возвращает позицию первого символа подстроки в строке. Если подстрока не найдена, возбуждает исключение <code>ValueError</code>
<code>find</code>	Возвращает позицию первого символа <i>первого</i> вхождения подстроки в строку, как и <code>index</code> . Но если строка не найдена, то возвращает <code>-1</code>
<code>rfind</code>	Возвращает позицию первого символа <i>последнего</i> вхождения подстроки в строку. Если строка не найдена, то возвращает <code>-1</code>
<code>replace</code>	Заменяет вхождения одной строки другой строкой
<code>strip</code> , <code>rstrip</code> , <code>lstrip</code>	Удаляет пробельные символы, в т. ч. символы новой строки в начале и (или) конце строки.
<code>split</code>	Разбивает строку на список подстрок по указанному разделителю
<code>lower</code> , <code>upper</code>	Преобразует буквы в нижний или верхний регистр соответственно
<code>ljust</code> , <code>rjust</code>	Выравнивает строку на левую или правую границу 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 имеется немало отличных пособий и справочных руководств, например, книга Зеда Шоу (Zed Shaw) «Learn Regex The Hard Way» (<http://regex.learncodethehardway.org/book/>).

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 [222]: import re

In [223]: text = "foo      bar\t baz  \tqux"

In [224]: re.split('\s+', text)
Out[224]: ['foo', 'bar', 'baz', 'qux']
```

При обращении `re.split(' \s+', text)` спачала компилируется регулярное выражение, а затем его методу `split` передается заданный текст. Можно просто откомпилировать регулярное выражение, создав тем самым объект, допускающий повторное использование:

```
In [225]: regex = re.compile(' \s+')

In [226]: regex.split(text)
Out[226]: ['foo', 'bar', 'baz', 'qux']
```

Чтобы получить список всех подстрок, отвечающих данному регулярному выражению, следует воспользоваться методом `findall`:

```
In [227]: regex.findall(text)
Out[227]: [' ', '\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` – только первое. А метод `match` находит *только* соответствие, начинающееся в начале строки. В качестве не столь тривиального примера рассмотрим блок текста и регулярное выражение, распознавающее 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 [229]: regex.findall(text)
Out[229]: ['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 позиции найденного в строке образца:

```
In [230]: m = regex.search(text)

In [231]: m
Out[231]: <_sre.SRE_Match at 0x10a05de00>

In [232]: text[m.start():m.end()]
Out[232]: 'dave@google.com'
```

Метод `regex.match` возвращает `None`, потому что он находит соответствие образцу только в начале строки:

```
In [233]: print regex.match(text)
None
```

Метод `sub` возвращает новую строку, в которой вхождения образца заменены указанной строкой:

```
In [234]: 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 [235]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
```

```
In [236]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Метод `groups` объекта соответствия, порожденного таким модифицированным регулярным выражением, возвращает кортеж компонентов образца:

```
In [237]: m = regex.match('wesm@bright.net')
```

```
In [238]: m.groups()
```

```
Out[238]: ('wesm', 'bright', 'net')
```

Если в образце есть группы, то метод `findall` возвращает список кортежей:

```
In [239]: regex.findall(text)
```

```
Out[239]:
```

```
[('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 [240]: 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 большая часть этого материала выходит за рамки данной книги. Просто для иллюстрации приведу вариант регулярного выражения для распознавания почтовых адресов, в котором группам присваиваются имена:

```
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @@
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

Объект соответствия, порожденный таким регулярным выражением, может генерировать удобный словарь с указанными именами групп:

```
In [242]: m = regex.match('wesm@bright.net')
```

```
In [243]: m.groupdict()
```

```
Out[243]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}
```

Таблица 7.4. Методы регулярных выражений

Метод	Описание
findall, finditer	Возвращает все непересекающиеся образцы, найденные в строке. <code>findall</code> возвращает список всех образцов, а <code>finditer</code> – итератор, который перебирает их поодиночке
match	Ищет соответствие образцу в начале строки и факультативно выделяет в образце группы. Если образец найден, возвращает объект соответствия, иначе <code>None</code>
search	Ищет в строке образец; если найден, возвращает объект соответствия. В отличие от <code>match</code> , 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 (<code>sub</code>) или только первые <code>n</code> (<code>subn</code>) вхождений образца указанной строкой. Чтобы в указанной строке сослаться на группы, выделенные в образце, используйте конструкции <code>\1, \2, ...</code>

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 [244]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
.....:           'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [245]: data = Series(data)

In [246]: data
Out[246]:
Dave      dave@google.com
Rob       rob@gmail.com
Steve     steve@gmail.com
Wes        NaN

In [247]: data.isnull()
Out[247]:
Dave      False
Rob      False
Steve     False
Wes       True
```

Методы строк и регулярных выражений можно применить к каждому значению с помощью метода `data.map` (к которому передается лямбда или другая функция), но для отсутствующих значений они «грехнутся». Чтобы справиться 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 [248]: data.str.contains('gmail')
Out[248]:
Dave      False
Rob       True
Steve     True
Wes        NaN
```

Регулярные выражения тоже можно так использовать, равно как и их флаги типа IGNORECASE:

```
In [249]: pattern
Out[249]: '(([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.\.([A-Z]{2,4}))'
```

```
In [250]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[250]:
Dave      [('dave', 'google', 'com')]
Rob       [('rob', 'gmail', 'com')]
Steve    [('steve', 'gmail', 'com')]
Wes          NaN
```

Существует два способа векторной выборки элементов: str.get или доступ к атрибуту str по индексу:

```
In [251]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [252]: matches
Out[252]:
Dave      ('dave', 'google', 'com')
Rob       ('rob', 'gmail', 'com')
Steve    ('steve', 'gmail', 'com')
Wes          NaN
```

```
In [253]: matches.str.get(1)      In [254]: matches.str[0]
Out[253]:                      Out[254]:
Dave      google                Dave     dave
Rob       gmail                 Rob      rob
Steve    gmail                 Steve   steve
Wes        NaN                  Wes     NaN
```

Аналогичный синтаксис позволяет вырезать строки:

```
In [255]: data.str[:5]
Out[255]:
Dave      dave@
Rob       rob@g
Steve    steve
Wes        NaN
```

Таблица 7.5. Векторные методы 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
endswith, startswith	Эквивалентно x.endswith(pattern) или x.startswith(pattern) для каждого элемента
findall	Возвращает список всех вхождений образца для каждой строки

Метод	Описание
get	Доступ по индексу ко всем элементам (выбрать i-ый элемент)
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, rstrip, lstrip	Убирает пробельные символы, в т. ч. знак новой строки, в начале, в конце или с обеих сторон строки; эквивалентно x.strip() (и соответственно rstrip, lstrip) для каждого элемента

Пример: база данных о продуктах питания министерства сельского хозяйства США

Министерство сельского хозяйства США публикует данные о пищевой ценности продуктов питания. Английский программист Эшли Уильямс (Ashley Williams) преобразовал эту базу данных в формат JSON (<http://ashleyw.co.uk/project/food-nutrient-database>). 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 затем можете загрузить его в Python-программу с помощью любой библиотеки для работы с JSON. Я воспользуюсь стандартным модулем Python json:

```
In [256]: import json

In [257]: db = json.load(open('ch07/foods-2011-10-03.json'))

In [258]: len(db)
Out[258]: 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 [259]: db[0].keys() In [260]: db[0]['nutrients'][0]
Out[259]: Out[260]:
[u'portions', {u'description': u'Protein',
  u'description', u'group': u'Composition',
  u'tags', u'units': u'g',
  u'nutrients', u'value': 25.18},
 u'group',
 u'id',
 u'manufacturer']

In [261]: nutrients = DataFrame(db[0]['nutrients'])

In [262]: nutrients[:7]
Out[262]:
      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 [263]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [264]: info = DataFrame(db, columns=info_keys)

In [265]: info[:5]
Out[265]:
          description           group   id  manufacturer
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

In [266]: info
Out[266]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
description  6636 non-null values
group        6636 non-null values
id           6636 non-null values
manufacturer 5195 non-null values
dtypes: int64(1), object(3)
```

Метод `value_counts` покажет распределение продуктов питания по группам:

```
In [267]: pd.value_counts(info.group)[:10]
Out[267]:
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
```

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 помешу этот DataFrame в список. После этого 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 [269]: nutrients
Out[269]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 389355 entries, 0 to 389354
Data columns:
description    389355 non-null values
group          389355 non-null values
units           389355 non-null values
value           389355 non-null values
id              389355 non-null values
dtypes: float64(1), int64(1), object(3)

```

Я заметил, что по какой-то причине в этом `DataFrame` есть дубликаты, поэтому лучше их удалить:

```

In [270]: nutrients.duplicated().sum()

Out[270]: 14179
In [271]: nutrients = nutrients.drop_duplicates()

```

Поскольку столбцы `'group'` и `'description'` есть в обоих объектах `DataFrame`, переименуем их, чтобы было понятно, что есть что:

```

In [272]: col_mapping = {'description' : 'food',
.....:                 'group' : 'fgroup'}

In [273]: info = info.rename(columns=col_mapping, copy=False)

In [274]: info
Out[274]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
food          6636 non-null values
fgroup        6636 non-null values
id            6636 non-null values
manufacturer  5195 non-null values
dtypes: int64(1), object(3)

In [275]: col_mapping = {'description' : 'nutrient',
.....:                 'group' : 'nutgroup'}

In [276]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [277]: nutrients
Out[277]:

```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 389354
Data columns:
nutrient    375176 non-null values
nutgroup    375176 non-null values
units       375176 non-null values
value       375176 non-null values
id          375176 non-null values
dtypes: float64(1), int64(1), object(3)
```

Сделав все это, мы можем слить info с nutrients:

```
In [278]: ndata = pd.merge(nutrients, info, on='id', how='outer')
```

```
In [279]: ndata
Out[279]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns:
nutrient    375176 non-null values
nutgroup    375176 non-null values
units       375176 non-null values
value       375176 non-null values
id          375176 non-null values
food        375176 non-null values
fgroup      375176 non-null values
manufacturer 293054 non-null values
dtypes: float64(1), int64(1), object(6)
```

```
In [280]: ndata.ix[30000]
Out[280]:
nutrient              Folic acid
nutgroup              Vitamins
units                 mcg
value                 0
id                    5658
food      Ostrich, top loin, cooked
fgroup      Poultry Products
manufacturer
Name: 30000
```

Средства, необходимые для формирования продольных и поперечных срезов, агрегирования и визуализации этого набора данных, будут подробно рассмотрены в следующих двух главах; познакомившись с ними, вы сможете вернуться к этому набору. Для примера можно было бы построить график медианных значений по группе и типу питательного элемента (рис. 7.1):

```
In [281]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)
```

```
In [282]: 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.xs(x.value.idxmax())
get_minimum = lambda x: x.xs(x.value.idxmin())

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# Немного уменьшить продукт питания
max_foods.food = max_foods.food.str[:50]

```

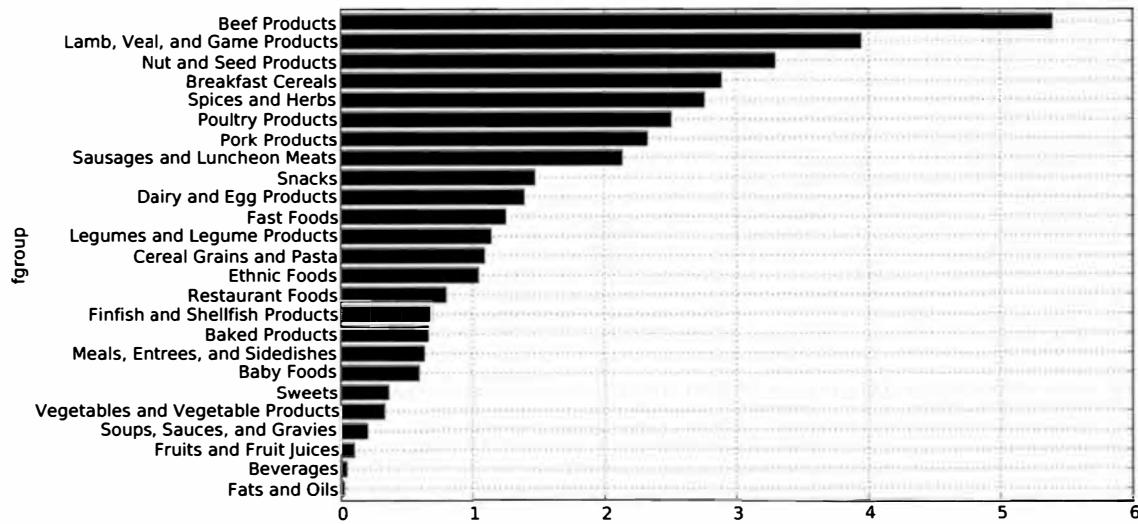


Рис. 7.1. Медианные значения цинка по группе питательных элементов

Получившийся объект DataFrame слишком велик для того, чтобы приводить его полностью. Ниже приведена только группа питательных элементов 'Amino Acids' (аминокислоты):

```

In [284]: max_foods.ix['Amino Acids']['food']
Out[284]:
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
Glycine                                 Gelatins, dry powder, unsweetened
Histidine                            Whale, beluga, meat, dried (Alaska Native)
Hydroxyproline                         KENTUCKY FRIED CHICKEN, Fried Chicken, ORIGINAL R
Isoleucine                            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Leucine                                Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Lysine                                 Seal, bearded (Oogruk), meat, dried (Alaska Nativ
Methionine                           Fish, cod, Atlantic, dried and salted
Phenylalanine                         Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Proline                                Gelatins, dry powder, unsweetened
Serine                                 Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Threonine                            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Tryptophan                           Sea lion, Steller, meat with fat (Alaska Native)

```

Tyrosine

Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA

Valine

Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA

Name: food