

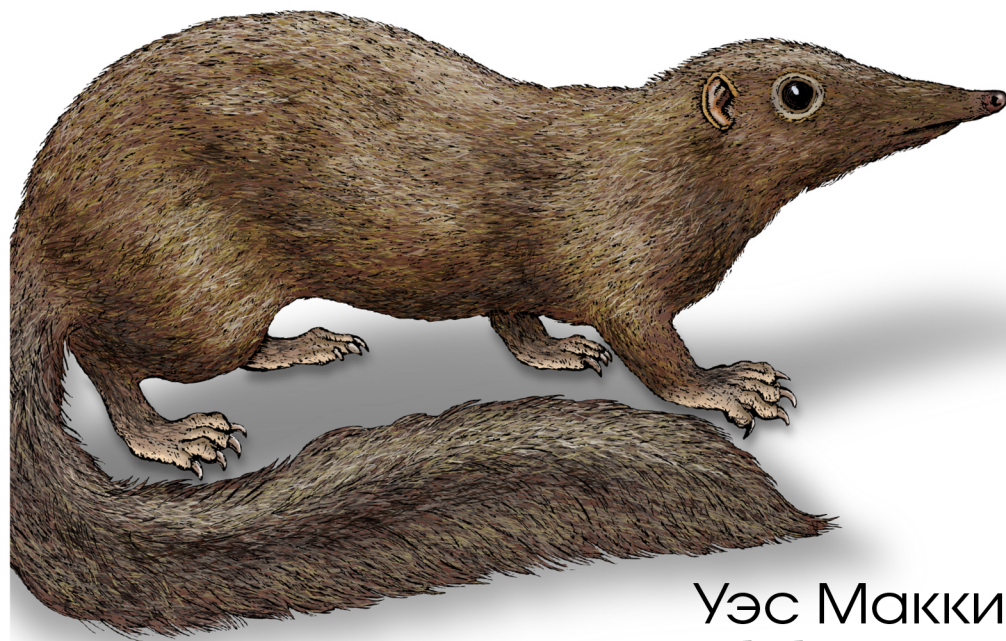
O'REILLY®

Python

и анализ данных

Первичная обработка данных
с применением pandas, NumPy и Jupiter

Третье издание



Уэс Маккинни,
создатель библиотеки pandas

Уэс Маккинни

Python и анализ данных

Первичная обработка данных с применением
pandas, NumPy и Jupiter

Third edition

Python for Data Analysis

Data Wrangling with pandas,
NumPy & Jupyter

Wes McKinney

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Третье издание

Python и анализ данных

Первичная обработка данных с применением
pandas, NumPy и Jupiter

Уэс Маккинни



Москва, 2023

УДК 004.438Python:004.6

ББК 32.973.22

M15

Уэс Маккинни

M15 Python и анализ данных: Первичная обработка данных с применением pandas, NumPy и Jupiter / пер. с англ. А. А. Слинкина. 3-е изд. – М.: МК Пресс, 2023. – 536 с.: ил.

ISBN 978-5-93700-174-0

Перед вами авторитетный справочник по переформатированию, очистке и обработке наборов данных на Python. Третье издание, переработанное с учетом версий Python 3.10 и pandas 1.4, содержит практические примеры, демонстрирующие эффективное решение широкого круга задач анализа данных. По ходу дела вы узнаете о последних версиях pandas, NumPy и Jupiter.

Книга принадлежит перу Уэса Маккинни, создателя библиотеки pandas, и может служить практическим современным руководством по инструментарию науки о данных на Python. Она идеально подойдет как аналитикам, только начинающим осваивать Python, так и программистам на Python, еще незнакомым с наукой о данных и научными приложениями. Файлы данных и прочие материалы к книге находятся в репозитории на GitHub и на сайте издательства dmkpress.com.

УДК 004.438Python:004.6

ББК 32.973.22

Authorized Russian translation of the English edition of Python for Data Analysis, 2nd edition. ISBN 9781491957660 © 2022 Wesley McKinney.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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.

ISBN (анг.) 978-1-09810-403-0

ISBN (рус.) 978-5-93700-174-0

Copyright © 2022 Wesley McKinney

© Оформление, издание, перевод, ДМК Пресс, 2023

Оглавление

Об авторе	13
Об иллюстрации на обложке	14
Предисловие от издательства	15
Предисловие	16
Графические выделения	16
О примерах кода	17
Как с нами связаться	17
Благодарности	18
Глава 1. Предварительные сведения.....	22
1.1. О чем эта книга?.....	22
Какого рода данные?	22
1.2. Почему именно Python?	23
Python как клей.....	23
Решение проблемы «двух языков»	24
Недостатки Python.....	24
1.3. Необходимые библиотеки для Python	25
NumPy.....	25
pandas	25
matplotlib.....	27
IPython и Jupyter	27
SciPy.....	28
scikit-learn	28
statsmodels	29
1.4. Установка и настройка.....	29
Miniconda в Windows	30
GNU/Linux.....	30
Miniconda в macOS.....	31
Установка необходимых пакетов	32
Интегрированные среды разработки (IDE)	33
1.5. Сообщество и конференции.....	33
1.6. Структура книги.....	34
Примеры кода	35
Данные для примеров	35
Соглашения об импорте.....	36

Глава 2. Основы языка Python, IPython и Jupyter-блокноты..... 37

2.1. Интерпретатор Python.....	38
2.2. Основы IPython	39
Запуск оболочки IPython.....	39
Запуск Jupyter-блокнота.....	40
Завершение по нажатию клавиши Tab.....	43
Интроспекция.....	45
2.3. Основы языка Python.....	46
Семантика языка	46
Скалярные типы	53
Поток управления.....	61
2.4. Заключение	64

Глава 3. Встроенные структуры данных, функции и файлы..... 65

3.1. Структуры данных и последовательности	65
Кортеж	65
Список	68
Словарь.....	72
Множество.....	76
Встроенные функции последовательностей	78
Списковое, словарное и множественное включения.....	80
3.2. Функции.....	82
Пространства имен, области видимости и локальные функции	83
Возврат нескольких значений	84
Функции являются объектами.....	85
Анонимные (лямбда-) функции	87
Генераторы.....	87
Обработка исключений.....	90
3.3. Файлы и операционная система	92
Байты и Unicode в применении к файлам	96
3.4. Заключение	98

Глава 4. Основы NumPy: массивы и векторные вычисления 99

4.1. NumPy ndarray: объект многомерного массива.....	101
Создание ndarray	102
Тип данных для ndarray	104
Арифметические операции с массивами NumPy.....	107
Индексирование и вырезание	108
Булево индексирование	113
Прихотливое индексирование.....	116
Транспонирование массивов и перестановка осей	117
4.2. Генерирование псевдослучайных чисел.....	119
4.3. Универсальные функции: быстрые поэлементные операции над массивами	120
4.4. Программирование на основе массивов.....	123
Запись логических условий в виде операций с массивами.....	125

Математические и статистические операции.....	126
Методы булевых массивов.....	128
Сортировка.....	128
Устранение дубликатов и другие теоретико-множественные операции	130
4.5. Файловый ввод-вывод массивов	130
4.6. Линейная алгебра.....	131
4.7. Пример: случайное блуждание.....	133
Моделирование сразу нескольких случайных блужданий	135
4.8. Заключение	136
Глава 5. Первое знакомство с pandas.....	137
5.1. Введение в структуры данных pandas	138
Объект Series	138
Объект DataFrame	142
Индексные объекты.....	149
5.2. Базовая функциональность.....	151
Переиндексация	151
Удаление элементов из оси.....	154
Доступ по индексу, выборка и фильтрация	155
Арифметические операции и выравнивание данных.....	165
Применение функций и отображение	170
Сортировка и ранжирование	172
Индексы по осям с повторяющимися значениями	175
5.3. Редукция и вычисление описательных статистик.....	177
Корреляция и ковариация	180
Уникальные значения, счетчики значений и членство.....	181
5.4. Заключение	185
Глава 6. Чтение и запись данных, форматы файлов.....	186
6.1. Чтение и запись данных в текстовом формате.....	186
Чтение текстовых файлов порциями	193
Вывод данных в текстовом формате.....	195
Обработка данных в других форматах с разделителями.....	196
Данные в формате JSON.....	198
XML и HTML: разбор веб-страниц.....	200
6.2. Двоичные форматы данных.....	203
Формат HDF5.....	205
6.3. Взаимодействие с HTML и Web API	208
6.4. Взаимодействие с базами данных	209
6.5. Заключение	211
Глава 7. Очистка и подготовка данных.....	212
7.1. Обработка отсутствующих данных	212
Фильтрация отсутствующих данных	214
Восполнение отсутствующих данных.....	216

7.2. Преобразование данных	218
Устранение дубликатов	218
Преобразование данных с помощью функции или отображения	220
Замена значений	221
Переименование индексов осей	222
Дискретизация и группировка по интервалам	223
Обнаружение и фильтрация выбросов	226
Перестановки и случайная выборка	227
Вычисление индикаторных переменных	229
7.3. Расширение типов данных	232
7.4. Манипуляции со строками	235
Встроенные методы строковых объектов	235
Регулярные выражения	237
Строковые функции в pandas	240
7.5. Категориальные данные	243
Для чего это нужно	244
Расширенный тип Categorical в pandas	245
Вычисления с объектами Categorical	248
Категориальные методы	250
7.6. Заключение	253

Глава 8. Переформатирование данных:

соединение, комбинирование и изменение формы..... 254

8.1. Иерархическое индексирование	254
Переупорядочение и уровни сортировки	257
Сводная статистика по уровню	258
Индексирование столбцами DataFrame	258
8.2. Комбинирование и слияние наборов данных	260
Слияние объектов DataFrame как в базах данных	260
Соединение по индексу	265
Конкатенация вдоль оси	269
Комбинирование перекрывающихся данных	274
8.3. Изменение формы и поворот	276
Изменение формы с помощью иерархического индексирования	276
Поворот из «длинного» в «широкий» формат	279
Поворот из «широкого» в «длинный» формат	282
8.4. Заключение	284

Глава 9. Построение графиков и визуализация..... 285

9.1. Краткое введение в API библиотеки matplotlib	286
Рисунки и подграфики	287
Цвета, маркеры и стили линий	291
Риски, метки и надписи	292
Аннотации и рисование в подграфике	295
Сохранение графиков в файле	297

Конфигурирование matplotlib	298
9.2. Построение графиков с помощью pandas и seaborn.....	299
Линейные графики.....	299
Столбчатые диаграммы	302
Гистограммы и графики плотности	308
Диаграммы рассеяния.....	310
Фасетные сетки и категориальные данные	313
9.3. Другие средства визуализации для Python	315
9.4. Заключение	316

Глава 10. Агрегирование данных и групповые операции 317

10.1. Как представлять себе групповые операции	318
Обход групп.....	322
Выборка столбца или подмножества столбцов	323
Группировка с помощью словарей и объектов Series	324
Группировка с помощью функций.....	325
Группировка по уровням индекса.....	325
10.2. Агрегирование данных	326
Применение функций, зависящих от столбца, и нескольких функций	328
Возврат агрегированных данных без индексов строк	332
10.3. Метод apply: общий принцип	
разделения–применения–объединения	332
Подавление групповых ключей.....	334
Квантильный и интервальный анализы.....	335
Пример: подстановка зависящих от группы значений	
вместо отсутствующих	337
Пример: случайная выборка и перестановка	339
Пример: групповое взвешенное среднее и корреляция.....	341
Пример: групповая линейная регрессия	343
10.4. Групповые преобразования и «развернутая» группировка	343
10.5. Сводные таблицы и перекрестная табуляция	347
Перекрестная табуляция: crosstab.....	350
10.5. Заключение.....	351

Глава 11. Временные ряды 352

11.1. Типы данных и инструменты, относящиеся к дате и времени	353
Преобразование между строкой и datetime	354
11.2. Основы работы с временными рядами.....	356
Индексирование, выборка, подмножества	358
Временные ряды с неуникальными индексами.....	360
11.3. Диапазоны дат, частоты и сдвиг	361
Генерирование диапазонов дат	362
Частоты и смещения дат	364
Сдвиг данных (с опережением и с запаздыванием)	366
11.4. Часовые пояса.....	369
Локализация и преобразование	369

Операции над объектами Timestamp с учетом часового пояса	371
Операции над датами из разных часовых поясов	372
11.5. Периоды и арифметика периодов	373
Преобразование частоты периода	374
Квартальная частота периода	376
Преобразование временных меток в периоды и обратно	377
Создание PeriodIndex из массивов	379
11.6. Передискретизация и преобразование частоты	380
Понижающая передискретизация	382
Повышающая передискретизация и интерполяция	384
Передискретизация периодов	386
Групповая передискретизация по времени	387
11.7. Скользящие оконные функции	389
Экспоненциально взвешенные функции	392
Бинарные скользящие оконные функции	394
Скользящие оконные функции, определенные пользователем	395
11.8. Заключение	396

Глава 12. Введение в библиотеки моделирования на Python..... 397

12.1. Интерфейс между pandas и кодом модели	397
12.2. Описание моделей с помощью Patsy	400
Преобразование данных в формулах Patsy	402
Категориальные данные и Patsy	404
12.3. Введение в statsmodels	406
Оценивание линейных моделей	407
Оценивание процессов с временными рядами	409
12.4. Введение в scikit-learn	410
12.5. Заключение	414

Глава 13. Примеры анализа данных..... 415

13.1. Набор данных Bitly с сайта 1.usa.gov	415
Подсчет часовых поясов на чистом Python	416
Подсчет часовых поясов с помощью pandas	418
13.2. Набор данных MovieLens 1M	424
Измерение несогласия в оценках	428
13.3. Имена, которые давали детям в США за период с 1880 по 2010 год....	432
Анализ тенденций в выборе имен	437
13.4. База данных о продуктах питания министерства сельского хозяйства США	446
13.5. База данных Федеральной избирательной комиссии	451
Статистика пожертвований по роду занятий и месту работы	454
Распределение суммы пожертвований по интервалам	457
Статистика пожертвований по штатам	459
13.6. Заключение	460

Приложение А. Дополнительные сведения**о библиотеке NumPy 461**

А.1. Внутреннее устройство объекта ndarray.....	461
Иерархия типов данных в NumPy	462
А.2. Дополнительные манипуляции с массивами.....	463
Изменение формы массива	464
Упорядочение элементов массива в С и в Fortran.....	465
Конкатенация и разбиение массива	466
Эквиваленты прихотливого индексирования: функции take и put	470
А.3. Укладывание	471
Укладывание по другим осям.....	474
Установка элементов массива с помощью укладывания	476
А.4. Дополнительные способы использования универсальных функций.....	477
Методы экземпляра u-функций	477
Написание новых u-функций на Python.....	479
А.5. Структурные массивы и массивы записей	480
Вложенные типы данных и многомерные поля.....	481
Зачем нужны структурные массивы?.....	482
А.6. Еще о сортировке.....	482
Косвенная сортировка: методы argsort и lexsort.....	483
Альтернативные алгоритмы сортировки	485
Частичная сортировка массивов	485
Метод numpy.searchsorted: поиск элементов в отсортированном массиве	486
А.7. Написание быстрых функций для NumPy с помощью Numba	487
Создание пользовательских объектов numpy.ufunc с помощью Numba	489
А.8. Дополнительные сведения о вводе-выводе массивов.....	489
Файлы, отображенные на память.....	489
HDF5 и другие варианты хранения массива.....	491
А.9. Замечания о производительности	491
Важность непрерывной памяти	491

Приложение В. Еще о системе IPython..... 494

В.1. Комбинации клавиш	494
В.2. О магических командах	495
Команда %run	497
Исполнение кода из буфера обмена.....	498
В.3. История команд	499
Поиск в истории команд и повторное выполнение.....	500
Входные и выходные переменные	500
В.4. Взаимодействие с операционной системой	501
Команды оболочки и псевдонимы	502
Система закладок на каталоги	503

В.5. Средства разработки программ.....	504
Интерактивный отладчик.....	504
Хронометраж программы: %time и %timeit	508
Простейшее профилирование: %prun и %run -p.....	510
Построчное профилирование функции.....	512
В.6. Советы по продуктивной разработке кода с использованием IPython	514
Перезагрузка зависимостей модуля.....	514
Советы по проектированию программ.....	515
В.7. Дополнительные возможности IPython	516
Профили и конфигурирование.....	516
В.8. Заключение	517
Предметный указатель	518

Об авторе

Уэс Маккинни – разработчик программного обеспечения и предприниматель из Нэшвилла. Получив степень бакалавра математики в МТИ в 2007 году, он поступил на работу в компанию AQR Capital Management в Гринвиче, где занимался финансовой математикой. Неудовлетворенный малоприспособленными средствами анализа данных, Уэс изучил язык Python и приступил к созданию того, что в будущем стало проектом pandas. Сейчас он активный член сообщества обработки данных на Python и агитирует за использование Python в анализе данных, финансовых задачах и математической статистике.

Впоследствии Уэс стал сооснователем и генеральным директором компании DataPad, технологические активы и коллектив которой в 2014 году приобрела компания Cloudera. С тех пор он занимается технологиями больших данных и является членом комитетов по управлению проектами Apache Arrow и Apache Parquet, курируемых фондом Apache Software Foundation. В 2018 году он основал компанию Ursa Labs, некоммерческую организацию, ориентированную на разработку проекта Apache Arrow в партнерстве с компаниями RStudio и Two Sigma Investments. В 2021 году вошел в число учредителей технологического стартапа Voltron Data, где в настоящее время занимает пост технического директора.

Об иллюстрации на обложке

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

Эти дикие млекопитающие, обитающие в основном в Индонезии, Малайзии и Таиланде, известны хроническим потреблением алкоголя. Как выяснилось, малайские тупайи несколько часов в сутки пьют естественно ферментированный нектар пальмы *Eugeissona tristis*, что эквивалентно употреблению от 10 до 12 стаканов вина, содержащего 3.8 % алкоголя. Но это не приводит к интоксикации благодаря развитой способности расщеплять этиловый спирт, включая его в обмен веществ способами, недоступными человеку. Кроме того, поражает отношение массы мозга к массе тела – оно больше, чем у всех прочих млекопитающих, включая и человека.

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

Предисловие от издательства

Отзывы и пожелания

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

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

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

Список опечаток

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

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

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

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

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

Предисловие

Первое издание этой книги вышло в 2012 году, когда Python-библиотеки для анализа данных с открытым исходным кодом (в частности, pandas) были еще вновь и быстро развивались. Когда в 2016–2017 годах пришло время написать второе издание, мне пришлось не только привести текст в соответствие с версией Python (в первом издании использовалась версия Python 2.7), но и учесть многочисленные изменения, внесенные в pandas за прошедшие пять лет. Теперь, в 2022 году, изменений в Python оказалось меньше (сейчас текущей является версия 3.10, а на подходе 3.11), зато развитие pandas продолжилось.

В третьем издании я ставил целью привести материал в соответствие с текущими версиями Python, NumPy, pandas и другими проектами, не слишком увлекаясь обсуждением новых проектов на Python, появившихся за последние пять лет. Поскольку книга стала важным ресурсом для многих университетских курсов и практикующих профессионалов, я постараюсь избегать тем, которые рискуют устареть через год-другой. Это позволит сохранить актуальность печатных экземпляров в 2023–2024 годах и, возможно, в более длительной перспективе.

В третьем издании добавилось новшество – открытый доступ к онлайн-версии, размещенной на моем сайте <https://wesmckinney.com/book>. Это будет удобно для владельцев печатных и цифровых версий книги. Я собираюсь поддерживать текст в более-менее актуальном состоянии, поэтому если вы встретите в печатной версии что-то не работающее, как должно, всегда можно будет справиться с последними обновлениями.

ГРАФИЧЕСКИЕ ВЫДЕЛЕНИЯ

В книге применяются следующие графические выделения.

Курсив

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

Моноширинный

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

Моноширинный полужирный

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

Моноширинный курсив

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



Так обозначается совет или рекомендация.



Так обозначается замечание общего характера.



Так обозначается предупреждение или предостережение.

О ПРИМЕРАХ КОДА

Файлы данных и прочие материалы, организованные по главам, можно найти в репозитории книги на GitHub по адресу <http://github.com/wesm/pydata-book> или на его зеркале на Gitee (для тех, у кого нет доступа к GitHub) по адресу <https://gitee.com/wesmckinn/pydata-book>.

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

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Python for Data Analysis by Wes McKinney (O'Reilly). Copyright 2022 Wes McKinney, 78-1-098-10403-0».

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

КАК С НАМИ СВЯЗАТЬСЯ

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: http://bit.ly/python_data_analysis_3e.

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

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

Ищите нас в LinkedIn: <https://linkedin.com/company/oreilly-media>.

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

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

Благодарности

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

Памяти Джона Д. Хантера (1968–2012)

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

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

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

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

Благодарности к третьему изданию (2022)

Прошло уже больше десяти лет с тех пор, как я начал работать над первым изданием этой книги, и больше пятнадцати с момента, когда я начал карьеру

программиста на Python. За это время многое изменилось! Из нишевого языка для анализа данных Python превратился в самый популярный и распространенный язык, лежащий в основе значительной (если не подавляющей!) части науки о данных, машинного обучения и искусственного интеллекта.

Я не принимал активного участия в развитии проекта с открытым исходным кодом pandas с 2013 года, но сложившееся вокруг него всемирное сообщество разработчиков процветало и служило образцом разработки открытого ПО силами сообщества. Многие Python-проекты «следующего поколения» для работы с табличными данными строят пользовательские интерфейсы по образцу pandas, так что проект продолжает оказывать неослабевающее влияние на пути развития экосистемы науки о данных на Python.

Я надеюсь, что эта книга остается ценным подспорьем для студентов и всех, кто хочет узнать о работе с данными на Python.

Я очень признателен издательству O'Reilly, разрешившему мне опубликовать на моем сайте <https://wesmckinney.com/book> версию книги с «открытым доступом», которая, как я надеюсь, откроет мир анализа для еще большего количества людей. Дж. Дж. Аллэр (J. J. Allaire) оказался той палочкой-выручалочкой, благодаря которой мне удалось перенести текст книги из Docbook XML в Quarto (<https://quarto.org/>), замечательную новую систему подготовки научно-технических текстов для печати и публикации в вебе.

Отдельное спасибо техническим рецензентам Полу Бэрри (Paul Barry), Жану-Кристофу Лейдеру (Jean-Christophe Leyder), Абдулле Карасану (Abdullah Karasan) и Уильяму Джамиру (William Jamir) за их подробные отзывы, значительно улучшившие книгу с точки зрения удобочитаемости, ясности и понятности содержимого.

Благодарности ко второму изданию (2017)

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

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

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

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

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

Благодарности к первому изданию (2012)

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

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

В техническом рецензировании книги принимало участие много народу. Мартин Лас (Martin Blais) и Хью Браун (Hugh Brown) оказали неоценимую помощь в повышении качества примеров, ясности изложения и улучшении организации книги в целом. Джеймс Лонг (James Long), Дрю Конвей (Drew Conway), Фернандо Перес, Брайан Грейнджер, Томас Клуйвер (Thomas Kluyver), Адам Клейн, Джон Клейн, Чань Ши и Стефан ван дер Вальт (Stefan van der Walt) отрецензировали по одной или по нескольким главам и предложили ценные замечания с разных точек зрения.

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

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

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

Если говорить о личной жизни, то я благодарен Кэйси Динкин (Casey Dinkin), чью каждодневную поддержку невозможно переоценить, 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.

Предварительные сведения

1.1. О чем эта книга?

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

Спустя некоторое время после выхода этой книги в 2012 году термин «наука о данных» (data science) стали употреблять для всего на свете: от простой описательной статистики до более сложного статистического анализа и машинного обучения. Открытая экосистема для анализа данных (или науки о данных) на Python с тех пор также значительно расширилась. Сейчас имеется много книг, посвященных специально более продвинутым методологиям. Лыщу себя надеждой, что эта книга подготовит вас к изучению ресурсов, ориентированных на конкретные области применения.



Возможно, кто-то сочтет, что книга в большей степени посвящена «манипулированию данными», а не «анализу данных». Мы используем также термины «первичная обработка данных» (data wrangling) и «подготовка данных» (data munging) в качестве синонимов «манипулированию данными».

Какого рода данные?

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

- табличные данные, 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;
- многомерные списки (матрицы);

- данные, представленные в виде нескольких таблиц, связанных между собой по ключевым столбцам (то, что в SQL называется первичными и внешними ключами);
- равноотстоящие и неравноотстоящие временные ряды.

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 удастся, 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.

1.2. 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, поскольку он наводит на мысль, будто для создания ответственного программного обеспечения язык не годится. Из всех интерпретируемых языков Python выделяется большим и активным сообществом научных расчетов и анализа данных. За последние 20 лет Python перешел из разряда ультрасовременного языка научных расчетов, которым пользуются на свой страх и риск, в один из самых важных языков, применяемых в науке о данных, машинном обучении и разработке ПО общего назначения в академических учреждениях и промышленности.

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

санных 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 много лет программ.

Большинство программ 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.

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.

За последнее десятилетие появились новые подходы к решению проблемы «двух языков», в частности язык программирования Julia. Во многих случаях для получения максимальной пользы от Python *необходимо* программировать на языке более низкого уровня типа C или C++ и создавать интерфейс между таким кодом и Python. Вместе с тем технология «своевременных» (JIT) компиляторов, предлагаемая такими библиотеками, как Numba, позволила добиваться великолепной производительности многих численных алгоритмов, не покидая среду программирования на Python.

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 обработки 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. Написанные на C или C++ расширения Python, пользующиеся платформенной многопоточностью, могут исполнять код параллельно и не ограничены механизмом GIL, при условии что им не нужно регулярно взаимодействовать с Python-объектами.

1.3. НЕОБХОДИМЫЕ БИБЛИОТЕКИ ДЛЯ PYTHON

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

NumPy

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

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

Помимо 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 в другое представление. Таким образом, многие средства вычислений, ориентированные на Python, либо используют массивы NumPy в качестве основной структуры данных, либо каким-то иным способом организуют интеграцию с NumPy.

pandas

Библиотека pandas (<https://pandas.pydata.org/>) предоставляет структуры данных и функции, призванные сделать работу со структурированными данными

ми простым, быстрым и выразительным делом. С момента своего появления в 2010 году она способствовала превращению Python в мощную и продуктивную среду анализа данных. Основные объекты pandas, используемые в этой книге, – `DataFrame` – двумерная таблица, в которой строки и столбцы имеют метки, и `Series` – объект одномерного массива с метками.

Библиотека pandas сочетает высокую производительность средств работы с массивами, присущую NumPy, с гибкими возможностями манипулирования данными, 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. Поскольку манипулирование данными, их подготовка и очистка играют огромное значение в анализе данных, в этой книге библиотека pandas будет одним из основных инструментов.

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

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

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

Значительную часть 2011 и 2012 годов я потратил на расширение возможностей pandas, в чем мне помогали бывшие коллеги по компании AQR Адам Клейн (Adam Klein) и Чань Ше (Chang She). В 2013 году я отошел от ежедневной работы по развитию проекта, и pandas перешла в полную собственность сообщества, в которое входит свыше двух тысяч программистов со всего мира.

Пользователям языка статистических расчетов R название `DataFrame` покажется знакомым, потому что оно выбрано по аналогии с объектом `data.frame` в R. В отличие от Python, фреймы данных уже встроены в язык R и его стандартную

библиотеку. Поэтому многие средства, присутствующие в pandas, либо являются частью ядра R, либо предоставляются дополнительными пакетами.

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

matplotlib

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

IPython и Jupyter

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

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

Сам IPython стал компонентом более широкого проекта Jupyter с открытым исходным кодом, предоставляющего продуктивную среду для интерактивных исследовательских вычислений. В своем самом старом и простом «режиме» это улучшенная оболочка Python, имеющая целью ускорить написание, тестирование и отладку кода на Python. Систему IPython можно использовать также через Jupyter-блокнот.

Jupyter-блокноты позволяют создавать контент на языках разметки Markdown и HTML, т. е. готовить комбинированные документы, содержащие код и текст.

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

В сопроводительных материалах к книге (<https://github.com/wesm/pydata-book>) вы найдете Jupyter-блокноты, содержащие примеры кода к каждой главе. Если у вас нет доступа к GitHub, попробуйте зеркало на сайте Gitee (<https://gitee.com/wesmckinn/pydata-book>).

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.

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

scikit-learn

Проект scikit-learn (<https://scikit-learn.org/stable/>), запущенный в 2007 году, с самого начала стал основным инструментарием машинного обучения для программистов на Python. На момент написания книги сообщество насчитывает более 2000 разработчиков. В нем имеются подмодули для следующих моделей.

- Классификация: метод опорных векторов, метод ближайших соседей, случайные леса, логистическая регрессия и т. д.
- Регрессия: Lasso, гребневая регрессия и т. д.

- Кластеризация: метод k -средних, спектральная кластеризация и т. д.
- Понижение размерности: метод главных компонент, отбор признаков, матричная факторизация и т. д.
- Выбор модели: поиск на сетке, перекрестный контроль, метрики.
- Предобработка: выделение признаков, нормировка.

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

statsmodels

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

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

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

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

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

1.4. УСТАНОВКА И НАСТРОЙКА

Поскольку Python используется в самых разных приложениях, не существует единственно верной процедуры установки Python и необходимых дополнительных пакетов. У многих читателей, скорее всего, нет среды, подходящей для научных применений Python и проработки этой книги, поэтому я приведу подробные инструкции для разных операционных систем. Я буду использовать `Miniconda`, минимальный дистрибутив менеджера пакетов `conda`, а вмес-

те с ним conda-forge (<https://conda-forge.org/>), поддерживаемый сообществом дистрибутив, основанный на conda. В этой книге всюду используется версия Python 3.10, но если вам доведется читать ее в будущем, то ничто не мешает установить более новую версию.

Если по какой-то причине эти инструкции устареют к моменту чтения, загляните на сайт книги (<https://wesmckinney.com/book/>), где я постараюсь выкладывать актуальные инструкции по установке.

Miniconda в Windows

Чтобы начать работу в Windows, скачайте последний из имеющихся установщиков Miniconda (в настоящее время для версии Python 3.9) по адресу <https://conda.io>. Я рекомендую следовать инструкциям по установке для Windows на сайте conda, которые могли измениться за время, прошедшее с момента выхода книги. На большинстве компьютеров установлена 64-разрядная версия, но если на вашей машине она не заработает, можете установить 32-разрядную.

В ответ на вопрос, хотите ли вы установить программу только для себя или для всех пользователей системы, решите, что вам больше подходит. Установки для себя достаточно, чтобы выполнять приведенные в книге примеры. Кроме того, установщик спросит, нужно ли добавлять путь к Miniconda в системную переменную окружения PATH. Если вы ответите утвердительно (я обычно так и делаю), то данная установка Miniconda может переопределить другие установленные версии Python. Если нет, то для запуска Miniconda нужно будет использовать добавленную в меню **Пуск** ссылку. Соответствующий пункт в меню **Пуск** может называться как-то вроде «Anaconda3 (64-bit)».

Я буду предполагать, что вы не добавляли Miniconda в системную переменную PATH. Чтобы убедиться, что все настроено правильно, щелкните по пункту «Anaconda Prompt (Miniconda3)» под пунктом «Anaconda3 (64-bit)» в меню **Пуск**. Затем попробуйте запустить интерпретатор Python, набрав команду **python**. Должно появиться сообщение вида

```
(base) C:\Users\Wes>python
Python 3.9 [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Чтобы выйти из оболочки Python, введите команду **exit()** и нажмите **Enter**.

GNU/Linux

Детали установки в Linux варьируются в зависимости от дистрибутива, я опишу процедуру, работающую в дистрибутивах Debian, Ubuntu, CentOS и Fedora. Установка в основных чертах производится так же, как для macOS, отличается только порядок установки Miniconda. Большая часть читателей, вероятно, захочет скачать предлагаемый по умолчанию 64-разрядный файл установщика, предназначенный для архитектуры x86 (но вполне возможно, что в будущем у большинства пользователей будут машины под управлением Linux с архитектурой aarch64). Установщик представляет собой скрипт оболочки, запускаемый из терминала. Имя соответствующего файла имеет вид *Miniconda3-latest-Linux-x86_64.sh*. Для установки нужно выполнить такую команду:

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```



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

Вам будет предложено указать место установки файлов Miniconda. Я рекомендую устанавливать их в свой домашний каталог, например `/home/$USER/miniconda` (вместо `$USER` подставьте свое имя пользователя).

Установщик спросит, хотите ли вы модифицировать скрипты оболочки, так чтобы Miniconda активировалась автоматически. Я рекомендую так и поступить (выберите «yes»), поскольку это удобнее.

По завершении установки запустите новый процесс терминала и убедитесь, что выбирается новая версия Miniconda:

```
(base) $ python
Python 3.9 | (main) [GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Чтобы выйти из оболочки, нажмите **Ctrl-D** или введите команду `exit()` и нажмите **Enter**.

Miniconda в macOS

Скачайте установщик Miniconda для macOS, для компьютеров на базе систем серии Apple Silicon, выпущенных начиная с 2020 года, он должен называться как-то вроде *Miniconda3-latest-MacOSX-arm64.sh*, а для компьютеров с процессором Intel, выпущенных до 2020 года, – *Miniconda3-latest-MacOSX-x86_64.sh*. Откройте приложение **Терминал** и выполните установщик (скорее всего, находится в вашем каталоге [Downloads](#)) с помощью `bash`:

```
$ bash $HOME/Downloads/Miniconda3-latest-MacOSX-arm64.sh
```

В процессе работы установщик по умолчанию конфигурирует Miniconda в вашем профиле оболочки, подразумеваемом по умолчанию. Скорее всего, это файл `/Users/$USER/.zshrc`. Я рекомендую не возражать против этого действия. Если же вы категорически не хотите разрешать установщику модификацию своей среды, то почитайте документацию Miniconda, прежде чем продолжать.

Для проверки работоспособности попробуйте запустить Python из системной оболочки (для получения командной строки откройте приложение **Терминал**):

```
$ python
Python 3.9 (main) [Clang 12.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Чтобы выйти из оболочки, нажмите **Ctrl-D** или введите команду `exit()` и нажмите **Enter**.

Установка необходимых пакетов

Теперь, когда менеджер Miniconda установлен, самое время установить основные пакеты, которые понадобятся нам в этой книге. Первый шаг – сделать conda-forge подразумеваемым по умолчанию каналом получения пакетов. Для этого выполните в оболочке следующие команды:

```
(base) $ conda config --add channels conda-forge
(base) $ conda config --set channel_priority strict
```

Далее создадим новую «среду» conda для Python 3.10 командой `conda create`:

```
(base) $ conda create -y -n pydata-book python=3.10
```

По завершении установки активируйте среду командой `conda activate`:

```
(base) $ conda activate pydata-book
(pydata-book) $
```



Команду `conda activate` нужно использовать при каждом открытии нового терминала. В любой момент можно получить информацию об активной среде conda, выполнив в терминале команду `conda info`.

Далее установим необходимые пакеты (вместе с их зависимостями) командой `conda install`:

```
(pydata-book) $ conda install -y pandas jupyter matplotlib
```

Мы будем использовать и некоторые другие пакеты, но установить их можно позже. Есть два способа установки пакетов: командами `conda install` и `pip install`. При работе с Miniconda предпочтительнее использовать `conda install`, но некоторые пакеты через conda недоступны, поэтому если `conda install $package_name` не работает, попробуйте `$package_name`.



Если вы хотите сразу установить все пакеты, которые нам понадобятся по ходу дела, то можете выполнить такую команду:

```
conda install lxml beautifulsoup4 html5lib openpyxl \
requests sqlalchemy seaborn scipy statsmodels \
patsy scikit-learn pyarrow pytables numba
```

В Windows в качестве символа продолжения строки используйте знак крышки ^ вместо знака \, применяемого в Linux и macOS.

Для обновления пакетов служит команда `conda update`:

```
conda update package_name
```

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

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

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



Если для установки пакетов вы используете и `conda`, и `pip`, то не следует пытаться обновлять пакеты `conda` с помощью `pip` (и наоборот), поскольку это может привести к повреждению среды. Я рекомендую сначала всегда пробовать команду `conda install` и прибегать к `pip`, только если установить пакет не получается.

Интегрированные среды разработки (IDE)

Когда меня спрашивают, какой средой разработки я пользуюсь, я почти всегда отвечаю: «IPython плюс текстовый редактор». Обычно я пишу программу и итеративно тестирую и отлаживаю ее по частям в IPython или Jupyter-блокнотах. Полезно также иметь возможность интерактивно 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 некоторые пользователи предпочитают разрабатывать программы в полноценной IDE, а не в сравнительно примитивном текстовом редакторе типа Emacs или Vim. Вот некоторые доступные варианты:

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

Благодаря популярности Python большинство текстовых редакторов, в частности VS Code и Sublime Text 2, обзавелись прекрасной поддержкой для него.

1.5. Сообщество и конференции

Помимо поиска в интернете, существуют полезные списки рассылки, посвященные использованию Python в научных расчетах и для обработки данных. Их участники быстро отвечают на вопросы. Вот некоторые из таких ресурсов:

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

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

Ежегодно в разных странах проводят конференции для программистов на Python. Если вы захотите пообщаться с другими программистами на Python, которые разделяют ваши интересы, то имеет смысл посетить какую-нибудь

(если есть такая возможность). Многие конференции оказывают финансовую поддержку тем, кто не может позволить себе вступительный взнос или транспортные расходы. Приведу неполный перечень конференций.

- PyCon and EuroPython: две самые крупные, проходящие соответственно в Северной Америке и в Европе.
- SciPy и EuroSciPy: конференции, ориентированные на научные применения Python, проходящие соответственно в Северной Америке и в Европе.
- PyData: мировая серия региональных конференций, посвященных науке о данных и анализу данных.
- Международные и региональные конференции PyCon (полный список см. на сайте <http://pycon.org>).

1.6. СТРУКТУРА КНИГИ

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

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

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

Взаимодействие с внешним миром

Чтение и запись в файлы и хранилища данных различных форматов.

Подготовка

Очистка, переформатирование, комбинирование, нормализация, изменение формы, получение продольных и поперечных срезов, трансформация данных для анализа.

Преобразование

Применение математических и статистических операций к группам наборов данных для получения новых наборов (например, агрегирование большой таблицы по некоторым переменным).

Моделирование и вычисления

Связывание данных со статистическими моделями, алгоритмами машинного обучения и иными вычислительными средствами.

Презентация

Создание интерактивных или статических графических визуализаций или текстовых сводных отчетов.

Примеры кода

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

```
In [5]: КОД
Out[5]: РЕЗУЛЬТАТ
```

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

Я изменил параметры вывода на консоль, подразумеваемые по умолчанию в NumPy и pandas, ради краткости и удобочитаемости. Так, числовые данные печатаются с большей точностью. Чтобы при выполнении примеров результаты выглядели так же, как в книге, выполните следующий Python-код перед запуском примеров:

```
import numpy as np
import pandas as pd
pd.options.display.max_columns = 20
pd.options.display.max_rows = 20
pd.options.display.max_colwidth = 80
np.set_printoptions(precision=4, suppress=True)
```

Данные для примеров

Наборы данных для примеров из каждой главы находятся в репозитории на сайте GitHub: <https://github.com/wesm/pydata-book> (или его зеркале на Gitee по адресу <https://gitee.com/wesmckinn/pydata-book>, если у вас нет доступа к GitHub). Вы можете получить их либо с помощью командной утилиты системы управления версиями Git, либо скачав zip-файл репозитория с сайта. Если возникнут проблемы, заходите на мой сайт (<https://wesmckinney.com/book>), где выложены актуальные инструкции по получению материалов к книге.

Скачав zip-файл с примерами наборов данных, вы должны будете распаковать его в какой-нибудь каталог и перейти туда в терминале, прежде чем выполнять примеры:

```
$ pwd
/home/wesm/book-materials
$ ls
appa.ipynb ch05.ipynb ch09.ipynb ch13.ipynb README.md
ch02.ipynb ch06.ipynb ch10.ipynb COPYING requirements.txt
ch03.ipynb ch07.ipynb ch11.ipynb datasets
ch04.ipynb ch08.ipynb ch12.ipynb examples
```

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 book@wesmckinney.com. Самый лучший способ сообщить об ошибках, найденных в книге, – описать их на странице опечаток на сайте издательства O’Reilly (<https://www.oreilly.com/catalog/errata.csp?isbn=0636920519829>).

Соглашения об импорте

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

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

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

Основы языка Python, IPython и Jupyter-блокноты

В 2011 и 2012 годах, когда я писал первое издание этой книги, ресурсов для изучения анализа данных с применением Python было гораздо меньше. Тут мы имеем что-то похожее на проблему яйца и курицы: многие библиотеки, наличие которых мы сейчас считаем само собой разумеющимся, в т. ч. `pandas`, `scikit-learn` и `statsmodels`, тогда были еще относительно незрелыми. Ныне, в 2022 году, количество литературы по науке о данных, анализу данных и машинному обучению неуклонно растет, дополняя прежние работы по научным расчетам, рассчитанные на специалистов по информатике, физике и другим дисциплинам. Есть также замечательные книги о самом языке программирования Python и о том, как стать эффективным программистом.

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

Книга в основном посвящена инструментам табличного анализа и подготовки данных для работы с наборами данных, помещающимися на персональном компьютере. Чтобы применить эти инструменты, зачастую необходимо сначала преобразовать беспорядочные данные низкого качества в более удобную табличную (или *структурную*) форму. К счастью, Python – идеальный язык для этой цели. Чем свободнее вы владеете языком и встроенными в него типами данных, тем проще будет подготовить новый набор данных для анализа.

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



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

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

2.1. ИНТЕРПРЕТАТОР PYTHON

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

```
$ python
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

Строка `>>>` – это приглашение к вводу выражения. Для выхода из интерпретатора Python нужно либо ввести команду `exit()`, либо нажать `Ctrl-D` (только в Linux и macOS).

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

```
print("Hello world")
```

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

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

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

¹ Лусиану Рамальо. Python. К вершинам мастерства. 2-е изд. ДМК Пресс, 2022.

² Бретт Слаткин. Секреты Python. Вильямс, 2017.

```
$ ipython
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.
```

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

```
In [2]:
```

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

2.2. Основы IPython

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

Запуск оболочки IPython

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

```
$ ipython
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]: a = 5
```

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

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

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

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

In [7]: data
Out[7]:
[-0.20470765948471295,
 0.47894333805754824,
-0.5194387150567381,
-0.55573030434749,
 1.9657805725027142,
 1.3934058329729904,
 0.09290787674371767]
```

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

Многие объекты Python форматируются для удобства чтения; такая *красивая печать* отличается от обычного представления методом `print`. Тот же сло-

варь `data`, напечатанный в стандартном интерпретаторе Python, выглядел бы куда менее презентабельно:

```
>>> import numpy as np
>>> data = [np.random.standard_normal() for i in range(7)]
>>> print(data)
>>> data
[-0.5767699931966723, -0.1010317773535111, -1.7841005313329152,
-1.524392126408841, 0.22191374220117385, -1.9835710588082562,
-1.6081963964963528]
```

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

Запуск Jupyter-блокнота

Одним из основных компонентов Jupyter-проекта является *блокнот* – интерактивный документ, содержащий код, текст (простой или размеченный), визуализации и другие результаты выполнения кода. Jupyter-блокнот взаимодействует с *ядрами* – реализациями протокола интерактивных вычислений на различных языках программирования. В ядре Jupyter для Python в качестве основы используется IPython.

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

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4bb443a63f2d...
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
To access the notebook, open this file in a browser:
file:///home/wesm/.local/share/jupyter/runtime/nbserver-185259-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4...
or http://127.0.0.1:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4...
```

На многих платформах Jupyter автоматически открывается в браузере по умолчанию (если только при запуске не был указан флаг `--no-browser`). Если это не так, то можете сами ввести URL при запуске блокнота, в данном случае <http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4bb443a63f2d3055>. На рис. 2.1 показано, как выглядит блокнот в браузере Google Chrome.



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

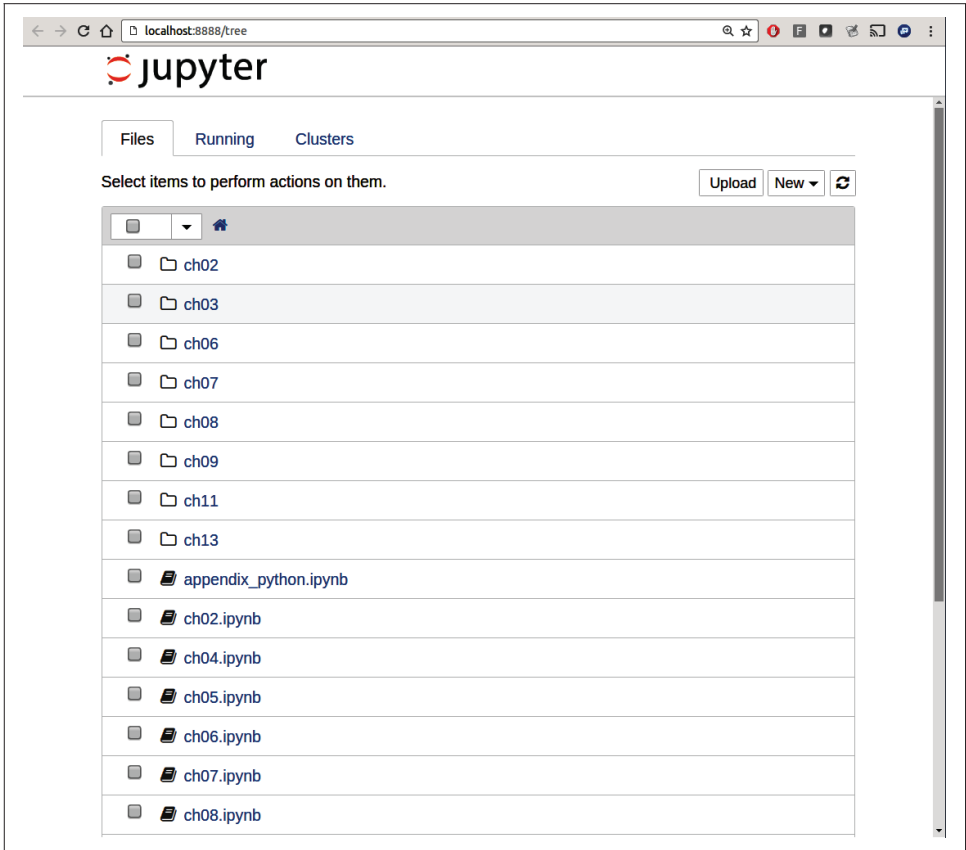


Рис. 2.1. Начальная страница Jupyter-блокнота

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

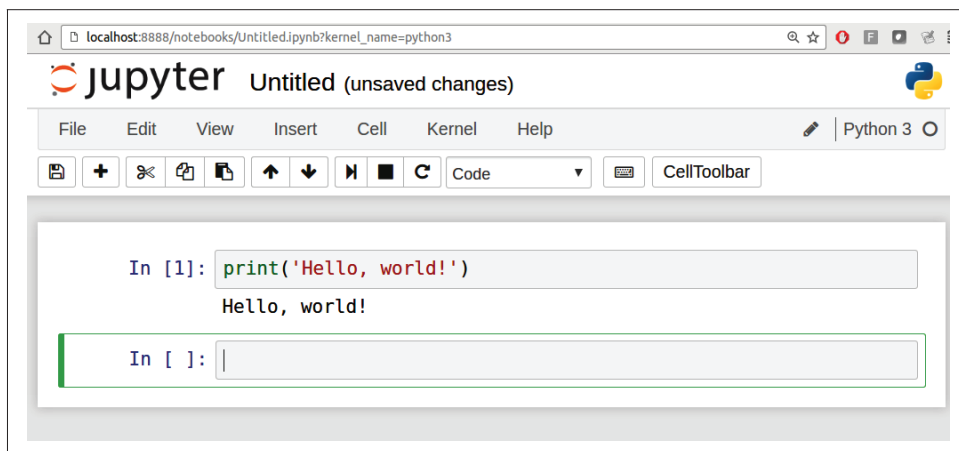


Рис. 2.2. Так выглядит новый Jupyter-блокнот

После сохранения блокнота (команда «Save and Checkpoint» в меню **File**) будет создан файл с расширением *.ipynb*. В нем содержится все, что сейчас находится в блокноте (включая все результаты выполнения кода).

Чтобы загрузить существующий блокнот, поместите файл в тот каталог, из которого был запущен блокнот (или в его подкаталог), и дважды щелкните по имени файла на начальной странице. Можете попробовать проделать это с моими блокнотами, находящимися в репозитории *wesm/pydata-book* на GitHub. См. рис. 2.3.

Когда захотите закрыть блокнот, выберите команду «Close and Halt» из меню **File**. Если вы просто закроете вкладку браузера, то ассоциированный с блокнотом процесс Python продолжит работать в фоновом режиме.

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

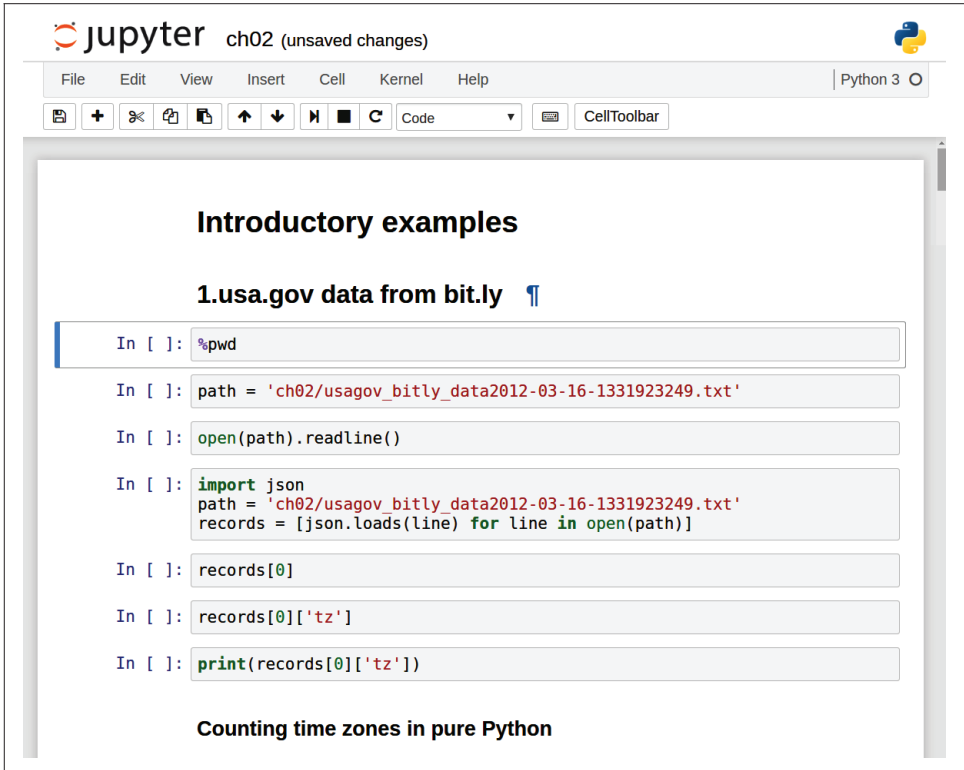


Рис. 2.3. Пример существующего Jupyter-блокнота

Завершение по нажатии клавиши Tab

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

```
In [1]: an_apple = 27

In [2]: an_example = 42

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

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


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

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

То же самое относится и к модулям:

```
In [1]: import datetime
```

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



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

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

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 `=`). См. рис. 2.4.

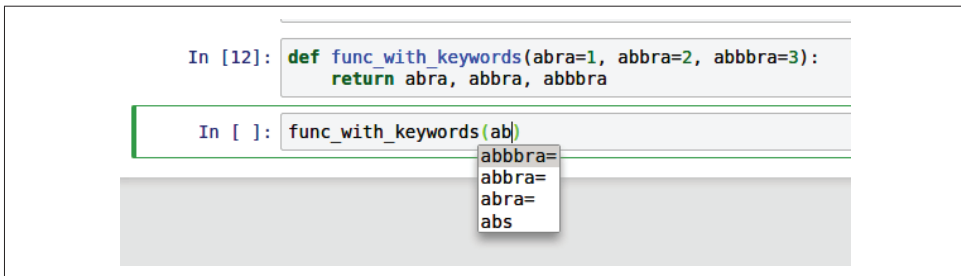


Рис. 2.4. Автозавершение ключевых аргументов функции в Jupyter-блокноте

Ниже мы еще поговорим о функциях.

Интроспекция

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

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

```
In [2]: b?
Type: list
String form: [1, 2, 3]
Length: 3
Docstring:
Built-in mutable sequence.
```

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

```
In [3]: print?
```

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

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

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

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

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

Тогда при вводе знака ? мы увидим строку документации:

```
In [6]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Сложить два числа
Возвращает
-----
the_sum : типа аргументов
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

И последнее применение ? – поиск в пространстве имен IPython по аналогии со стандартной командной строкой UNIX или Windows. Если ввести несколько

символов в сочетании с метасимволом `*`, то будут показаны все имена по указанной маске. Например, вот как можно получить список всех функций в пространстве имен верхнего уровня NumPy, имена которых содержат строку `load`:

```
In [9]: import numpy as np
```

```
In [10]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
```

2.3. Основы языка Python

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

Семантика языка

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

Отступы вместо скобок

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

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

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

Нравится вам это или нет, но синтаксически значимые пробелы – факт, с которым программисты на Python должны смириться. Поначалу такой стиль может показаться чужеродным, но со временем вы привыкнете и полюбите его.



Я настоятельно рекомендую использовать 4 пробела в качестве величины отступа по умолчанию и настроить редактор так, чтобы он заменял знаки табуляции четырьмя пробелами. IPython и Jupyter-блокноты автоматически вставляют четыре пробела в новой строке после двоеточия и заменяют знаки табуляции четырьмя пробелами.

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

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

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

Всё является объектом

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

Комментарии

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

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

Комментарии могут также встречаться после строки исполняемого кода. Некоторые программисты предпочитают располагать комментарий над строкой кода, к которой он относится, но и такой стиль временами бывает полезен:

```
print("Reached this line") # Простое сообщение о состоянии
```

Вызов функции и метода объекта

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

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

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

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

Функции могут принимать позиционные и именованные аргументы:

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

Подробнее об этом ниже.

Переменные и передача аргументов

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

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

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

```
In [9]: b = a
```

```
In [10]: b
```

```
Out[10]: [1, 2, 3]
```

В некоторых языках такое присваивание приводит к копированию данных `[1, 2, 3]`. В Python `a` и `b` указывают на один и тот же объект – исходный список `[1, 2, 3]` (это схематически изображено на рис. 2.5). Чтобы убедиться в этом, добавим в список `a` еще один элемент и проверим затем список `b`:

```
In [11]: a.append(4)
```

```
In [12]: b
```

```
Out[12]: [1, 2, 3, 4]
```

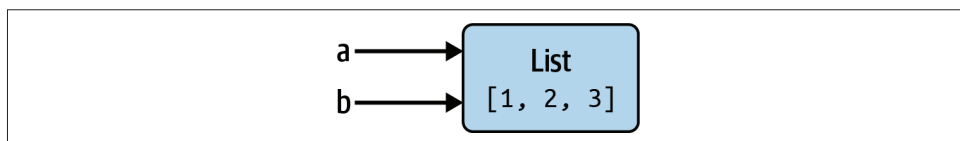


Рис. 2.5. Две ссылки на один объект

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



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

Когда объекты передаются функции в качестве аргументов, создаются новые локальные переменные, ссылающиеся на исходные объекты, – копирование не производится. Если новый объект связывается с переменной внутри функции, то переменная с таким же именем в «области видимости» вне этой функции («родительской области видимости») не перезаписывается. Поэтому функция может модифицировать внутреннее содержимое изменяемого аргумента. Пусть имеется такая функция:

```
In [13]: def append_element(some_list, element):
        ....:     some_list.append(element)
```

Тогда:

```
In [14]: data = [1, 2, 3]
```

```
In [15]: append_element(data, 4)
```

```
In [16]: data
```

```
Out[16]: [1, 2, 3, 4]
```

Динамические ссылки, строгие типы

С переменными в Python не связан никакой тип; достаточно выполнить присваивание, чтобы переменная стала указывать на объект другого типа. Следующий код не приведет к ошибке:

```
In [17]: a = 5

In [18]: type(a)
Out[18]: int

In [19]: a = "foo"

In [20]: type(a)
Out[20]: str
```

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

```
In [21]: "5" + 5
-----
TypeError                                Traceback (most recent call last)
<ipython-input-21-7fe5aa79f268> in <module>
----> 1 "5" + 5
TypeError: can only concatenate str (not "int") to str
```

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

```
In [22]: a = 4.5

In [23]: b = 2

# Форматирование строки, объясняется ниже
In [24]: print(f"a is {type(a)}, b is {type(b)}")
a is <class 'float'>, b is <class 'int'>

In [25]: a / b
Out[25]: 2.25
```

Здесь `b` – целое число, но оно неявно преобразуется в тип с плавающей точкой, прежде чем выполнить операцию деления.

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

```
In [26]: a = 5

In [27]: isinstance(a, int)
Out[27]: True
```

Функция `isinstance` может также принимать кортеж типов и тогда проверяет, что тип переданного объекта присутствует в кортеже:

```
In [28]: a = 5; b = 4.5

In [29]: isinstance(a, (int, float))
Out[29]: True

In [30]: isinstance(b, (int, float))
Out[30]: True
```

Атрибуты и методы

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

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

In [2]: a.<Press Tab>
capitalize() index() isspace() removesuffix() startswith()
casefold() isprintable() istitle() replace() strip()
center() isalnum() isupper() rfind() swapcase()
count() isalpha() join() rindex() title()
encode() isascii() ljust() rjust() translate()
endswith() isdecimal() lower() rpartition()
expandtabs() isdigit() lstrip() rsplit()
find() isidentifier() maketrans() rstrip()
format() islower() partition() split()
format_map() isnumeric() removeprefix() splitlines()
```

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

```
In [32]: getattr(a, "split")
Out[32]: <function str.split(sep=None, maxsplit=-1)>
```

Хотя в этой книге мы почти не используем функцию `getattr`, а также родственные ей `hasattr` и `setattr`, они весьма эффективны для написания обобщенного, повторно используемого кода.

Утиная типизация

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

```
In [33]: def isiterable(obj):
.....:     try:
.....:         iter(obj)
.....:         return True
.....:     except TypeError: # не допускает итерирования
.....:         return False
```

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

```
In [34]: iterable("a string")
Out[34]: True

In [35]: iterable([1, 2, 3])
Out[35]: True

In [36]: iterable(5)
Out[36]: False
```

Импорт

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

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

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

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

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

Или эквивалентно:

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

Ключевое слово `as` позволяет переименовать импортированные сущности:

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

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

Бинарные операторы и операции сравнения

В большинстве бинарных математических операций и операций сравнения используется такой же синтаксис, как в других языках программирования:

```
In [37]: 5 - 7
Out[37]: -2

In [38]: 12 + 21.5
Out[38]: 33.5

In [39]: 5 <= 2
Out[39]: False
```


Список всех бинарных операторов приведен в табл. 2.1.

Таблица 2.1. Бинарные операторы

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

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

```
In [40]: a = [1, 2, 3]
```

```
In [41]: b = a
```

```
In [42]: c = list(a)
```

```
In [43]: a is b
```

```
Out[43]: True
```

```
In [44]: a is not c
```

```
Out[44]: True
```

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

```
In [45]: a == c
Out[45]: True
```

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

```
In [46]: a = None

In [47]: a is None
Out[47]: True
```

Изменяемые и неизменяемые объекты

Многие объекты в Python – списки, словари, массивы NumPy и почти все определенные пользователем типы (классы) – *изменяемы*. Это означает, что объект или значения, которые в нем хранятся, можно модифицировать.

```
In [48]: a_list = ["foo", 2, [4, 5]]

In [49]: a_list[2] = (3, 4)

In [50]: a_list
Out[50]: ['foo', 2, (3, 4)]
```

Но некоторые объекты, например строки и кортежи, неизменяемы, т. е. их внутренние данные нельзя модифицировать:

```
In [51]: a_tuple = (3, 5, (4, 5))

In [52]: a_tuple[1] = "four"
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-52-cd2a018a7529> in <module>
----> 1 a_tuple[1] = "four"
TypeError: 'tuple' object does not support item assignment
```

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

Скалярные типы

В Python есть небольшой набор встроенных типов для работы с числовыми данными, строками, булевыми значениями (`True` и `False`), датами и временем. Эти типы «с одним значением» иногда называются скалярными, и мы будем называть их просто скалярами. Перечень основных скалярных типов приведен в табл. 2.2. Работа с датами и временем будет рассмотрена отдельно, потому что эти типы определены в стандартном модуле `datetime`.

Таблица 2.2. Стандартные скалярные типы в Python

Тип	Описание
<code>None</code>	Значение «null» в Python (существует только один экземпляр объекта <code>None</code>)
<code>str</code>	Тип строки. Может содержать любые символы Unicode
<code>bytes</code>	Неинтерпретируемые двоичные данные
<code>float</code>	Число с плавающей точкой двойной точности (отдельный тип <code>double</code> не предусмотрен)
<code>bool</code>	Булево значение <code>True</code> или <code>False</code>
<code>int</code>	Целое с произвольной точностью

Числовые типы

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

```
In [53]: ival = 17239871

In [54]: ival ** 6
Out[54]: 26254519291092456596965462913230729701102721
```

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

```
In [55]: fval = 7.243

In [56]: fval2 = 6.78e-5
```

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

```
In [57]: 3 / 2
Out[57]: 1.5
```

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

```
In [58]: 3 // 2
Out[58]: 1
```

Строки

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

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

В Python типом строки является `str`.

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

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

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

```
In [60]: c.count("\n")
Out[60]: 3
```

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

```
In [61]: a = "this is a string"
In [62]: a[10] = "f"
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-62-3b2d95f10db4> in <module>
----> 1 a[10] = "f"
TypeError: 'str' object does not support item assignment
```

Для интерпретации этого сообщения об ошибке читайте снизу вверх. Мы пытались заменить символ («item») в позиции 10 буквой «f», но для строковых объектов это запрещено. Если нужно модифицировать строку, то следует использовать функцию или метод, создающие новую строку, например `replace`:

```
In [63]: b = a.replace("string", "longer string")

In [64]: b
Out[64]: 'this is a longer string'
```

После этой операции переменная `a` не изменилась:

```
In [65]: a
Out[65]: 'this is a string'
```

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

```
In [66]: a = 5.6

In [67]: s = str(a)

In [68]: print(s)
5.6
```

Строки – это последовательности символов Unicode и потому могут рассматриваться как любые другие последовательности, например списки или кортежи:

```
In [69]: s = "python"
```

```
In [70]: list(s)
```

```
Out[70]: ['p', 'y', 't', 'h', 'o', 'n']
```

```
In [71]: s[:3]
```

```
Out[71]: 'pyt'
```

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

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

```
In [72]: s = "12\\34"
```

```
In [73]: print(s)
```

```
12\34
```

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

```
In [74]: s = r"this\has\no\special\characters"
```

```
In [75]: s
```

```
Out[75]: 'this\\has\\no\\special\\characters'
```

Буква `r` здесь – сокращение от *raw*.

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

```
In [76]: a = "this is the first half "
```

```
In [77]: b = "and this is the second half"
```

```
In [78]: a + b
```

```
Out[78]: 'this is the first half and this is the second half'
```

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

```
In [79]: template = "{0:.2f} {1:s} are worth US${2:d}"
```

Здесь:

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

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

```
In [80]: template.format(88.46, "Argentine Pesos", 1)
Out[80]: '88.46 Argentine Pesos are worth US$1'
```

В Python 3.6 были добавлены f-строки (сокращение от «форматированные строковые литералы»), благодаря которым создание форматированных строк стало еще удобнее. Чтобы создать строку, поместите букву `f` непосредственно перед строковым литералом. А внутри строки заключайте выражения Python в фигурные скобки, тогда вместо них в строку будет подставлено значение выражения:

```
In [81]: amount = 10

In [82]: rate = 88.46

In [83]: currency = "Pesos"

In [84]: result = f"{amount} {currency} is worth US${amount / rate}"
```

После каждого выражения можно поставить спецификатор формата – в таком же синтаксисе, как в шаблонах строк выше:

```
In [85]: f"{amount} {currency} is worth US${amount / rate:.2f}"
Out[85]: '10 Pesos is worth US$0.11'
```

Форматирование строк – обширная тема; существует несколько методов и многочисленные параметры и ухищрения, призванные контролировать, как именно должны форматироваться значения, подставляемые в результирующую строку. Подробные сведения можно найти в официальной документации по Python (<https://docs.python.org/3/library/string.html>).

Байты и Unicode

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

```
In [86]: val = "español"

In [87]: val
Out[87]: 'español'
```

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

```
In [88]: val_utf8 = val.encode("utf-8")

In [89]: val_utf8
Out[89]: b'espa\xcc3\xbb1ol'

In [90]: type(val_utf8)
Out[90]: bytes
```

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

```
In [91]: val_utf8.decode("utf-8")
Out[91]: 'español'
```

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

```
In [92]: val.encode("latin1")
Out[92]: b'espa\xfaol'
```

```
In [93]: val.encode("utf-16")
Out[93]: b'\xff\xfe\x00s\x00p\x00a\x00\xfa\x00o\x00l\x00'
```

```
In [94]: val.encode("utf-16le")
Out[94]: b'e\x00s\x00p\x00a\x00\xfa\x00o\x00l\x00'
```

Чаще всего объекты типа `bytes` встречаются при работе с файлами, когда не-явно перекодировать все данные в Unicode-строки не всегда желательно.

Булевы значения

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

```
In [95]: True and True
Out[95]: True
```

```
In [96]: False or True
Out[96]: True
```

В процессе преобразования в число `False` становится равным 0, а `True` — 1:

```
In [97]: int(False)
Out[97]: 0
```

```
In [98]: int(True)
Out[98]: 1
```

Ключевое слово `not` переводит `True` в `False` и наоборот:

```
In [99]: a = True
```

```
In [100]: b = False
```

```
In [101]: not a
Out[101]: False
```

```
In [102]: not b
Out[102]: True
```

Приведение типов

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

```

In [103]: s = "3.14159"

In [104]: fval = float(s)

In [105]: type(fval)
Out[105]: float

In [106]: int(fval)
Out[106]: 3

In [107]: bool(fval)
Out[107]: True

In [108]: bool(0)
Out[108]: False

```

Отметим, что ненулевые значения после приведения к типу `bool` чаще всего становятся равны `True`.

Тип None

`None` – это тип, позволяющий записать значение `null` в Python.

```

In [109]: a = None

In [110]: a is None
Out[110]: True

In [111]: b = 5

In [112]: b is not None
Out[112]: True

```

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

```

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

    if c is not None:
        result = result * c

    return result

```

Дата и время

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

```

In [113]: from datetime import datetime, date, time

In [114]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [115]: dt.day
Out[115]: 29

In [116]: dt.minute
Out[116]: 30

```


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

```
In [117]: dt.date()
Out[117]: datetime.date(2011, 10, 29)
```

```
In [118]: dt.time()
Out[118]: datetime.time(20, 30, 21)
```

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

```
In [119]: dt.strftime("%Y-%m-%d %H:%M")
Out[119]: '2011-10-29 20:30'
```

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

```
In [120]: datetime.strptime("20091031", "%Y%m%d")
Out[120]: datetime.datetime(2009, 10, 31, 0, 0)
```

В табл. 11.2 приведен полный перечень спецификаций формата.

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

```
In [121]: dt_hour = dt.replace(minute=0, second=0)
```

```
In [122]: dt_hour
Out[122]: datetime.datetime(2011, 10, 29, 20, 0)
```

Поскольку тип `datetime.datetime` неизменяемый, эти и другие подобные методы порождают новые объекты. Так, в предыдущем примере объект `dt` не изменится в результате применения метода `replace`:

```
In [123]: dt
Out[123]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

Результат вычитания объектов `datetime` является объект типа `datetime.timedelta`:

```
In [124]: dt2 = datetime(2011, 11, 15, 22, 30)
```

```
In [125]: delta = dt2 - dt
```

```
In [126]: delta
Out[126]: datetime.timedelta(days=17, seconds=7179)
```

```
In [127]: type(delta)
Out[127]: datetime.timedelta
```

Результат `timedelta(17, 7179)` показывает, что в `timedelta` закодировано смещение 17 дней и 7179 секунд.

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

```
In [128]: dt
Out[128]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
In [129]: dt + delta
Out[129]: datetime.datetime(2011, 11, 15, 22, 30)
```

Поток управления

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

if, elif, else

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

```
x = -5
if x < 0:
    print("Отрицательно")
```

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

```
if x < 0:
    print("Отрицательно")
elif x == 0:
    print("Равно нулю")
elif 0 < x < 5:
    print("Положительно, но меньше 5")
else:
    print("Положительно и больше или равно 5")
```

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

```
In [130]: a = 5; b = 7

In [131]: c = 8; d = 4

In [132]: if a < b or c > d:
.....:     print("Сделано")
Сделано
```

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

Можно также строить цепочки сравнений:

```
In [133]: 4 > 3 > 2 > 1
Out[133]: True
```

Циклы for

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

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

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

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

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

```
In [134]: for i in range(4):
.....:     for j in range(4):
.....:         if j > i:
.....:             break
.....:         print((i, j))
.....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

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

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

Циклы while

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

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

Ключевое слово pass

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

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

Функция range

Функция `range` порождает последовательность равноотстоящих целых чисел:

```
In [135]: range(10)
Out[135]: range(0, 10)

In [136]: list(range(10))
Out[136]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
In [137]: list(range(0, 20, 2))
Out[137]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [138]: list(range(5, 0, -1))
Out[138]: [5, 4, 3, 2, 1]
```

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

```
In [139]: seq = [1, 2, 3, 4]

In [140]: for i in range(len(seq)):
.....:     print(f"элемент {i}: {seq[i]}")
элемент 0: 1
элемент 1: 2
элемент 2: 3
элемент 3: 4
```

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

```
In [141]: total = 0

In [142]: for i in range(100_000):
.....:     # % - оператор деления по модулю
.....:     if i % 3 == 0 or i % 5 == 0:
.....:         total += i

In [143]: print(total)
2333316668
```

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

2.4. ЗАКЛЮЧЕНИЕ

В этой главе было приведено краткое введение в языковые конструкции Python и в среды программирования IPython и Jupyter. В следующей главе мы обсудим многочисленные встроенные типы данных, функции и средства ввода-вывода, которыми будем пользоваться на протяжении всей книги.

Встроенные структуры данных, функции и файлы

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

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

3.1. СТРУКТУРЫ ДАННЫХ И ПОСЛЕДОВАТЕЛЬНОСТИ

Структуры данных в Python просты, но эффективны. Чтобы стать хорошим программистом на Python, необходимо овладеть ими в совершенстве. Мы начнем с кортежа, списка и словаря – наиболее часто используемых типов *последовательностей*.

Кортеж

Кортеж – это одномерная неизменяемая последовательность объектов Python фиксированной длины, которую нельзя изменить после первоначального присваивания. Проще всего создать кортеж, записав последовательность значений через запятую:

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

```
In [3]: tup  
Out[3]: (4, 5, 6)
```

Во многих контекстах скобки можно опускать, так что предыдущий пример можно было бы записать и так:

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

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

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

```
In [6]: tuple([4, 0, 2])
Out[6]: (4, 0, 2)

In [7]: tup = tuple('string')

In [8]: tup
Out[8]: ('s', 't', 'r', 'i', 'n', 'g')
```

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

```
In [9]: tup[0]
Out[9]: 's'
```

Когда кортеж определяется в составе более сложного выражения, часто бывает необходимо заключать значения в скобки, как в следующем примере, где создается кортеж кортежей:

```
In [10]: nested_tup = (4, 5, 6), (7, 8)

In [11]: nested_tup
Out[11]: ((4, 5, 6), (7, 8))

In [12]: nested_tup[0]
Out[12]: (4, 5, 6)

In [13]: nested_tup[1]
Out[13]: (7, 8)
```

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

```
In [14]: tup = tuple(['foo', [1, 2], True])

In [15]: tup[2] = False
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-15-b89d0c4ae599> in <module>
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

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

```
In [16]: tup[1].append(3)

In [17]: tup
Out[17]: ('foo', [1, 2, 3], True)
```

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

```
In [18]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[18]: (4, None, 'foo', 6, 0, 'bar')
```

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

```
In [19]: ('foo', 'bar') * 4
Out[19]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

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

Распаковка кортежей

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

```
In [20]: tup = (4, 5, 6)
```

```
In [21]: a, b, c = tup
```

```
In [22]: b
```

```
Out[22]: 5
```

Распаковать можно даже вложенный кортеж:

```
In [23]: tup = 4, 5, (6, 7)
```

```
In [24]: a, b, (c, d) = tup
```

```
In [25]: d
```

```
Out[25]: 7
```

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

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

Однако в Python обменять значения можно и так:

```
In [26]: a, b = 1, 2
```

```
In [27]: a
```

```
Out[27]: 1
```

```
In [28]: b
```

```
Out[28]: 2
```

```
In [29]: b, a = a, b
```

```
In [30]: a
```

```
Out[30]: 2
```

```
In [31]: b
```

```
Out[31]: 1
```

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


```
In [32]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [33]: for a, b, c in seq:
.....:     print(f'a={a}, b={b}, c={c}')
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

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

Бывают ситуации, когда требуется «отщепить» несколько элементов из начала кортежа. Для этого применяется специальный синтаксис `*rest`, используемый также в сигнатурах функций, чтобы обозначить сколь угодно длинный список позиционных аргументов:

```
In [34]: values = 1, 2, 3, 4, 5
```

```
In [35]: a, b, *rest = values
```

```
In [36]: a
Out[36]: 1
```

```
In [37]: b
Out[37]: 2
```

```
In [38]: rest
Out[38]: [3, 4, 5]
```

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

```
In [39]: a, b, *_ = values
```

Методы кортежа

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

```
In [40]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [41]: a.count(2)
Out[41]: 4
```

Список

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

```
In [42]: a_list = [2, 3, 7, None]
```

```
In [43]: tup = ("foo", "bar", "baz")
```

```
In [44]: b_list = list(tup)
```

```
In [45]: b_list
Out[45]: ['foo', 'bar', 'baz']

In [46]: b_list[1] = "peekaboo"

In [47]: b_list
Out[47]: ['foo', 'peekaboo', 'baz']
```

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

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

```
In [48]: gen = range(10)

In [49]: gen
Out[49]: range(0, 10)

In [50]: list(gen)
Out[50]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Добавление и удаление элементов

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

```
In [51]: b_list.append("dwarf")

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

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

```
In [53]: b_list.insert(1, "red")

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

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



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

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

```
In [55]: b_list.pop(2)
Out[55]: 'peekaboo'

In [56]: b_list
Out[56]: ['foo', 'red', 'baz', 'dwarf']
```

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

```
In [57]: b_list.append("foo")

In [58]: b_list
Out[58]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [59]: b_list.remove("foo")

In [60]: b_list
Out[60]: ['red', 'baz', 'dwarf', 'foo']
```

Если снижение производительности из-за использования методов `append` и `remove` не составляет проблемы, то список Python можно использовать в качестве структуры данных, подобной множеству (хотя в Python есть настоящие множества, которые будут описаны ниже).

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

```
In [61]: "dwarf" in b_list
Out[61]: True
```

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

Конкатенация и комбинирование списков

Как и в случае кортежей, операция сложения конкатенирует списки:

```
In [63]: [4, None, "foo"] + [7, 8, (2, 3)]
Out[63]: [4, None, 'foo', 7, 8, (2, 3)]
```

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

```
In [64]: x = [4, None, "foo"]

In [65]: x.extend([7, 8, (2, 3)])

In [66]: x
Out[66]: [4, None, 'foo', 7, 8, (2, 3)]
```

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

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

быстрее, чем эквивалентная конкатенация:

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

Сортировка

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

```
In [67]: a = [7, 2, 5, 1, 3]
```

```
In [68]: a.sort()
```

```
In [69]: a
```

```
Out[69]: [1, 2, 3, 5, 7]
```

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

```
In [70]: b = ["saw", "small", "He", "foxes", "six"]
```

```
In [71]: b.sort(key=len)
```

```
In [72]: b
```

```
Out[72]: ['He', 'saw', 'six', 'small', 'foxes']
```

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

Вырезание

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

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

```
In [74]: seq[1:5]
```

```
Out[74]: [2, 3, 7, 5]
```

Срезу также можно присваивать последовательность:

```
In [75]: seq[3:5] = [6, 3]
```

```
In [76]: seq
```

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

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

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

```
In [77]: seq[:5]
```

```
Out[77]: [7, 2, 3, 6, 3]
```

```
In [78]: seq[3:]
```

```
Out[78]: [6, 3, 6, 0, 1]
```

Если индекс в срезе отрицателен, то отсчет ведется от конца последовательности:

```
In [79]: seq[-4:]
Out[79]: [3, 6, 0, 1]
```

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

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

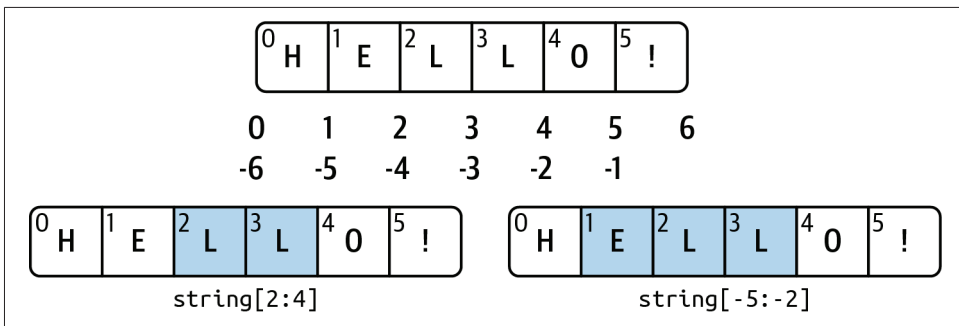


Рис. 3.1. Иллюстрация соглашений о вырезании в Python

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

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 0]
```

Если задать шаг `-1`, то список или кортеж будет инвертирован:

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

Словарь

Словарь, или `dict`, пожалуй, является самой важной из встроенных в Python структур данных. В других языках программирования словарь иногда называют *хешем*, *отображением* или *ассоциативным массивом*. Он представляет собой коллекцию пар *ключ-значение*, в которой и *ключ*, и *значение* – объекты Python. С каждым ключом ассоциировано значение, так что значение можно извлекать, вставлять, изменять или удалять, если известен ключ. Создать словарь можно, в частности, с помощью фигурных скобок `{}`, отделяя ключи от значений двоеточием:

```
In [83]: empty_dict = {}
```

```
In [84]: d1 = {"a": "some value", "b": [1, 2, 3, 4]}
```

```
In [85]: d1
Out[85]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

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

```
In [86]: d1[7] = "an integer"

In [87]: d1
Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

```
In [88]: d1["b"]
Out[88]: [1, 2, 3, 4]
```

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

```
In [89]: "b" in d1
Out[89]: True
```

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

```
In [90]: d1[5] = "some value"

In [91]: d1
Out[91]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}
```

```
In [92]: d1["dummy"] = "another value"

In [93]: d1
Out[93]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}
```

```
In [94]: del d1[5]

In [95]: d1
Out[95]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}
```

```
In [96]: ret = d1.pop("dummy")

In [97]: ret
Out[97]: 'another value'
```

```
In [98]: d1
Out[98]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

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

```
In [99]: list(d1.keys())
Out[99]: ['a', 'b', 7]
```

```
In [100]: list(d1.values())
Out[100]: ['some value', [1, 2, 3, 4], 'an integer']
```

Чтобы обойти одновременно ключи и значения, воспользуйтесь методом `items`, который отдает 2-кортежи, состоящие из ключа и значения:

```
In [101]: list(d1.items())
Out[101]: [('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]
```

Два словаря можно объединить в один методом `update`:

```
In [102]: d1.update({"b": "foo", "c": 12})
```

```
In [103]: d1
Out[103]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

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

Создание словаря из последовательностей

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

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

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

```
In [104]: tuples = zip(range(5), reversed(range(5)))
```

```
In [105]: tuples
Out[105]: <zip at 0x7fefe4553a00>
```

```
In [106]: mapping = dict(tuples)
```

```
In [107]: mapping
Out[107]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

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

Значения по умолчанию

Часто можно встретить код, реализующий такую логику:

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

```
else:
    value = default_value
```

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

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

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

```
In [108]: words = ["apple", "bat", "bar", "atom", "book"]
```

```
In [109]: by_letter = {}
```

```
In [110]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
.....:
```

```
In [111]: by_letter
```

```
Out[111]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

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

```
In [112]: by_letter = {}
```

```
In [113]: for word in words:
.....:     letter = word[0]
.....:     by_letter.setdefault(letter, []).append(word)
.....:
```

```
In [114]: by_letter
```

```
Out[114]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

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

```
In [115]: from collections import defaultdict
```

```
In [116]: by_letter = defaultdict(list)
```

```
In [117]: for word in words:
.....:     by_letter[word[0]].append(word)
```

Допустимые типы ключей словаря

Значениями словаря могут быть произвольные объекты Python, но ключами должны быть неизменяемые объекты, например скалярные типы (`int`, `float`, строка) или кортежи (причем все объекты кортежа тоже должны быть неиз-

меняемыми). Технически это свойство называется *хешируемостью*. Проверить, является ли объект хешируемым (и, стало быть, может быть ключом словаря), позволяет функция `hash`:

```
In [118]: hash("string")
Out[118]: 3634226001988967898

In [119]: hash((1, 2, (2, 3)))
Out[119]: -9209053662355515447

In [120]: hash((1, 2, [2, 3])) # ошибка, потому что списки изменяемы
-----
TypeError                                Traceback (most recent call last)
<ipython-input-120-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # ошибка, потому что списки изменяемы
TypeError: unhashable type: 'list'
```

Хеш-значения, которые возвращает функция `hash`, вообще говоря, зависят от версии Python.

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

```
In [121]: d = {}

In [122]: d[tuple([1, 2, 3])] = 5

In [123]: d
Out[123]: {(1, 2, 3): 5}
```

Множество

Множество – это неупорядоченная коллекция уникальных элементов. Создать множество можно двумя способами: с помощью функции `set` или задав *множество-литерал* в фигурных скобках:

```
In [124]: set([2, 2, 2, 1, 3, 3])
Out[124]: {1, 2, 3}

In [125]: {2, 2, 2, 1, 3, 3}
Out[125]: {1, 2, 3}
```

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

```
In [126]: a = {1, 2, 3, 4, 5}

In [127]: b = {3, 4, 5, 6, 7, 8}
```

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

```
In [128]: a.union(b)
Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}

In [129]: a | b
Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}
```

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

```
In [130]: a.intersection(b)
Out[130]: {3, 4, 5}
```

```
In [131]: a & b
Out[131]: {3, 4, 5}
```

Наиболее употребительные методы множеств перечислены в табл. 3.1.

Таблица 3.1. Операции над множествами в Python

Функция	Альтернативный синтаксис	Описание
<code>a.add(x)</code>	Нет	Добавить элемент <code>x</code> в множество <code>a</code>
<code>a.clear(x)</code>	Нет	Опустошить множество, удалив из него все элементы
<code>a.remove(x)</code>	Нет	Удалить элемент <code>x</code> из множества <code>a</code>
<code>a.pop(x)</code>	Нет	Удалить какой-то элемент <code>x</code> из множества <code>a</code> и возбудить исключение <code>KeyError</code> , если множество пусто
<code>a.union(b)</code>	<code>a b</code>	Найти все уникальные элементы, входящие либо в <code>a</code> , либо в <code>b</code>
<code>a.update(b)</code>	<code>a = b</code>	Присвоить <code>a</code> объединение элементов <code>a</code> и <code>b</code>
<code>a.intersection(b)</code>	<code>a & b</code>	Найти все элементы, входящие и в <code>a</code> , и в <code>b</code>
<code>a.intersection_update(b)</code>	<code>a &= b</code>	Присвоить <code>a</code> пересечение элементов <code>a</code> и <code>b</code>
<code>a.difference(b)</code>	<code>a - b</code>	Найти элементы, входящие в <code>a</code> , но не входящие в <code>b</code>
<code>a.difference_update(b)</code>	<code>a -= b</code>	Записать в <code>a</code> элементы, которые входят в <code>a</code> , но не входят в <code>b</code>
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>	Найти элементы, входящие либо в <code>a</code> , либо в <code>b</code> , но не в <code>a</code> и <code>b</code> одновременно
<code>a.symmetric_difference_update(b)</code>	<code>a ^= b</code>	Записать в <code>a</code> элементы, которые входят либо в <code>a</code> , либо в <code>b</code> , но не в <code>a</code> и <code>b</code> одновременно
<code>a.issubset(b)</code>	Нет	<code>True</code> , если все элементы <code>a</code> входят также и в <code>b</code>
<code>a.issuperset(b)</code>	Нет	<code>True</code> , если все элементы <code>b</code> входят также и в <code>a</code>
<code>a.isdisjoint(b)</code>	Нет	<code>True</code> , если у <code>a</code> и <code>b</code> нет ни одного общего элемента



Если подать на вход методам типа `union` или `intersection` значение, не являющееся множеством, то Python преобразует его в множество, прежде чем выполнять операцию. Для бинарных операторов оба аргумента уже должны быть множествами.

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

```
In [132]: c = a.copy()
In [133]: c |= b
In [134]: c
Out[134]: {1, 2, 3, 4, 5, 6, 7, 8}
In [135]: d = a.copy()
In [136]: d &= b
In [137]: d
Out[137]: {3, 4, 5}
```

Как и в случае словарей, элементы множества, вообще говоря, должны быть неизменяемыми и *хешируемыми* (т. е. вызов `hash` для значения не должен возбуждать исключения). Чтобы сохранить в множестве элементы, подобные списку, необходимо сначала преобразовать их в кортеж:

```
In [138]: my_data = [1, 2, 3, 4]
In [139]: my_set = {tuple(my_data)}
In [140]: my_set
Out[140]: {(1, 2, 3, 4)}
```

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

```
In [141]: a_set = {1, 2, 3, 4, 5}
In [142]: {1, 2, 3}.issubset(a_set)
Out[142]: True
In [143]: a_set.issuperset({1, 2, 3})
Out[143]: True
```

Множества называются равными, если состоят из одинаковых элементов:

```
In [144]: {1, 2, 3} == {3, 2, 1}
Out[144]: True
```

Встроенные функции последовательностей

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

enumerate

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

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

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

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

sorted

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

```
In [145]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[145]: [0, 1, 2, 2, 3, 6, 7]

In [146]: sorted("horse race")
Out[146]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

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

zip

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

```
In [147]: seq1 = ["foo", "bar", "baz"]

In [148]: seq2 = ["one", "two", "three"]

In [149]: zipped = zip(seq1, seq2)

In [150]: list(zipped)
Out[150]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

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

```
In [151]: seq3 = [False, True]

In [152]: list(zip(seq1, seq2, seq3))
Out[152]: [('foo', 'one', False), ('bar', 'two', True)]
```

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

```
In [153]: for index, (a, b) in enumerate(zip(seq1, seq2)):
.....:     print(f"{index}: {a}, {b}")
.....:
0: foo, one
```

```
1: bar, two
2: baz, three
```

reversed

Функция `reversed` перебирает элементы последовательности в обратном порядке:

```
In [154]: list(reversed(range(10)))
Out[154]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

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

Списковое, словарное и множественное включения

*Списковое включение*³ – одна из самых любимых особенностей Python. Этот механизм позволяет кратко записать создание нового списка, образованного фильтрацией элементов коллекции с одновременным преобразованием элементов, прошедших через фильтр. Основная синтаксическая форма такова:

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

Это эквивалентно следующему циклу `for`:

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

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

```
In [155]: strings = ["a", "as", "bat", "car", "dove", "python"]

In [156]: [x.upper() for x in strings if len(x) > 2]
Out[156]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

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

Словарное включение выглядит так:

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

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

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

Все виды включений – не более чем дополнительное удобство, упрощающее написание и чтение кода. Рассмотрим приведенный выше список

³ Более-менее устоявшийся перевод термина `list comprehension` – «списковое включение» – крайне неудачен и совершенно не отражает сути дела. Я предложил бы перевод «трансфильтрация», являющийся объединением слов «трансформация» и «фильтрация», но не уверен в реакции сообщества. – *Прим. перев.*

строк. Допустим, что требуется построить множество, содержащее длины входящих в коллекцию строк; это легко сделать с помощью множественного включения:

```
In [157]: unique_lengths = {len(x) for x in strings}
```

```
In [158]: unique_lengths
Out[158]: {1, 2, 3, 4, 6}
```

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

```
In [159]: set(map(len, strings))
Out[159]: {1, 2, 3, 4, 6}
```

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

```
In [160]: loc_mapping = {value: index for index, value in enumerate(strings)}

In [161]: loc_mapping
Out[161]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Вложенное списковое включение

Пусть имеется список списков, содержащий английские и испанские имена:

```
In [162]: all_data = [
.....:     ["John", "Emily", "Michael", "Mary", "Steven"],
.....:     ["Maria", "Juan", "Javier", "Natalia", "Pilar"]]
```

Допустим, что требуется получить один список, содержащий все имена, в которых встречается не менее двух букв `a`. Конечно, это можно было бы сделать в таком простом цикле `for`:

```
In [163]: names_of_interest = []

In [164]: for names in all_data:
.....:     enough_as = [name for name in names if name.count("a") >= 2]
.....:     names_of_interest.extend(enough_as)
.....:
In [165]: names_of_interest
Out[165]: ['Maria', 'Natalia']
```

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

```
In [166]: result = [name for names in all_data for name in names
.....:                if name.count("a") >= 2]

In [167]: result
Out[167]: ['Maria', 'Natalia']
```

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

```
In [168]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [169]: flattened = [x for tup in some_tuples for x in tup]
```

```
In [170]: flattened
Out[170]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
flattened = []

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

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

```
In [172]: [[x for x in tup] for tup in some_tuples]
Out[172]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

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

3.2. Функции

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

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

```
In [173]: def my_function(x, y):
.....:     return x + y
```

По достижении строки `return` значение или выражение, следующее после `return`, передается в контекст, из которого была вызвана функция, например:

```
In [174]: my_function(1, 2)
Out[174]: 3

In [175]: result = my_function(1, 2)

In [176]: result
Out[176]: 3
```

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

```
In [177]: def function_without_return(x):
.....:     print(x)

In [178]: result = function_without_return("hello!")
```

```
hello!
```

```
In [179]: print(result)
None
```

У функции могут быть *позиционные* и *именованные* аргументы. Именованные аргументы обычно используются для задания значений по умолчанию и необязательных аргументов. Ниже определена функция с факультативным аргументом `z`, который по умолчанию принимает значение `1.5`:

```
def my_function2(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

Именованные аргументы факультативны, но все позиционные аргументы должны быть заданы при вызове функции.

Передавать значения аргумента `z` можно с указанием и без указания имени, но рекомендуется все-таки имя указывать:

```
In [181]: my_function2(5, 6, z=0.7)
Out[181]: 0.06363636363636363

In [182]: my_function2(3.14, 7, 3.5)
Out[182]: 35.49
```

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

Пространства имен, области видимости и локальные функции

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

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

При вызове `func()` создается пустой список `a`, в него добавляется 5 элементов, а затем, когда функция завершается, список `a` уничтожается. Но допустим, что мы объявили `a` следующим образом:


```
In [184]: a = []
```

```
In [185]: def func():
.....:     for i in range(5):
.....:         a.append(i)
```

Каждый вызов `func` изменяет список `a`:

```
In [186]: func()
```

```
In [187]: a
Out[187]: [0, 1, 2, 3, 4]
```

```
In [188]: func()
```

```
In [189]: a
Out[189]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

Присваивать значение глобальной переменной вне области видимости функции допустимо, но такие переменные должны быть объявлены с помощью ключевого слова `global` или `nonlocal`:

```
In [190]: a = None
```

```
In [191]: def bind_a_variable():
.....:     global a
.....:     a = []
.....:     bind_a_variable()
.....:
```

```
In [192]: print(a)
[]
```

Ключевое слово `nonlocal` позволяет функции модифицировать переменные, определенные в объемлющей, но не глобальной области видимости. Поскольку употребляется оно в экзотических обстоятельствах (в этой книге не встречается ни разу), отсылаю вас к документации по Python.



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

Возврат нескольких значений

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

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

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

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

```
return_value = f()
```

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

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

Полезен ли такой способ, зависит от решаемой задачи.

Функции являются объектами

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

```
In [193]: states = [" Alabama ", "Georgia!", "Georgia", "georgia", "Fl0rIda",
.....:             "south carolina##", "West virginia?"]
```

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

```
import re

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

Вот как выглядит результат:

```
In [195]: clean_strings(states)
Out[195]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Другой подход, который иногда бывает полезен, – составить список операций, которые необходимо применить к набору строк:

```
def remove_punctuation(value):
    return re.sub("[!#?]", "", value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for func in ops:
            value = func(value)
        result.append(value)
    return result
```

Далее поступаем следующим образом:

```
In [197]: clean_strings(states, clean_ops)
Out[197]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Подобный *функциональный* подход позволяет задать способ модификации строк на очень высоком уровне. Степень общности и повторной используемости функции `clean_strings` определенно возросла!

Функции можно передавать в качестве аргументов другим функциям, например встроенной функции `map`, которая применяет переданную функцию к последовательности:

```
In [198]: for x in map(remove_punctuation, states):
.....:     print(x)
Alabama
Georgia
Georgia
georgia
FLOrIda
south carolina
West virginia
```

Анонимные (лямбда-) функции

Python поддерживает так называемые *анонимные*, или *лямбда*-, функции. По существу, это простые однострочные функции, возвращающие значение. Определяются они с помощью ключевого слова `lambda`, которое означает всего лишь «мы определяем анонимную функцию» и ничего более.

```
In [199]: def short_function(x):
.....:     return x * 2
```

```
In [200]: equiv_anon = lambda x: x * 2
```

В этой книге я обычно употребляю термин «лямбда-функция». Они особенно удобны в ходе анализа данных, потому что, как вы увидите, во многих случаях функции преобразования данных принимают другие функции в качестве аргументов. Часто быстрее (и чище) передать лямбда-функцию, чем писать полноценное объявление функции или даже присваивать лямбда-функцию локальной переменной. Рассмотрим пример:

```
In [201]: def apply_to_list(some_list, f):
.....:     return [f(x) for x in some_list]
```

```
In [202]: ints = [4, 0, 1, 5, 6]
```

```
In [203]: apply_to_list(ints, lambda x: x * 2)
Out[203]: [8, 0, 2, 10, 12]
```

Можно было бы, конечно, написать `[x * 2 for x in ints]`, но в данном случае нам удалось передать функции `apply_to_list` пользовательский оператор.

Еще пример: пусть требуется отсортировать коллекцию строк по количеству различных букв в строке.

```
In [204]: strings = ["foo", "card", "bar", "aaaa", "abab"]
```

Для этого можно передать лямбда-функцию методу списка `sort`:

```
In [205]: strings.sort(key=lambda x: len(set(x)))
```

```
In [206]: strings
Out[206]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

Генераторы

Многие объекты в Python поддерживают итерирование, например элементов списка или строк в файле. Эта возможность реализована с помощью протокола *итератора*, общего механизма, наделяющего объекты свойством итерируемости. Например, при обходе (итерировании) словаря мы получаем хранящиеся в нем ключи:

```
In [207]: some_dict = {"a": 1, "b": 2, "c": 3}
```

```
In [208]: for key in some_dict:
.....:     print(key)
```

```
a
b
c
```

Встречая конструкцию `for key in some_dict`, интерпретатор Python сначала пытается создать итератор из `some_dict`:

```
In [209]: dict_iterator = iter(some_dict)

In [210]: dict_iterator
Out[210]: <dict_keyiterator at 0x7fefe45465c0>
```

Итератор – это любой объект, который отдает интерпретатору Python объекты при использовании в контексте, аналогичном циклу `for`. Методы, ожидающие получить список или похожий на список объект, как правило, удовлетворяются любым итерируемым объектом. Это относится, в частности, к встроенным методам, например `min`, `max` и `sum`, и к конструкторам типов, например `list` и `tuple`:

```
In [211]: list(dict_iterator)
Out[211]: ['a', 'b', 'c']
```

Генератор – это удобный механизм конструирования итерируемого объекта, похожий на обычную функцию. Если обычная функция выполняется и возвращает единственное значение, то генератор может возвращать последовательность значений, приостанавливаясь после возврата каждого в ожидании запроса следующего. Чтобы создать генератор, нужно вместо `return` использовать ключевое слово `yield`:

```
def squares(n=10):
    print(f"Генерируются квадраты чисел от 1 до {n ** 2}")
    for i in range(1, n + 1):
        yield i ** 2
```

В момент вызова генератора никакой код не выполняется:

```
In [213]: gen = squares()

In [214]: gen
Out[214]: <generator object squares at 0x7fefe437d620>
```

И лишь после запроса элементов генератор начинает выполнять свой код:

```
In [215]: for x in gen:
.....:     print(x, end=" ")
Генерируются квадраты чисел от 1 до 100
1 4 9 16 25 36 49 64 81 100
```



Поскольку генераторы отдают значения по одному, а не весь список сразу, программа потребляет меньше памяти.

Генераторные выражения

Еще один способ создать генератор – воспользоваться *генераторным выражением*. Такой генератор аналогичен списковому, словарному и множественному включениям; чтобы его создать, заключите выражение, которое выглядит как списковое включение, в круглые скобки вместо квадратных:

```
In [216]: gen = (x ** 2 for x in range(100))

In [217]: gen
Out[217]: <generator object <genexpr> at 0x7fefe437d000>
```

Это в точности эквивалентно следующему более многословному определению генератора:

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

В некоторых случаях генераторные выражения можно передавать функциям вместо списковых включений:

```
In [218]: sum(x ** 2 for x in range(100))
Out[218]: 328350

In [219]: dict((i, i ** 2) for i in range(5))
Out[219]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

В зависимости от того, сколько элементов порождает списковое включение, версия с генератором может оказаться заметно быстрее.

Модуль `itertools`

Стандартный библиотечный модуль `itertools` содержит набор генераторов для многих общеупотребительных алгоритмов. Так, генератор `groupby` принимает произвольную последовательность и функцию и группирует соседние элементы последовательности по значению, возвращенному функцией, например:

```
In [220]: import itertools

In [221]: def first_letter(x):
.....:     return x[0]

In [222]: names = ["Alan", "Adam", "Wes", "Will", "Albert", "Steven"]

In [223]: for letter, names in itertools.groupby(names, first_letter):
.....:     print(letter, list(names)) # names - это генератор
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

В табл. 3.2 описаны некоторые функции из модуля `itertools`, которыми я часто пользуюсь. Дополнительную информацию об этом полезном стандартном модуле можно почерпнуть из официальной документации по адресу <https://docs.python.org/3/library/itertools.html>.

Таблица 3.2. Некоторые полезные функции из модуля `itertools`

Функция	Описание
<code>chain(*iterables)</code>	Генерирует последовательность сцепленных итераторов. После исчерпания элементов в первом итераторе возвращаются элементы из второго и т. д.
<code>combinations(iterable, k)</code>	Генерирует последовательность всех возможных <code>k</code> -кортежей, составленных из элементов <code>iterable</code> , без учета порядка
<code>permutations(iterable, k)</code>	Генерирует последовательность всех возможных <code>k</code> -кортежей, составленных из элементов <code>iterable</code> , с учетом порядка
<code>groupby(iterable[, keyfunc])</code>	Генерирует пары (ключ, субитератор) для каждого уникального ключа
<code>product(*iterables, repeat=1)</code>	Генерирует декартово произведение входных итерируемых величин в виде кортежей, как если бы использовался вложенный цикл <code>for</code>

Обработка исключений

Обработка ошибок, или *исключений*, в Python – важная часть создания надежных программ. В приложениях для анализа данных многие функции работают только для входных данных определенного вида. Например, функция `float` может привести строку к типу числа с плавающей точкой, но если формат строки заведомо некорректен, то завершается с ошибкой `ValueError`:

```
In [224]: float("1.2345")
Out[224]: 1.2345

In [225]: float("something")
-----
ValueError                                Traceback (most recent call last)
<ipython-input-225-5ccfe07933f4> in <module>
----> 1 float("something")
ValueError: could not convert string to float: 'something'
```

Пусть требуется написать версию `float`, которая не завершается с ошибкой, а возвращает поданный на вход аргумент. Это можно сделать, обернув вызов `float` блоком `try/except`:

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

Код в части `except` будет выполняться, только если `float(x)` возбуждает исключение:

```
In [227]: attempt_float("1.2345")
Out[227]: 1.2345

In [228]: attempt_float("something")
Out[228]: 'something'
```

Кстати, `float` может возбуждать и другие исключения, не только `ValueError`:

```
In [229]: float((1, 2))
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-229-82f777b0e564> in <module>
----> 1 float((1, 2))
TypeError: float() argument must be a string or a real number, not 'tuple'
```

Возможно, вы хотите перехватить только исключение `ValueError`, поскольку `TypeError` (аргумент был не строкой и не числом) может свидетельствовать об ожидаемой ошибке в программе. Для этого нужно написать после `except` тип исключения:

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

В таком случае получим:

```
In [231]: attempt_float((1, 2))
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-231-8b0026e9e6b7> in <module>
----> 1 attempt_float((1, 2))
<ipython-input-230-6209ddec2b5> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a real number, not 'tuple'
```

Можно перехватывать исключения нескольких типов, для этого достаточно написать кортеж типов (скобки обязательны):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

Иногда исключение не нужно перехватывать, но какой-то код должен быть выполнен вне зависимости от того, возникло исключение в блоке `try` или нет. Для этого служит предложение `finally`:

```
f = open(path, mode="w")

try:
    write_to_file(f)
finally:
    f.close()
```

Здесь объект файла `f` закрывается *в любом случае*. Можно также написать код, который выполняется, только если в блоке `try` не было исключения, для этого используется ключевое слово `else`:


```
f = open(path, mode="w")

try:
    write_to_file(f)
except:
    print("Ошибка")
else:
    print("Все хорошо")
finally:
    f.close()
```

Исключения в IPython

Если исключение возникает в процессе выполнения скрипта командой `%run` или при выполнении любого предложения, то IPython по умолчанию распечатывает весь стек (выполняет трассировку стека) и несколько строк вокруг каждого предложения в стеке, чтобы можно было понять контекст:

```
In [10]: %run examples/ipython_bug.py
-----
AssertionError                                Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
      13 throws_an_exception()
      14
----> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
      11 def calling_things():
      12     works_fine()
----> 13     throws_an_exception()
      14
      15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
       7 a = 5
       8 b = 6
---->  9 assert(a + b == 10)
      10
      11 def calling_things():

AssertionError:
```

Наличие дополнительного контекста уже является весомым преимуществом по сравнению со стандартным интерпретатором Python (который никакого контекста не выводит). Объемом контекста можно управлять с помощью магической команды `%xmode`, он может варьироваться от `Plain` (так же, как в стандартном интерпретаторе Python) до `Verbose` (печатаются значения аргументов функций и многое другое). Ниже в приложении В мы увидим, что можно *пошагово выполнять стек* (с помощью команды `%debug` или `%pdb`) после возникновения ошибки, т. е. производить интерактивную постоперационную отладку.

3.3. Файлы и ОПЕРАЦИОННАЯ СИСТЕМА

В этой книге для чтения файла с диска и загрузки данных из него в структуры Python, как правило, используются такие высокоуровневые средства, как

функция `pandas.read_csv`. Однако важно понимать основы работы с файлами в Python. По счастью, здесь все очень просто, и именно поэтому Python так часто выбирают, когда нужно работать с текстом или файлами.

Чтобы открыть файл для чтения или для записи, пользуйтесь встроенной функцией `open`, которая принимает относительный или абсолютный путь:

```
In [233]: path = "examples/segismundo.txt"
```

```
In [234]: f = open(path, encoding="utf-8")
```

По умолчанию файл открывается только для чтения – в режиме `'r'`. Далее описатель файла `f` можно рассматривать как список и перебирать строки:

```
for line in f:
    print(line)
```

У строк, прочитанных из файла, сохраняется признак конца строки (EOL), поэтому часто можно встретить код, который удаляет концы строк:

```
In [235]: lines = [x.rstrip() for x in open(path, encoding="utf-8")]
```

```
In [236]: lines
```

```
Out[236]:
```

```
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

Если для создания объекта файла использовалась функция `open`, то следует явно закрывать файл по завершении работы с ним. Закрывание файла возвращает ресурсы операционной системе:

```
In [237]: f.close()
```

Упростить эту процедуру позволяет предложение `with`:

```
In [238]: with open(path) as f:
.....:     lines = [x.rstrip() for x in f]
```

В таком случае файл `f` автоматически закрывается по выходе из блока `with`. Если не позаботиться о закрытии файлов, то в небольших скриптах может и не произойти ничего страшного, но в программах, работающих с большим количеством файлов, возможны проблемы.

Если бы мы написали `f = open(path, 'w')`, то был бы создан *новый* файл `examples/segismundo.txt` (будьте осторожны!), а старый был бы перезаписан. Существует также режим `'x'`, в котором создается допускающий запись файл, но лишь

в том случае, если его еще не существует, а в противном случае возбуждается исключение. Все допустимые режимы ввода-вывода перечислены в табл. 3.3.

Таблица 3.3. Режимы открытия файла в Python

Режим	Описание
<code>r</code>	Режим чтения
<code>w</code>	Режим записи. Создается новый файл (а старый с тем же именем удаляется)
<code>x</code>	Режим записи. Создается новый файл, но возникает ошибка, если файл с таким именем уже существует
<code>a</code>	Дописывание в конец существующего файла (если файл не существует, он создается)
<code>r+</code>	Чтение и запись
<code>b</code>	Уточнение режима для двоичных файлов: <code>'rb'</code> или <code>'wb'</code>
<code>t</code>	Текстовый режим (байты автоматически декодируются в Unicode). Этот режим подразумевается по умолчанию, если не указано противное

При работе с файлами, допускающими чтение, чаще всего употребляются методы `read`, `seek` и `tell`. Метод `read` возвращает определенное число символов из файла. Что такое «символ», определяется кодировкой файла. Если же файл открыт в двоичном режиме, то под символами понимаются просто байты:

```
In [239]: f1 = open(path)

In [240]: f1.read(10)
Out[240]: 'Sueca el r'

In [241]: f2 = open(path, mode="rb") # Двоичный режим

In [242]: f2.read(10)
Out[242]: b'Sue\xc3\xb1a el '
```

Метод `read` продвигает указатель файла вперед на количество прочитанных байтов. Метод `tell` сообщает текущую позицию:

```
In [243]: f1.tell()
Out[243]: 11

In [244]: f2.tell()
Out[244]: 10
```

Хотя мы прочитали из файла 10 символов, позиция равна 11, потому что именно столько байтов пришлось прочитать, чтобы декодировать 19 символов в подразумеваемой по умолчанию кодировке файла. Чтобы узнать кодировку по умолчанию, воспользуемся модулем `sys`:

```
In [245]: import sys

In [246]: sys.getdefaultencoding()
Out[246]: 'utf-8'
```

Чтобы поведение на разных платформах было одинаковым, рекомендуется явно передавать кодировку (например, широко используемую кодировку `encoding="utf-8"`) при открытии файлов.

Метод `seek` изменяет позицию в файле на указанную:

```
In [247]: f1.seek(3)
Out[247]: 3
```

```
In [248]: f1.read(1)
Out[248]: 'ñ'
```

```
In [249]: f1.tell()
Out[249]: 5
```

Наконец, не забудем закрыть файлы:

```
In [250]: f1.close()
In [251]: f2.close()
```

Для записи текста в файл служат методы `write` или `writelines`. Например, можно было бы создать вариант файла `examples/segismundo.txt` без пустых строк:

```
In [252]: path
Out[252]: 'examples/segismundo.txt'
```

```
In [253]: with open("tmp.txt", mode="w") as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)
```

```
In [254]: with open("tmp.txt") as f:
.....:     lines = f.readlines()
```

```
In [255]: lines
Out[255]:
['Sueña el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueña el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueña el que a medrar empieza,\n',
 'sueña el que afana y pretende,\n',
 'sueña el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
 'todos sueñan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

В табл. 3.4 приведены многие из наиболее употребительных методов работы с файлами.

Таблица 3.4. Наиболее употребительные методы и атрибуты для работы с файлами в Python

Метод	Описание
<code>read([size])</code>	Возвращает прочитанные из файла данные в виде строки. Необязательный аргумент <code>size</code> говорит, сколько байтов читать
<code>readable()</code>	Возвращает <code>True</code> , если файл поддерживает операции <code>read</code>
<code>readlines([size])</code>	Возвращает список прочитанных из файла строк. Необязательный аргумент <code>size</code> говорит, сколько строк читать

Метод	Описание
<code>write(string)</code>	Записывает переданную строку в файл
<code>writable()</code>	Возвращает <code>True</code> , если файл поддерживает операции <code>write</code>
<code>writelines(strings)</code>	Записывает переданную последовательность строк в файл
<code>close()</code>	Закрывает дескриптор файла
<code>flush()</code>	Сбрасывает внутренний буфер ввода-вывода на диск
<code>seek(pos)</code>	Перемещает указатель чтения-записи на байт файла с указанным номером
<code>seekable()</code>	Возвращает <code>True</code> , если файл поддерживает поиск, а следовательно, произвольный доступ (некоторые файлоподобные объекты этого не делают)
<code>tell()</code>	Возвращает текущую позицию в файле в виде целого числа
<code>closed</code>	<code>True</code> , если файл закрыт
<code>encoding</code>	Кодировка, используемая при интерпретации байтов файла как Unicode (обычно UTF-8)

Байты и Unicode в применении к файлам

По умолчанию Python открывает файлы (как для чтения, так и для записи) в *текстовом режиме*, предполагая, что вы намереваетесь работать со строками (которые хранятся в Unicode). Чтобы открыть файл в *двоичном режиме*, следует добавить к основному режиму букву `b`. Рассмотрим файл из предыдущего раздела (содержащий не-ASCII символы в кодировке UTF-8):

```
In [258]: with open(path) as f:
.....:     chars = f.read(10)
In [259]: chars
Out[259]: 'Sueña e l r'
```

```
In [260]: len(chars)
Out[260]: 10
```

UTF-8 – это кодировка Unicode переменной длины, поэтому когда я запрашиваю чтение нескольких символов из файла, Python читает столько байтов, чтобы после декодирования получилось указанное количество символов (это может быть всего 10, а может быть и целых 40 байт). Если вместо этого открыть файл в режиме `'rb'`, то `read` прочитает ровно столько байтов, сколько запрошено:

```
In [261]: with open(path, mode="rb") as f:
.....:     data = f.read(10)
```

```
In [262]: data
Out[262]: b'Sue\xc3\xb1a e l '
```

Зная кодировку текста, вы можете декодировать байты в объект `str` самостоятельно, но только в том случае, когда последовательность байтов корректна и полна:

```
In [263]: data.decode("utf-8")
Out[263]: 'Sueña el '
```

```
In [264]: data[:4].decode("utf-8")
```

```
-----
UnicodeDecodeError                                Traceback (most recent call last)
<ipython-input-264-846a5c2fed34> in <module>
----> 1 data[:4].decode("utf-8")
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3:
unexpected end of data
```

Текстовый режим в сочетании с параметром `encoding` функции `open` – удобный способ преобразовать данные из одной кодировки Unicode в другую:

```
In [265]: sink_path = "sink.txt"
```

```
In [266]: with open(path) as source:
.....:     with open(sink_path, "x", encoding="iso-8859-1") as sink:
.....:         sink.write(source.read())
```

```
In [267]: with open(sink_path, encoding="iso-8859-1") as f:
.....:     print(f.read(10))
Sueña el r
```

Будьте осторожны, вызывая метод `seek` для файла, открытого не в двоичном режиме. Если указанная позиция окажется в середине последовательности байтов, образующих один символ Unicode, то последующие операции чтения завершатся ошибкой:

```
In [269]: f = open(path, encoding='utf-8')
```

```
In [270]: f.read(5)
Out[270]: 'Sueca'
```

```
In [271]: f.seek(4)
Out[271]: 4
```

```
In [272]: f.read(1)
```

```
-----
UnicodeDecodeError                                Traceback (most recent call last)
<ipython-input-272-5a354f952aa4> in <module>
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.10/codecs.py in decode(self, input, final)
    320         # decode input (taking the buffer into account)
    321         data = self.buffer + input
--> 322         (result, consumed) = self._buffer_decode(data, self.errors, final)
    )
    323         # keep undecoded input until the next call
    324         self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid
start byte
```

```
In [273]: f.close()
```

Если вы регулярно анализируете текстовые данные в кодировке, отличной от ASCII, то уверенное владение средствами работы с Unicode, имеющимися в Python, придется весьма кстати. Дополнительные сведения смотрите в онлайн-документации по адресу <https://docs.python.org/3/>.

3.4. ЗАКЛЮЧЕНИЕ

Вооружившись базовыми знаниями о языке и Python и его среде, мы можем перейти к изучению NumPy и вычислениям с массивами.

Основы NumPy: массивы и векторные вычисления

Numerical Python, или сокращенно NumPy, – один из важнейших пакетов для численных расчетов в Python. Во многих пакетах для научных расчетов используются объекты массивов NumPy, которые являются *универсальным языком обмена данными*. Значительная часть знаний о NumPy, изложенных в этой главе, переносится и на pandas.

Вот лишь часть того, что предлагает NumPy:

- `ndarray`, эффективный многомерный массив, предоставляющий быстрые арифметические операции с массивами и гибкий механизм *укладывания*;
- математические функции для выполнения быстрых операций над целыми массивами без явного выписывания циклов;
- средства для чтения массива данных с диска и записи его на диск, а также для работы с проецируемыми на память файлами;
- алгоритмы линейной алгебры, генерация случайных чисел и преобразование Фурье;
- средства для интеграции с кодом, написанным на C, C++ или Fortran.

Благодаря наличию полного и хорошо документированного 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++ и Fortran с приданием ему динамического и простого в использовании интерфейса.

Хотя сам по себе пакет NumPy почти не содержит средств для моделирования и научных расчетов, понимание массивов NumPy и ориентированных на эти массивы вычислений поможет гораздо эффективнее использовать инструменты типа pandas. Поскольку NumPy – обширная тема, я вынес описание многих продвинутых средств NumPy, в частности укладывания, в приложение А. Многие из этих средств не понадобятся для чтения данной книги, но помогут глубже разобраться с программированием научных приложений на Python.

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

- быстрые векторные операции для переформатирования и очистки данных, выборки подмножеств и фильтрации, преобразований и других видов вычислений;

- стандартные алгоритмы работы с массивами, например фильтрация, удаление дубликатов и теоретико-множественные операции;
- эффективная описательная статистика, агрегирование и обобщение данных;
- выравнивание данных и реляционные операции объединения и соединения разнородных наборов данных;
- описание условной логики в виде выражений-массивов вместо циклов с ветвлением `if-elif-else`;
- групповые операции с данными (агрегирование, преобразование, применение функции).

Хотя в NumPy имеются вычислительные основы для этих операций, по большей части для анализа данных (особенно структурированных или табличных) лучше использовать библиотеку `pandas`, потому что она предлагает развитый высокоуровневый интерфейс для решения большинства типичных задач обработки данных – простой и лаконичный. Кроме того, в `pandas` есть кое-какая предметно-ориентированная функциональность, например операции с временными рядами, отсутствующая в NumPy.



Вычисления с массивами в Python уходят корнями в 1995 год, когда Джим Хьюганин (Jim Hugunin) создал библиотеку `Numeric`. В течение следующих десяти лет программирование с массивами распространилось во многих научных сообществах, но в начале 2000-х годов библиотечная экосистема оказалась фрагментированной. В 2005 году Трэвис Олифант (Travis Oliphant) сумел собрать проект NumPy из существовавших тогда проектов `Numeric` и `Numarray`, чтобы объединить сообщество вокруг единой вычислительной инфраструктуры.

Одна из причин, по которым NumPy играет такую важную роль в численных расчетах, – тот факт, что она проектировалась с прицелом на эффективную работу с большими массивами данных. Отметим, в частности, следующие аспекты.

- NumPy хранит данные в непрерывном блоке памяти независимо от других встроенных объектов Python. Алгоритмы NumPy, написанные на языке C, могут работать с этим блоком, не обременяя себя проверкой типов и другими накладными расходами. Массивы NumPy потребляют гораздо меньше памяти, чем встроенные в Python последовательности.
- В NumPy сложные операции применяются к массивам целиком, так что отпадает необходимость в циклах `for`, которые для больших последовательностей могут работать медленно. NumPy быстрее кода на чистом Python, потому что в написанных на C алгоритмах нет накладных расходов, свойственных обычному интерпретируемому Python-коду.

Чтобы вы могли составить представление о выигрыше в производительности, рассмотрим массив NumPy, содержащий миллион чисел, и эквивалентный список Python:

```
In [7]: import numpy as np
```

```
In [8]: my_arr = np.arange(1_000_000)
```

```
In [9]: my_list = list(range(1_000_000))
```

Умножим каждую последовательность на 2:

```
In [10]: %timeit my_arr2 = my_arr * 2
715 us +- 13.2 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

```
In [11]: %timeit my_list2 = [x * 2 for x in my_list]
48.8 ms +- 298 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Алгоритмы, основанные на NumPy, в общем случае оказываются в 10–100 раз (а то и больше) быстрее аналогов, написанных на чистом Python, и потребляют гораздо меньше памяти.

4.1. NUMPY NDARRAY: ОБЪЕКТ МНОГОМЕРНОГО МАССИВА

Одна из ключевых особенностей NumPy – объект `ndarray` для представления N -мерного массива; это быстрый и гибкий контейнер для хранения больших наборов данных в Python. Массивы позволяют выполнять математические операции над целыми блоками данных, применяя такой же синтаксис, как для соответствующих операций над скалярами.

Чтобы показать, как NumPy позволяет производить пакетные вычисления, применяя такой же синтаксис, как для встроенных в Python скалярных объектов, я начну с импорта NumPy и генерации небольшого массива случайных данных:

```
In [12]: import numpy as np
```

```
In [13]: data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])
```

```
In [14]: data
```

```
Out[14]:
array([[ 1.5, -0.1,  3. ],
       [ 0. , -3. ,  6.5]])
```

Затем я произведу математические операции над `data`:

```
In [15]: data * 10
Out[15]:
array([[ 15., -1., 30.],
       [ 0., -30., 65.]])
```

```
In [16]: data + data
Out[16]:
array([[ 3. , -0.2,  6. ],
       [ 0. , -6. , 13. ]])
```

В первом примере все элементы умножены на 10. Во втором примере соответственные элементы в каждой «ячейке» складываются.



В этой главе и во всей книге я буду придерживаться стандартного соглашения NumPy и писать `import numpy as np`. Можно было бы включать в код предложение `from numpy import *`, чтобы не писать каждый раз `np.`, но я рекомендую не брать это в привычку. Пространство имен `numpy` очень велико и содержит ряд функций, имена которых совпадают с именами встроенных в Python функций (например, `min` и `max`). Соблюдать стандартные соглашения типа этого почти всегда имеет смысл.

`ndarray` – это обобщенный многомерный контейнер для однородных данных, т. е. в нем могут храниться только элементы одного типа. У любого массива есть атрибут `shape` – кортеж, описывающий размер по каждому измерению, и атрибут `dtype` – объект, описывающий тип данных в массиве:

```
In [17]: data.shape
Out[17]: (2, 3)
```

```
In [18]: data.dtype
Out[18]: dtype('float64')
```

В этой главе мы познакомимся с основами работы с массивами NumPy в объеме, достаточном для чтения книги. Для многих аналитических приложений глубокое понимание NumPy необязательно, но овладение стилем мышления и методами программирования, ориентированными на массивы, – ключевой этап на пути становления эксперта по применению Python в научных приложениях.



Слова «массив», «массив NumPy» и «`ndarray`» в этой книге почти всегда означают одно и то же: объект `ndarray`.

Создание `ndarray`

Проще всего создать массив с помощью функции `array`. Она принимает любой объект, похожий на последовательность (в том числе другой массив), и порождает новый массив NumPy, содержащий переданные данные. Например, такое преобразование можно проделать со списком:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

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

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
```

```
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Поскольку `data2` – список списков, массив NumPy `arr2` имеет два измерения, а его форма выведена из данных. В этом легко убедиться, прочитав атрибуты `ndim` и `shape`:

```
In [25]: arr2.ndim
Out[25]: 2
```

```
In [26]: arr2.shape
Out[26]: (2, 4)
```

Если явно не задано иное (подробнее об этом ниже), то функция `np.array` пытается самостоятельно определить подходящий тип данных для создаваемого массива. Этот тип данных хранится в специальном объекте `dtype`; например, в примерах выше имеем:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')
```

```
In [28]: arr2.dtype
Out[28]: dtype('int64')
```

Помимо `numpy.array`, существует еще ряд функций для создания массивов. Например, `numpy.zeros` и `numpy.ones` создают массивы заданной длины или формы, состоящие из нулей и единиц соответственно, а `numpy.empty` создает массив, не инициализируя его элементы. Для создания многомерных массивов нужно передать кортеж, описывающий форму:

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

```
In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]])
```



Предполагать, что `numpy.empty` возвращает массив из одних нулей, небезопасно. Часто возвращается массив, содержащий неинициализированный мусор, – как в примере выше. Эту функцию следует использовать, только если вы собираетесь заполнить новый массив данными.

Функция `numpy.arange` – вариант встроенной в Python функции `range`, только возвращаемым значением является массив:

```
In [32]: np.arange(15)
Out[32]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

В табл. 4.1 приведен краткий список стандартных функций создания массива. Поскольку NumPy ориентирована 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
<code>array</code>	Преобразует входные данные (список, кортеж, массив или любую другую последовательность) в <code>ndarray</code> . Тип <code>dtype</code> задается явно или выводится неявно. Входные данные по умолчанию копируются
<code>asarray</code>	Преобразует входные данные в <code>ndarray</code> , но не копирует, если на вход уже подан <code>ndarray</code>
<code>arange</code>	Аналогична встроенной функции <code>range</code> , но возвращает массив, а не список
<code>ones, ones_like</code>	Порождает массив, состоящий из одних единиц, с заданными атрибутами <code>shape</code> и <code>dtype</code> . Функция <code>ones_like</code> принимает другой массив и порождает массив из одних единиц с такими же значениями <code>shape</code> и <code>dtype</code>
<code>zeros, zeros_like</code>	Аналогичны <code>ones</code> и <code>ones_like</code> , только порождаемый массив состоит из одних нулей
<code>empty, empty_like</code>	Создают новые массивы, выделяя под них память, но, в отличие от <code>ones</code> и <code>zeros</code> , не инициализируют ее
<code>full, full_like</code>	Создают массивы с заданными атрибутами <code>shape</code> и <code>dtype</code> , в которых все элементы равны заданному символу-заполнителю. <code>full_like</code> принимает массив и порождает заполненный массив с такими же значениями атрибутов <code>shape</code> и <code>dtype</code>
<code>eye, identity</code>	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 (метаданные), необходимую `ndarray` для интерпретации содержимого блока памяти:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Типам данных NumPy в значительной мере обязана своей эффективностью и гибкостью. В большинстве случаев они точно соответствуют внутреннему машинному представлению, что позволяет без труда читать и записывать двоичные потоки данных на диск, а также обмениваться данными с кодом, написанным на языке низкого уровня типа C или Fortran. Числовые dtype именуются единообразно: имя типа, например `float` или `int`, затем число, указывающее разрядность одного элемента. Стандартное значение с плавающей точкой двойной точности (хранящееся во внутреннем представлении объекта Python типа `float`) занимает 8 байт 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>	8-разрядное (1 байт) целое со знаком и без знака
<code>int16, uint16</code>	<code>i2, u2</code>	16-разрядное (2 байта) целое со знаком и без знака
<code>int32, uint32</code>	<code>i4, u4</code>	32-разрядное (4 байта) целое со знаком и без знака
<code>int64, uint64</code>	<code>i8, u8</code>	64-разрядное (8 байт) целое со знаком и без знака
<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</code> или <code>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>0</code>	Тип объекта Python
<code>string_</code>	<code>S</code>	Тип ASCII-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>)



Существуют целые типы *со знаком* и *без знака*, и многие читатели, скорее всего, незнакомы с этой терминологией. Целое *со знаком* может представлять как положительные, так и отрицательные целые числа, а целое *без знака* – только неотрицательные. Например, типом `int8` (8-разрядное целое со знаком) можно представить целые числа от –128 до 127 (включительно), а типом `uint8` (8-разрядное целое без знака) – числа от 0 до 255.

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

```
In [37]: arr = np.array([1, 2, 3, 4, 5])
```

```
In [38]: arr.dtype
```

```
Out[38]: dtype('int64')
```

```
In [39]: float_arr = arr.astype(np.float64)
```

```
In [40]: float_arr
```

```
Out[40]: array([1., 2., 3., 4., 5.])
```

```
In [41]: float_arr.dtype
```

```
Out[41]: 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 [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [43]: arr
```

```
Out[43]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [44]: arr.astype(np.int32)
```

```
Out[44]: 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 [45]: numeric_strings = np.array(["1.25", "-9.6", "42"], dtype=np.string_)
```

```
In [46]: numeric_strings.astype(float)
```

```
Out[46]: array([ 1.25, -9.6 , 42.  ])
```



Будьте осторожнее при работе с типом `numpy.string_`, поскольку в NumPy размер строковых данных фиксирован и входные данные могут быть обрезаны без предупреждения. Поведение pandas для нечисловых данных лучше согласуется с интуицией.

Если по какой-то причине выполнить приведение не удастся (например, если строку нельзя преобразовать в тип `float64`), то будет возбуждено исключение `ValueError`. В примере выше я поленился и написал `float` вместо `np.float64`, но NumPy оказалась достаточно «умной» – она умеет подменять типы Python эквивалентными собственными.

Можно также использовать атрибут `dtype` другого массива:

```
In [47]: int_array = np.arange(10)

In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [49]: int_array.astype(calibers.dtype)
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]
```

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

```
In [50]: zeros_uint32 = np.zeros(8, dtype="u4")

In [51]: zeros_uint32
Out[51]: array([0, 0, 0, 0, 0, 0, 0, 0], dtype=uint32)
```



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

Арифметические операции с массивами NumPy

Массивы важны, потому что позволяют выразить операции над совокупностями данных без выписывания циклов `for`. Обычно это называется *векторизацией*. Любая арифметическая операция над массивами одинакового размера применяется к соответственным элементам:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [53]: arr
Out[53]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [54]: arr * arr
Out[54]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Арифметические операции, в которых участвует скаляр, применяются к каждому элементу массива:

```
In [56]: 1 / arr
Out[56]:
array([[1. , 0.5 , 0.3333],
       [0.25 , 0.2 , 0.1667]])

In [57]: arr ** 2
Out[57]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```


Сравнение массивов одинакового размера дает булев массив:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
In [59]: arr2
Out[59]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])
```

```
In [60]: arr2 > arr
Out[60]:
array([[False,  True, False],
       [ True, False,  True]])
```

Операции между массивами разного размера называются *укладыванием*, мы будем подробно рассматривать их в приложении А. Глубокое понимание укладывания необязательно для чтения большей части этой книги.

Индексирование и вырезание

Индексирование массивов NumPy – обширная тема, поскольку подмножество массива или его отдельные элементы можно выбрать различными способами. С одномерными массивами все просто; на поверхностный взгляд, они ведут себя как списки Python:

```
In [61]: arr = np.arange(10)

In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [63]: arr[5]
Out[63]: 5

In [64]: arr[5:8]
Out[64]: array([5, 6, 7])

In [65]: arr[5:8] = 12

In [66]: arr
Out[66]: 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 встроенных в Python списков состоит в том, что срез массива является *представлением* исходного массива. Это означает, что данные на самом деле не копируются, а любые изменения, внесенные в представление, попадают и в исходный массив.

Для демонстрации я сначала создам срез массива `arr`:

```
In [67]: arr_slice = arr[5:8]

In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

Если теперь изменить значения в `arr_slice`, то изменения отразятся и на исходном массиве `arr`:

```
In [69]: arr_slice[1] = 12345

In [70]: arr
Out[70]:
array([ 0,  1,  2,  3,  4, 12, 12345, 12,  8,  9])
```

Присваивание неуточненному срезу `[:]` приводит к записи значения во все элементы массива:

```
In [71]: arr_slice[:] = 64

In [72]: arr
Out[72]: 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()`. Ниже мы увидим, что pandas работает так же.

Для массивов большей размерности и вариантов тоже больше. В случае двумерного массива элемент с заданным индексом является не скаляром, а одномерным массивом:

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [74]: arr2d[2]
Out[74]: 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 [75]: arr2d[0][2]
Out[75]: 3

In [76]: arr2d[0, 2]
Out[76]: 3
```

На рис. 4.1 показано индексирование двумерного массива. Лично мне удобно представлять ось 0 как «строки» массива, а ось 1 – как «столбцы».

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

```
In [78]: arr3d
Out[78]:
array([[[ 1,  2,  3],
         [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Тогда `arr3d[0]` – массив размерности 2×3 :

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Выражению `arr3d[0]` можно присвоить как скалярное значение, так и массив:

```
In [80]: old_values = arr3d[0].copy()
```

```
In [81]: arr3d[0] = 42
```

```
In [82]: arr3d
Out[82]:
array([[42, 42, 42],
       [42, 42, 42]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

```
In [83]: arr3d[0] = old_values
```

```
In [84]: arr3d
Out[84]:
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 [85]: arr3d[1, 0]
Out[85]: array([7, 8, 9])
```

Результат такой же, как если бы мы индексировали в два приема:

```
In [86]: x = arr3d[1]
```

```
In [87]: x
Out[87]:
array([[ 7,  8,  9],
       [10, 11, 12]])
```

```
In [88]: x[0]
Out[88]: array([7, 8, 9])
```

Отметим, что во всех случаях, когда выбираются участки массива, результат является представлением.



Этот синтаксис многомерного индексирования, применяемый в NumPy, не будет работать со встроенными в Python объектами, например списками списков.

Индексирование срезами

Как и для одномерных объектов наподобие списков Python, для объектов `ndarray` можно формировать срезы:

```
In [89]: arr
Out[89]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
In [90]: arr[1:6]
Out[90]: array([ 1,  2,  3,  4, 64])
```

Рассмотрим приведенный выше двумерный массив `arr2d`. Применение к нему вырезания дает несколько иной результат:

```
In [91]: arr2d
Out[91]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [92]: arr2d[:2]
Out[92]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Как видите, вырезание производится вдоль оси 0, первой оси. Поэтому срез содержит диапазон элементов вдоль этой оси. Выражение `arr2d[:2]` полезно читать так: «выбрать первые две строки `arr2d`».

Можно указать несколько срезов – как несколько индексов:

```
In [93]: arr2d[:, 1:]  
Out[93]:  
array([[2, 3],  
       [5, 6]])
```

При таком вырезании мы всегда получаем представления массивов с таким же числом измерений, как у исходного. Сочетая срезы и целочисленные индексы, можно получить массивы меньшей размерности.

Например, я могу выбрать вторую строку, а в ней только первые два столбца:

```
In [94]: lower_dim_slice = arr2d[1, :2]
```

Здесь массив `arr2d` двумерный, но `lower_dim_slice` одномерный, а по форме это кортеж с размером по одной оси:

```
In [95]: lower_dim_slice.shape  
Out[95]: (2,)
```

Аналогично я могу выбрать третий столбец, а в нем только первые две строки:

```
In [96]: arr2d[:, 2]  
Out[96]: array([3, 6])
```

Иллюстрация приведена на рис. 4.2. Отметим, что двоеточие без указания числа означает, что нужно взять всю ось целиком, поэтому для вырезания только по осям высших размерностей можно поступить следующим образом:

```
In [97]: arr2d[:, :1]  
Out[97]:  
array([[1],  
       [4],  
       [7]])
```

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

```
In [98]: arr2d[:, 1:] = 0
```

```
In [99]: arr2d  
Out[99]:  
array([[1, 0, 0],  
       [4, 0, 0],  
       [7, 8, 9]])
```

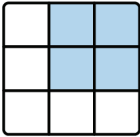
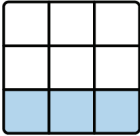
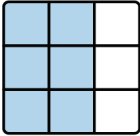
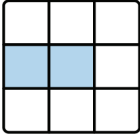
	Выражение	Форма
	<code>arr[:,1:]</code>	(2,2)
	<code>arr[2]</code>	(3,)
	<code>arr[2, :]</code>	(3,)
	<code>arr[2:, :]</code>	(1,3)
	<code>arr[:, :2]</code>	(3,2)
	<code>arr[1, :2]</code>	(2,)
	<code>arr[1:2, :2]</code>	(1,2)

Рис. 4.2. Вырезание из двумерного массива

Булево индексирование

Пусть имеется некоторый массив с данными и массив имен, содержащий дубликаты:

```
In [100]: names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])

In [101]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2],
.....:                    [-12, -4], [3, 4]])

In [102]: names
Out[102]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [103]: data
Out[103]:
array([[ 4,  7],
       [ 0,  2],
       [-5,  6],
       [ 0,  0],
       [ 1,  2],
       [-12, -4],
       [ 3,  4]])
```

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 [104]: names == "Bob"
Out[104]: array([ True, False, False, True, False, False, False])
Этот булев массив можно использовать для индексирования другого массива:
In [105]: data[names == "Bob"]
Out[105]:
array([[4, 7],
       [0, 0]])
```

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

В следующих примерах я выбираю строки, в которых `names == 'Bob'`, и одновременно задаю индекс столбцов:

```
In [106]: data[names == "Bob", 1:]
Out[106]:
array([[7],
       [0]])
```

```
In [107]: data[names == "Bob", 1]
Out[107]: array([7, 0])
```

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

```
In [108]: names != "Bob"
Out[108]: array([False, True, True, False, True, True, True])
```

```
In [109]: ~(names == "Bob")
Out[109]: array([False, True, True, False, True, True, True])
```

```
In [110]: data[~(names == "Bob")]
Out[110]:
array([[ 0,  2],
       [-5,  6],
       [ 1,  2],
       [-12, -4],
       [ 3,  4]])
```

Оператор `~` пригодится, когда требуется инвертировать массив, адресуемый переменной:

```
In [111]: cond = names == "Bob"
```

```
In [112]: data[~cond]
Out[112]:
array([[ 0,  2],
       [-5,  6],
       [ 1,  2],
       [-12, -4],
       [ 3,  4]])
```

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

```
In [113]: mask = (names == "Bob") | (names == "Will")

In [114]: mask
Out[114]: array([ True, False,  True,  True,  True, False, False])

In [115]: data[mask]
Out[115]:
array([[ 4,  7],
       [-5,  6],
       [ 0,  0],
       [ 1,  2]])
```

При выборке данных из массива путем булева индексирования *всегда* создается копия данных, даже если возвращенный массив совпадает с исходным.



Ключевые слова Python `and` и `or` с булевыми массивами не работают. Используйте вместо них `&` (и) и `|` (или).

Если значение устанавливается с применением булева массива, то значения из массива в правой части копируются в левую часть только тогда, когда значение соответствующего элемента булева массива равно `True`. Например, чтобы заменить все отрицательные значения в массиве `data` нулями, нужно всего лишь написать:

```
In [116]: data[data < 0] = 0

In [117]: data
Out[117]:
array([[4, 7],
       [0, 2],
       [0, 6],
       [0, 0],
       [1, 2],
       [0, 0],
       [3, 4]])
```

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

```
In [118]: data[names != "Joe"] = 7

In [119]: data
Out[119]:
array([[7, 7],
       [0, 2],
       [7, 7],
       [7, 7],
       [7, 7],
       [0, 0],
       [3, 4]])
```

Как мы вскоре увидим, такого рода операции над двумерными данными удобно производить в `pandas`.

Прихотливое индексирование

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

```
In [120]: arr = np.zeros((8, 4))
```

```
In [121]: for i in range(8):
.....:     arr[i] = i
```

```
In [122]: arr
Out[122]:
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 [123]: arr[[4, 3, 0, 6]]
Out[123]:
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 [124]: arr[[-3, -5, -7]]
Out[124]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

При передаче нескольких массивов индексов делается несколько иное: выбирается одномерный массив элементов, соответствующих каждому кортежу индексов:

```
In [125]: arr = np.arange(32).reshape((8, 4))
```

```
In [126]: arr
Out[126]:
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 [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[127]: array([ 4, 23, 29, 10])
```

О методе `reshape` мы подробнее поговорим в приложении А.

Здесь отбираются элементы в позициях (1, 0), (5, 3), (7, 1) и (2, 2). Результат прихотливого индексирования в случае, когда целочисленных массивов столько же, сколько осей, всегда одномерный.

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 [128]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[128]:
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 копирует данные в новый массив, когда результат присваивается новой переменной. Если присвоить значения с помощью прихотливого индексирования, то индексированные значения будут модифицированы:

```
In [129]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[129]: array([ 4, 23, 29, 10])
```

```
In [130]: arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0
```

```
In [131]: arr
Out[131]:
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 8,  9,  0, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22,  0],
       [24, 25, 26, 27],
       [28,  0, 30, 31]])
```

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 [132]: arr = np.arange(15).reshape((3, 5))
```

```
In [133]: arr
Out[133]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
In [134]: arr.T
```

```
Out[134]:
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 скалярное произведение двух матриц методом `numpy.dot`:

```
In [135]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])
```

```
In [136]: arr
Out[136]:
array([[ 0,  1,  0],
       [ 1,  2, -2],
       [ 6,  3,  2],
       [-1,  0, -1],
       [ 1,  0,  1]])
```

```
In [137]: np.dot(arr.T, arr)
Out[137]:
array([[39, 20, 12],
       [20, 14,  2],
       [12,  2, 10]])
```

Матрицы можно перемножить и по-другому, воспользовавшись инфиксным оператором `@`:

```
In [138]: arr.T @ arr
Out[138]:
array([[39, 20, 12],
       [20, 14,  2],
       [12,  2, 10]])
```

Обычное транспонирование с помощью `.T` – частный случай перестановки осей. У объекта `ndarray` имеется метод `swapaxes`, который принимает пару номеров осей и меняет их местами, в результате чего данные реорганизуются:

```
In [139]: arr
Out[139]:
array([[ 0,  1,  0],
       [ 1,  2, -2],
       [ 6,  3,  2],
       [-1,  0, -1],
       [ 1,  0,  1]])

In [140]: arr.swapaxes(0, 1)
Out[140]:
array([[ 0,  1,  6, -1,  1],
       [ 1,  2,  3,  0,  0],
       [ 0, -2,  2, -1,  1]])
```

Метод `swapaxes` также возвращает представление без копирования данных.

4.2. ГЕНЕРИРОВАНИЕ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ

Модуль `numpy.random` дополняет встроенный модуль `random` функциями, которые генерируют целые массивы случайных чисел с различными распределениями вероятности. Например, с помощью функции `numpy.random.standard_normal` можно получить случайный массив 4×4, выбранный из стандартного нормального распределения:

```
In [141]: samples = np.random.standard_normal(size=(4, 4))
```

```
In [142]: samples
```

```
Out[142]:
```

```
array([[ -0.2047,  0.4789, -0.5194, -0.5557],
        [ 1.9658,  1.3934,  0.0929,  0.2817],
        [ 0.769 ,  1.2464,  1.0072, -1.2962],
        [ 0.275 ,  0.2289,  1.3529,  0.8864]])
```

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 [143]: from random import normalvariate
```

```
In [144]: N = 1_000_000
```

```
In [145]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.04 s +- 11.4 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

```
In [146]: %timeit np.random.standard_normal(N)
21.9 ms +- 155 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Эти числа не истинно случайные, а *псевдослучайные*, они порождаются настраиваемым детерминированным генератором случайных чисел. Функции типа `numpy.random.standard_normal` пользуются генератором по умолчанию из модуля `numpy.random`, но вы можете сконфигурировать генератор явно:

```
In [147]: rng = np.random.default_rng(seed=12345)
```

```
In [148]: data = rng.standard_normal((2, 3))
```

Здесь аргумент `seed` определяет начальное состояние генератора. Состояние изменяется при каждом использовании объекта `rng` для генерирования случайного числа. Объект `rng` изолирован от прочего кода, в котором может использоваться модуль `numpy.random`:

```
In [149]: type(rng)
```

```
Out[149]: numpy.random._generator.Generator
```

В табл. 4.3 приведен неполный перечень методов таких объектов, как `rng`. Далее в этой главе я буду использовать созданный выше объект `rng` для генерирования случайных данных.

Таблица 4.3. Методы генератора случайных чисел NumPy

Функция	Описание
<code>permutation</code>	Возвращает случайную перестановку последовательности или диапазона
<code>shuffle</code>	Случайным образом переставляет последовательность на месте
<code>integers</code>	Случайная выборка целых чисел из заданного диапазона
<code>standard_normal</code>	Случайная выборка из нормального распределения со средним 0 и стандартным отклонением 1
<code>binomial</code>	Случайная выборка из биномиального распределения
<code>normal</code>	Случайная выборка из нормального (гауссова) распределения
<code>beta</code>	Случайная выборка из бета-распределения
<code>chisquare</code>	Случайная выборка из распределения хи-квадрат
<code>gamma</code>	Случайная выборка из гамма-распределения
<code>uniform</code>	Случайная выборка из равномерного распределения на полуинтервале $[0, 1)$

4.3. УНИВЕРСАЛЬНЫЕ ФУНКЦИИ: БЫСТРЫЕ ПОЭЛЕМЕНТНЫЕ ОПЕРАЦИИ НАД МАССИВАМИ

Универсальной функцией, или *u-функцией*, называется функция, которая выполняет поэлементные операции над данными, хранящимися в объектах `ndarray`. Можно считать, что это векторные обертки вокруг простых функций, которые принимают одно или несколько скалярных значений и порождают один или несколько скалярных результатов.

Многие u-функции – простые поэлементные преобразования, например `numpy.sqrt` или `numpy.exp`:

```
In [150]: arr = np.arange(10)

In [151]: arr
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [152]: np.sqrt(arr)
Out[152]:
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458, 2.8284, 3.    ])

In [153]: np.exp(arr)
Out[153]:
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 `numpy.add` или `numpy.maximum`, принимают 2 массива (и потому называются бинарными) и возвращают один результирующий массив:

```

In [154]: x = rng.standard_normal(8)

In [155]: y = rng.standard_normal(8)

In [156]: x
Out[156]:
array([-1.3678,  0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,
        0.9022])

In [157]: y
Out[157]:
array([-0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ,  1.3223,
       -0.2997])

In [158]: np.maximum(x, y)
Out[158]:
array([-0.467 ,  0.6489,  0.7888, -1.2567,  2.3474,  1.399 ,  1.3223,
        0.9022])

```

Здесь `numpy.maximum` вычисляет поэлементные максимумы массивов `x` и `y`.

Хотя и нечасто, но можно встретить `u`-функцию, возвращающую несколько массивов. Примером может служить `numpy.modf`, векторный вариант встроенной в Python функции `math.modf`: она возвращает дробные и целые части хранящихся в массиве чисел с плавающей точкой:

```

In [159]: arr = rng.standard_normal(7) * 5

In [160]: arr
Out[160]: array([ 4.5146, -8.1079, -0.7909, 2.2474, -6.718 , -0.4084, 8.6237])

In [161]: remainder, whole_part = np.modf(arr)

In [162]: remainder
Out[162]: array([ 0.5146, -0.1079, -0.7909, 0.2474, -0.718 , -0.4084, 0.6237])

In [163]: whole_part
Out[163]: array([ 4., -8., -0., 2., -6., -0., 8.])

```

`U`-функции принимают необязательный аргумент `out`, который позволяет выполнять операции над массивами на месте:

```

In [164]: arr
Out[164]: array([ 4.5146, -8.1079, -0.7909, 2.2474, -6.718 , -0.4084, 8.6237])

In [165]: out = np.zeros_like(arr)

In [166]: np.add(arr, 1)
Out[166]: array([ 5.5146, -7.1079, 0.2091, 3.2474, -5.718 , 0.5916, 9.6237])

In [167]: np.add(arr, 1, out=out)
Out[167]: array([ 5.5146, -7.1079, 0.2091, 3.2474, -5.718 , 0.5916, 9.6237])

In [168]: out
Out[168]: array([ 5.5146, -7.1079, 0.2091, 3.2474, -5.718 , 0.5916, 9.6237])

```

В табл. 4.4 и 4.5 перечислены некоторые `u`-функции. В NumPy добавляются новые `u`-функции, поэтому актуальный перечень лучше всего смотреть в онлайн-документации по NumPy.

Таблица 4.4. Некоторые унарные универсальные функции

Функция	Описание
<code>abs, fabs</code>	Вычислить абсолютную величину целых, вещественных или комплексных элементов массива
<code>sqrt</code>	Вычислить квадратный корень из каждого элемента. Эквивалентно <code>arg ** 0.5</code>
<code>square</code>	Вычислить квадрат каждого элемента. Эквивалентно <code>arg ** 2</code>
<code>exp</code>	Вычислить экспоненту e^x каждого элемента
<code>log, log10, log2, log1p</code>	Натуральный (по основанию e), десятичный, двоичный логарифм и функция <code>log(1 + x)</code> соответственно
<code>sign</code>	Вычислить знак каждого элемента: 1 (для положительных чисел), 0 (для нуля) или -1 (для отрицательных чисел)
<code>ceil</code>	Вычислить для каждого элемента наименьшее целое число, не меньшее его
<code>floor</code>	Вычислить для каждого элемента наибольшее целое число, не большее его
<code>rint</code>	Округлить элементы до ближайшего целого с сохранением <code>dtype</code>
<code>modf</code>	Вернуть дробные и целые части массива в виде отдельных массивов
<code>isnan</code>	Вернуть булев массив, показывающий, какие значения являются NaN (не числами)
<code>isfinite, isinf</code>	Вернуть булев массив, показывающий, какие элементы являются конечными (не <code>inf</code> и не <code>NaN</code>) или бесконечными соответственно
<code>cos, cosh, sin, sinh, tan, tanh</code>	Обычные и гиперболические тригонометрические функции
<code>arccos, arccosh, arcsin, arcsinh, arctan, arctanh</code>	Обратные тригонометрические функции
<code>logical_not</code>	Вычислить значение истинности <code>not x</code> для каждого элемента. Эквивалентно <code>~arg</code>

Таблица 4.5. Некоторые бинарные универсальные функции

Функция	Описание
<code>add</code>	Сложить соответственные элементы массивов
<code>subtract</code>	Вычесть элементы второго массива из соответственных элементов первого
<code>multiply</code>	Перемножить соответственные элементы массивов

Функция	Описание
<code>divide, floor_divide</code>	Деление и деление с отбрасыванием остатка
<code>power</code>	Возвести элементы первого массива в степени, указанные во втором массиве
<code>maximum, fmax</code>	Поэлементный максимум. Функция <code>fmax</code> игнорирует значения NaN
<code>minimum, fmin</code>	Поэлементный минимум. Функция <code>fmin</code> игнорирует значения NaN
<code>mod</code>	Поэлементный модуль (остаток от деления)
<code>copysign</code>	Копировать знаки значений второго массива в соответственные элементы первого массива
<code>greater, greater_equal, less, less_equal, equal, not_equal</code>	Поэлементное сравнение, возвращается булев массив. Эквивалентны инфиксным операторам <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code>
<code>logical_and, logical_or, logical_xor</code>	Вычислить логическое значение истинности логических операций. Эквивалентны инфиксным операторам <code>&</code> , <code> </code> , <code>^</code>

4.4. ПРОГРАММИРОВАНИЕ НА ОСНОВЕ 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 приложении А, я расскажу об *укладывании*, действенном методе векторизации вычислений.

В качестве простого примера предположим, что нужно вычислить функцию $\sqrt{x^2 + y^2}$ на регулярной сетке. Функция `np.meshgrid` принимает два одномерных массива и порождает две двумерные матрицы, соответствующие всем парам (x, y) элементов, взятых из обоих массивов:

```
In [169]: points = np.arange(-5, 5, 0.01) # 100 равноотстоящих точек
```

```
In [170]: xs, ys = np.meshgrid(points, points)
```

```
In [171]: ys
```

```
Out[171]:
```

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

In [173]: z
Out[173]:
array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       ...,
       [7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

Предвосхищая главу 9, я воспользуюсь библиотекой `matplotlib` для визуализации двумерного массива:

```
In [174]: import matplotlib.pyplot as plt

In [175]: plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5, 5])
Out[175]: <matplotlib.image.AxesImage at 0x7f624ae73b20>

In [176]: plt.colorbar()
Out[176]: <matplotlib.colorbar.Colorbar at 0x7f6253e43ee0>

In [177]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values")
Out[177]: Text(0.5, 1.0, 'Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values')
```

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

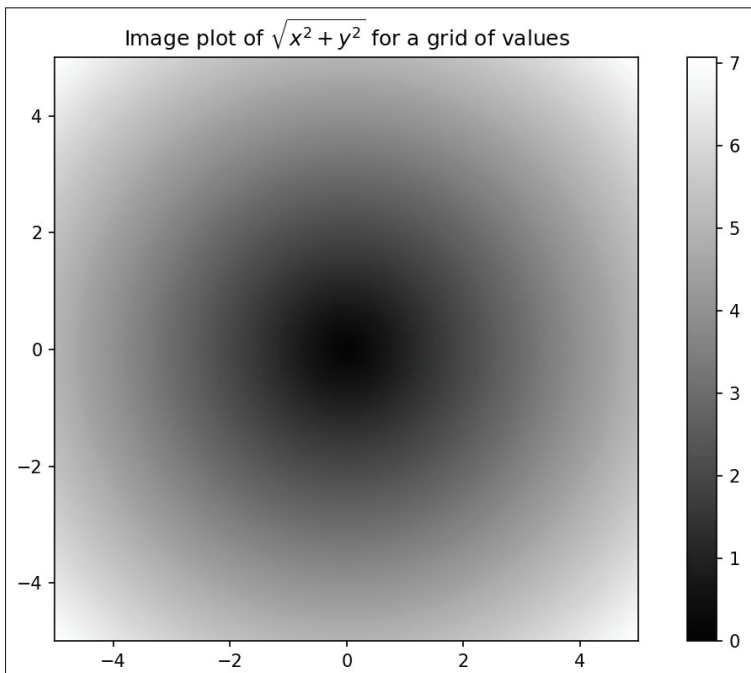


Рис. 4.3. График функции двух переменных на сетке

Если вы работаете в IPython, то для закрытия всех открытых окон с графикой можете выполнить метод `plt.close("all")`:

```
In [179]: plt.close("all")
```



Термин *vectorization* употребляется в информатике и для описания некоторых других понятий, но в этой книге я пользуюсь им для обозначения операций над целыми массивами данных вместо поэлементного применения в цикле `for`.

Запись логических условий в виде операций с массивами

Функция `numpy.where` – векторный вариант тернарного выражения `x if condition else y`. Пусть имеется булев массив и два массива значений:

```
In [180]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
In [182]: cond = np.array([True, False, True, True, False])
```

Допустим, что мы хотим брать значение из массива `xarr`, если соответствующее значение в массиве `cond` равно `True`, а в противном случае – значение из `yarr`. Эту задачу решает такая операция спискового включения:

```
In [183]: result = [(x if c else y)
.....:                for x, y, c in zip(xarr, yarr, cond)]
```

```
In [184]: result
```

```
Out[184]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

Здесь есть 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 `numpy.where` можно написать очень лаконичный код:

```
In [185]: result = np.where(cond, xarr, yarr)
```

```
In [186]: result
```

```
Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Второй и третий аргументы `numpy.where` не обязаны быть массивами – один или оба могут быть скалярами. При анализе данные `where` обычно применяются, чтобы создать новый массив на основе существующего. Предположим, имеется матрица со случайными данными, и мы хотим заменить все положительные значения на 2, а все отрицательные – на -2. С помощью `numpy.where` сделать это очень просто:

```
In [187]: arr = rng.standard_normal((4, 4))
```

```
In [188]: arr
```

```
Out[188]:
```

```
array([[ 2.6182,  0.7774,  0.8286, -0.959 ],
       [-1.2094, -1.4123,  0.5415,  0.7519],
       [-0.6588, -1.2287,  0.2576,  0.3129],
       [-0.1308,  1.27   , -0.093 , -0.0662]])
```

```
In [189]: arr > 0
Out[189]:
array([[ True,  True,  True, False],
       [False, False,  True,  True],
       [False, False,  True,  True],
       [False,  True, False, False]])
```

```
In [190]: np.where(arr > 0, 2, -2)
Out[190]:
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

С помощью `numpy.where` можно комбинировать скаляры и массивы. Например, я могу заменить все положительные элементы `arr` константой 2:

```
In [191]: np.where(arr > 0, 2, arr) # заменить положительные элементы на 2
Out[191]:
array([[ 2.    ,  2.    ,  2.    , -0.959 ],
       [-1.2094, -1.4123,  2.    ,  2.    ],
       [-0.6588, -1.2287,  2.    ,  2.    ],
       [-0.1308,  2.    , -0.093 , -0.0662]])
```

Математические и статистические операции

Среди методов класса массива имеются 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. При использовании таких функций NumPy, как `numpy.sum`, агрегируемый массив следует передавать в первом аргументе.

Ниже я сгенерирую случайные данные с нормальным распределением и вычислю некоторые агрегаты:

```
In [192]: arr = rng.standard_normal((5, 4))

In [193]: arr
Out[193]:
array([[ -1.1082,  0.136 ,  1.3471,  0.0611],
       [ 0.0709,  0.4337,  0.2775,  0.5303],
       [ 0.5367,  0.6184, -0.795 ,  0.3   ],
       [-1.6027,  0.2668, -1.2616, -0.0713],
       [ 0.474 , -0.4149,  0.0977, -1.6404]])

In [194]: arr.mean()
Out[194]: -0.08719744457434529

In [195]: np.mean(arr)
Out[195]: -0.08719744457434529

In [196]: arr.sum()
Out[196]: -1.743948891486906
```

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 [197]: arr.mean(axis=1)
Out[197]: array([ 0.109 , 0.3281, 0.165 , -0.6672, -0.3709])
```

```
In [198]: arr.sum(axis=0)
Out[198]: array([-1.6292, 1.0399, -0.3344, -0.8203])
```

Здесь `arr.mean(axis=1)` означает «вычислить среднее по столбцам», а `arr.sum(axis=0)` – «вычислить сумму по строкам».

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

```
In [199]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [200]: arr.cumsum()
Out[200]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

Для многомерных массивов функция `cumsum` и другие функции с нарастающим итогом возвращают массив того же размера, элементами которого являются частичные агрегаты по указанной оси, вычисленные для каждого среза меньшей размерности:

```
In [201]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

```
In [202]: arr
Out[202]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Выражение `arr.cumsum(axis=0)` вычисляет сумму с накопительным итогом по строкам, а `arr.cumsum(axis=1)` – сумму с накопительным итогом по столбцам:

```
In [203]: arr.cumsum(axis=0)
Out[203]:
array([[0, 1, 2],
       [3, 5, 7],
       [9, 12, 15]])
```

```
In [204]: arr.cumsum(axis=1)
Out[204]:
array([[0, 1, 3],
       [3, 7, 12],
       [6, 13, 21]])
```

Полный список приведен в табл. 4.6. Как многие из этих методов применяются на практике, мы увидим в последующих главах.

Таблица 4.6. 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 NaN
<code>std</code> , <code>var</code>	Стандартное отклонение и дисперсия соответственно

Метод	Описание
<code>min</code> , <code>max</code>	Минимум и максимум
<code>argmin</code> , <code>argmax</code>	Индексы минимального и максимального элементов
<code>cumsum</code>	Сумма с нарастающим итогом с начальным значением 0
<code>cumprod</code>	Произведение с нарастающим итогом с начальным значением 1

Методы булевых массивов

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

```
In [205]: arr = rng.standard_normal(100)

In [206]: (arr > 0).sum() # количество положительных значений
Out[206]: 48

In [207]: (arr <= 0).sum() # количество неположительных значений
Out[207]: 52
```

Здесь скобки в выражении `(arr > 0).sum()` необходимы, чтобы функция `sum()` применялась к временному результату вычисления `arr > 0`.

Существует еще два метода, `any` и `all`, особенно полезных в случае булевых массивов. Метод `any` проверяет, есть ли в массиве хотя бы одно значение, равное `True`, а `all` — что все значения в массиве равны `True`:

```
In [208]: bools = np.array([False, False, True, False])

In [209]: bools.any()
Out[209]: True

In [210]: bools.all()
Out[210]: 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 [211]: arr = rng.standard_normal(6)

In [212]: arr
Out[212]: array([ 0.0773, -0.6839, -0.7208, 1.1206, -0.0548, -0.0824])

In [213]: arr.sort()

In [214]: arr
Out[214]: array([-0.7208, -0.6839, -0.0824, -0.0548, 0.0773, 1.1206])
```

Любой одномерный участок многомерного массива можно отсортировать на месте, передав методу `sort` номер оси. В этом примере:

```
In [215]: arr = rng.standard_normal((5, 3))
```

```
In [216]: arr
```

```
Out[216]:
```

```
array([[ 0.936 ,  1.2385,  1.2728],
       [ 0.4059, -0.0503,  0.2893],
       [ 0.1793,  1.3975,  0.292 ],
       [ 0.6384, -0.0279,  1.3711],
       [-2.0528,  0.3805,  0.7554]])
```

Вызов `arr.sort(axis=0)` сортирует значения в каждом столбце, в `arr.sort(axis=1)` – в каждой строке:

```
In [217]: arr.sort(axis=0)
```

```
In [218]: arr
```

```
Out[218]:
```

```
array([[ -2.0528, -0.0503,  0.2893],
       [ 0.1793, -0.0279,  0.292 ],
       [ 0.4059,  0.3805,  0.7554],
       [ 0.6384,  1.2385,  1.2728],
       [ 0.936 ,  1.3975,  1.3711]])
```

```
In [219]: arr.sort(axis=1)
```

```
In [220]: arr
```

```
Out[220]:
```

```
array([[ -2.0528, -0.0503,  0.2893],
       [-0.0279,  0.1793,  0.292 ],
       [ 0.3805,  0.4059,  0.7554],
       [ 0.6384,  1.2385,  1.2728],
       [ 0.936 ,  1.3711,  1.3975]])
```

Метод верхнего уровня `numpy.sort` возвращает отсортированную копию массива (как встроенная функция `sorted`), а не сортирует массив на месте. Например:

```
In [221]: arr2 = np.array([5, -10, 7, 1, 0, -3])
```

```
In [222]: sorted_arr2 = np.sort(arr2)
```

```
In [223]: sorted_arr2
```

```
Out[223]: array([-10, -3, 0, 1, 5, 7])
```

Дополнительные сведения о методах сортировки в NumPy и о более сложных приемах, например косвенной сортировке, см. в приложении А. В библиотеке `pandas` есть еще несколько операций, относящихся к сортировке (например, сортировка таблицы по одному или нескольким столбцам).

Устранение дубликатов и другие теоретико-множественные операции

В NumPy имеются основные теоретико-множественные операции для одномерных массивов. Пожалуй, самой употребительной является `numpy.unique`, она возвращает отсортированное множество уникальных значений в массиве:

```
In [224]: names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])
```

```
In [225]: np.unique(names)
Out[225]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
In [226]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
```

```
In [227]: np.unique(ints)
Out[227]: array([1, 2, 3, 4])
```

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

```
In [228]: sorted(set(names))
Out[228]: ['Bob', 'Joe', 'Will']
```

Во многих случаях вариант из NumPy быстрее и возвращает массив NumPy, а не список Python.

Функция `numpy.in1d` проверяет, присутствуют ли значения из одного массива в другом, и возвращает булев массив:

```
In [229]: values = np.array([6, 0, 0, 3, 2, 5, 6])
```

```
In [230]: np.in1d(values, [2, 3, 6])
Out[230]: array([ True, False, False,  True,  True, False,  True])
```

В табл. 4.7 перечислены все теоретико-множественные функции, имеющиеся в NumPy.

Таблица 4.7. Теоретико-множественные операции с массивами

Метод	Описание
<code>unique(x)</code>	Вычисляет отсортированное множество уникальных элементов
<code>intersect1d(x, y)</code>	Вычисляет отсортированное множество элементов, общих для <code>x</code> и <code>y</code>
<code>union1d(x, y)</code>	Вычисляет отсортированное объединение элементов
<code>in1d(x, y)</code>	Вычисляет булев массив, показывающий, какие элементы <code>x</code> встречаются в <code>y</code>
<code>setdiff1d(x, y)</code>	Вычисляет разность множеств, т. е. элементы, принадлежащие <code>x</code> , но не принадлежащие <code>y</code>
<code>setxor1d(x, y)</code>	Симметрическая разность множеств; элементы, принадлежащие одному массиву, но не обоим сразу

4.5. Файловый ввод-вывод массивов

NumPy умеет сохранять на диске и загружать с диска данные в текстовом или двоичном формате. В этом разделе мы рассмотрим только встроенный

в NumPy двоичный формат, поскольку для загрузки текстовых и табличных данных большинство пользователей предпочитают pandas и другие инструменты (дополнительные сведения см. в главе 6).

`numpy.save` и `numpy.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 [231]: arr = np.arange(10)
```

```
In [232]: 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 `numpy.load`:

```
In [233]: np.load("some_array.npy")
```

```
Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Можно сохранить несколько массивов в zip-архиве с помощью функции `numpy.savez`, которой массивы передаются в виде именованных аргументов:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

При считывании npz-файла мы получаем похожий на словарь объект, который лениво загружает отдельные массивы:

```
In [235]: arch = np.load("array_archive.npz")
```

```
In [236]: arch["b"]
```

```
Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Если данные хорошо сжимаются, то можно вместо этого использовать метод `numpy.savez_compressed`:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

4.6. ЛИНЕЙНАЯ АЛГЕБРА

Линейно-алгебраические операции – умножение и разложение матриц, вычисление определителей и другие – важная часть любой библиотеки для работы с массивами. В отличие от некоторых языков, например MATLAB, в NumPy применение оператора `*` к двум двумерным массивам вычисляет поэлементное, а не матричное произведение. А для перемножения матриц имеется функция `dot` – как в виде метода массива, так и в виде функции в пространстве имен `numpy`:

```
In [241]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [242]: y = np.array([[6., 23.], [-1, 7], [8, 9]])
```

```
In [243]: x
```

```
Out[243]:  
array([[1., 2., 3.],  
       [4., 5., 6.]])
```

```
In [244]: y
```

```
Out[244]:
```



```
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
```

```
In [245]: x.dot(y)
Out[245]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

`x.dot(y)` эквивалентно `np.dot(x, y)`:

```
In [246]: np.dot(x, y)
Out[246]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

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

```
In [247]: x @ np.ones(3)
Out[247]: array([ 6., 15.] )
```

В модуле `numpy.linalg` имеет стандартный набор алгоритмов, в частности разложение матриц, нахождение обратной матрицы и вычисление определителя:

```
In [248]: from numpy.linalg import inv, qr
```

```
In [249]: X = rng.standard_normal((5, 5))
```

```
In [250]: mat = X.T @ X
```

```
In [251]: inv(mat)
Out[251]:
array([[ 3.4993,  2.8444,  3.5956, -16.5538,  4.4733],
       [ 2.8444,  2.5667,  2.9002, -13.5774,  3.7678],
       [ 3.5956,  2.9002,  4.4823, -18.3453,  4.7066],
       [-16.5538, -13.5774, -18.3453,  84.0102, -22.0484],
       [ 4.4733,  3.7678,  4.7066, -22.0484,  6.0525]])
```

```
In [252]: mat @ inv(mat)
Out[252]:
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.]])
```

Выражение `X.T.dot(X)` вычисляет произведение матрицы `X` на транспонированную к ней матрицу `X.T`.

В табл. 4.8 перечислены наиболее употребительные линейно-алгебраические функции.

Таблица 4.8. 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
<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

4.7. ПРИМЕР: СЛУЧАЙНОЕ БЛУЖДЕНИЕ

Проиллюстрируем операции с массивами на примере случайного блуждания (https://en.wikipedia.org/wiki/Random_walk). Сначала рассмотрим простое случайное блуждание с начальной точкой 0 и шагами 1 и –1, выбираемыми с одинаковой вероятностью.

Вот реализация одного случайного блуждания с 1000 шагов на чистом Python с помощью встроенного модуля `random`:

```
#!/ blockstart
import random
position = 0
walk = [position]
nsteps = 1000
for _ in range(nsteps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
#!/ blockend
```

На рис. 4.4 показаны первые 100 значений такого случайного блуждания.

```
In [255]: plt.plot(walk[:100])
```

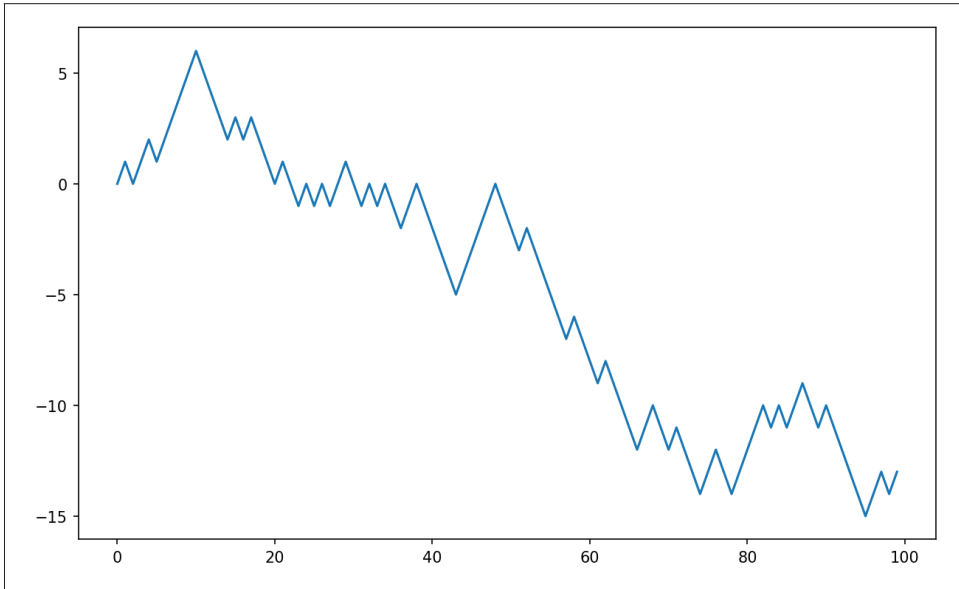


Рис. 4.4. Простое случайное блуждание

Наверное, вы обратили внимание, что `walk` – это просто сумма с нарастающим итогом случайных шагов, которую можно вычислить как выражение-массив. Поэтому я воспользуюсь модулем `numpy.random`, чтобы за один присест подбросить 1000 монет с исходами 1 и -1 и вычислить нарастающую сумму с нарастающим итогом:

```
In [256]: nsteps = 1000

In [257]: rng = np.random.default_rng(seed=12345) # Новый генератор случайных чисел

In [258]: draws = rng.integers(0, 2, size=nsteps)

In [259]: steps = np.where(draws == 0, 1, -1)

In [260]: walk = steps.cumsum()
```

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

```
In [261]: walk.min()
Out[261]: -8

In [262]: walk.max()
Out[262]: 50
```

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 [263]: (np.abs(walk) >= 10).argmax()
Out[263]: 155
```

Отметим, что использование здесь `argmax` не всегда эффективно, потому что она всегда просматривает весь массив. В данном частном случае мы знаем, что первое же встретившееся значение `True` является максимальным.

Моделирование сразу нескольких случайных блужданий

Если бы нам требовалось смоделировать много случайных блужданий, скажем 5000, то это можно было бы сделать путем совсем небольшой модификации приведенного выше кода. Если функциям из модуля `numpy.random` передать 2-кортеж, то они сгенерируют двумерный массив случайных чисел, и мы сможем вычислить нарастающие суммы по строкам, т. е. все 5000 случайных блужданий за одну операцию:

```
In [264]: nwalks = 5000

In [265]: nsteps = 1000

In [266]: draws = rng.integers(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [267]: steps = np.where(draws > 0, 1, -1)

In [268]: walks = steps.cumsum(axis=1)

In [269]: walks
Out[269]:
array([[ 1,  2,  3, ..., 22, 23, 22],
       [ 1,  0, -1, ..., -50, -49, -48],
       [ 1,  2,  3, ..., 50, 49, 48],
       ...,
       [-1, -2, -1, ..., -10, -9, -10],
       [-1, -2, -3, ...,  8,  9,  8],
       [-1,  0,  1, ..., -4, -3, -2]])
```

Теперь мы можем вычислить максимум и минимум по всем блужданиям:

```
In [270]: walks.max()
Out[270]: 114

In [271]: walks.min()
Out[271]: -120
```

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 [272]: hits30 = (np.abs(walks) >= 30).any(axis=1)

In [273]: hits30
```

```
Out[273]: array([False, True, True, ..., True, False, True])
```

```
In [274]: hits30.sum() # сколько раз достигалось 30 или -30
Out[274]: 3395
```

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 [275]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(axis=1)
```

```
In [276]: crossing_times
Out[276]: array([201, 491, 283, ..., 219, 259, 541])
```

И наконец, вычислим среднее минимальное время пересечения:

```
In [277]: crossing_times.mean()
Out[277]: 500.5699558173785
```

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

```
In [278]: draws = 0.25 * rng.standard_normal((nwalks, nsteps))
```



Помните, что при таком векторизованном подходе необходимо создать массив, содержащий `nwalks * nsteps` элементов; для продолжительного моделирования он может занимать много памяти. Если память ограничена, то придется применить какой-то другой подход.

4.8. ЗАКЛЮЧЕНИЕ

Хотя большая часть книги посвящена выработке навыков манипулирования данными в `pandas`, мы и дальше будем работать с массивами в таком же стиле. В приложении А мы более глубоко рассмотрим возможности NumPy и расскажем о дополнительных приемах вычислений с массивами.

Первое знакомство с pandas

Библиотека pandas будет нашим основным инструментом в оставшейся части книги. Она содержит структуры данных и средства манипуляции данными, спроектированные с целью максимально упростить и ускорить очистку и анализ данных в Python. Pandas часто используется в сочетании с такими инструментами численных расчетов, как NumPy и SciPy, аналитическими библиотеками statsmodels и scikit-learn и библиотеками визуализации данных типа matplotlib. Pandas переняла от NumPy многое из идиоматического стиля работы с массивами, в особенности функции манипуляции массивами и стремление избегать циклов `for` при обработке данных.

Хотя pandas взяла на вооружение многие идиомы кодирования NumPy, между этими библиотеками есть важное различие. Pandas предназначена для работы с табличными или неоднородными данными. Напротив, NumPy больше подходит для работы с однородными числовыми данными, организованными в виде массивов.

Pandas стала проектом с открытым исходным кодом в 2010 году и с тех пор превратилась в довольно большую библиотеку, применяемую в самых разных практических приложениях. Количество соработчиков перевалило за 2500, все они так или иначе помогают развивать проект, который используют в своей повседневной работе. Бьющие жизнью сообщества разработчиков и пользователей pandas являются ключевым элементом ее успеха.



Многие не знают, что я не участвую в процессе развития pandas с 2013 года; с того времени проект управляется только сообществом. Обязательно поблагодарите разработчиков ядра и всех, кто вносит свой вклад, за их нелегкую работу!

В этой книге используются следующие соглашения об импорте NumPy и pandas:

```
In [1]: import numpy as np
```

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

Стало быть, встретив в коде `pd.`, знайте, что имеется в виду pandas. Возможно, вы захотите также импортировать в локальное пространство имен классы `Series` и `DataFrame`, потому что они используются часто:

```
In [3]: from pandas import Series, DataFrame
```

5.1. ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ PANDAS

Чтобы начать работу с pandas, вы должны освоить две основные структуры данных: *Series* и *DataFrame*. Они, конечно, не являются универсальным решением любой задачи, но все же образуют солидную и простую для использования основу большинства приложений.

Объект Series

Series – одномерный похожий на массив объект, содержащий последовательность данных (типов, похожих на поддерживаемые NumPy) и ассоциированный с ним массив меток, который называется *индексом*. Простейший объект *Series* состоит только из массива данных:

```
In [14]: obj = pd.Series([4, 7, -5, 3])
```

```
In [15]: obj
```

```
Out[15]:
```

```
0    4
```

```
1    7
```

```
2   -5
```

```
3    3
```

```
dtype: int64
```

В строковом представлении *Series*, отображаемом в интерактивном режиме, индекс находится слева, а значения – справа. Поскольку мы не задали индекс для данных, то по умолчанию создается индекс, состоящий из целых чисел от 0 до $N - 1$ (где N – длина массива данных). Имея объект *Series*, получить представление самого массива и его индекса можно с помощью атрибутов *values* и *index* соответственно:

```
In [16]: obj.array
```

```
Out[16]:
```

```
<PandasArray>
```

```
[4, 7, -5, 3]
```

```
Length: 4, dtype: int64
```

```
In [17]: obj.index
```

```
Out[17]: RangeIndex(start=0, stop=4, step=1))
```

Результатом применения атрибута *.array* является объект *PandasArray*, который обычно обортывает массив NumPy, но может также содержать специальные типы расширений массива, которые мы обсудим в разделе 7.3 «Типы расширения данных».

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

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])
```

```
In [19]: obj2
```

```
Out[19]:
```

```
d    4
```

```
b    7
```

```
a   -5
```

```
c    3
```

```
dtype: int64
```

```
In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

В отличие от массивов NumPy, для выделения одного или нескольких значений можно использовать метки в индексе:

```
In [21]: obj2["a"]
Out[21]: -5

In [22]: obj2["d"] = 6

In [23]: obj2[["c", "a", "d"]]
Out[23]:
c    3
a   -5
d    6
dtype: int64
```

Здесь ['c', 'a', 'd'] интерпретируется как список индексов, хотя он содержит не целые числа, а строки.

Функции NumPy или похожие на них операции, например фильтрация с помощью булева массива, скалярное умножение или применение математических функций, сохраняют связь между индексом и значением:

```
In [24]: obj2[obj2 > 0]
Out[24]:
d    6
b    7
c    3
dtype: int64

In [25]: obj2 * 2
Out[25]:
d    12
b    14
a   -10
c     6
dtype: int64

In [26]: import numpy as np

In [27]: np.exp(obj2)
Out[27]:
d    403.428793
b   1096.633158
a     0.006738
c    20.085537
dtype: float64
```

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 [28]: "b" in obj2
Out[28]: True

In [29]: "e" in obj2
Out[29]: False
```


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

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [31]: obj3 = pd.Series(sdata)

In [32]: obj3
Out[32]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Объект Series можно преобразовать обратно в словарь методом `to_dict`:

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

Если передается только словарь, то в индексе получившегося объекта Series ключи будут храниться в порядке, который определяется методом словаря `keys` и зависит от того, в каком порядке ключи вставлялись. Этот порядок можно переопределить, передав индекс, содержащий ключи словаря в том порядке, в каком они должны находиться в результирующем объекте Series:

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]

In [35]: obj4 = pd.Series(sdata, index=states)

In [36]: obj4
Out[36]:
California      NaN
Ohio            35000.0
Oregon          16000.0
Texas           71000.0
dtype: float64
```

В данном случае три значения, найденных в `sdata`, помещены в соответствующие им позиции, а для метки `'California'` никакого значения не нашлось, поэтому ей соответствует признак `NaN` (не число), которым в pandas обозначаются отсутствующие значения. Поскольку строки `'Utah'` не было в списке `states`, то ее нет и в результирующем объекте.

Говоря об отсутствующих данных, я буду употреблять термины «NA» и «null» как синонимы. Для распознавания отсутствующих данных в pandas следует использовать функции `isna` и `notna`:

```
In [37]: pd.isna(obj4)
Out[37]:
California      True
Ohio            False
Oregon          False
Texas           False
dtype: bool

In [38]: pd.notna(obj4)
Out[38]:
```

```
California    False
Ohio          True
Oregon        True
Texas         True
dtype: bool
```

У объекта Series есть также методы экземпляра:

```
In [39]: obj4.isna()
Out[39]:
California    True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

Более подробно работа с отсутствующими данными будет обсуждаться в главе 7.

Во многих приложениях полезно, что при выполнении арифметических операций объект Series автоматически выравнивает данные по индексной метке:

```
In [40]: obj3
Out[40]:
Ohio      35000
Texas     71000
Oregon    16000
Utah      5000
dtype: int64
```

```
In [41]: obj4
Out[41]:
California    NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [42]: obj3 + obj4
Out[42]:
California    NaN
Ohio          70000.0
Oregon        32000.0
Texas        142000.0
Utah          NaN
dtype: float64
```

Вопрос о выравнивании данных будет подробнее рассмотрен ниже. Если у вас имеется опыт работы с базами данных, то можете считать, что это аналог операции соединения.

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

```
In [43]: obj4.name = "population"
```

```
In [44]: obj4.index.name = "state"
```

```
In [45]: obj4
Out[45]:
state
California    NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Индекс объекта Series можно изменить на месте с помощью присваивания:

```
In [46]: obj
Out[46]:
0    4
1    7
2   -5
3    3
dtype: int64

In [47]: obj.index = ["Bob", "Steve", "Jeff", "Ryan"]

In [48]: obj
Out[48]:
Bob      4
Steve    7
Jeff   -5
Ryan     3
dtype: int64
```

Объект DataFrame

Объект DataFrame представляет табличную структуру данных, состоящую из упорядоченной коллекции столбцов, причем типы значений (числовой, строковый, булев и т. д.) в разных столбцах могут различаться. В объекте DataFrame хранятся два индекса: по строкам и по столбцам. Можно считать, что это словарь объектов Series, имеющих общий индекс.



Хотя в DataFrame данные хранятся в двумерном формате, в виде таблицы, нетрудно представить и данные более высокой размерности, если воспользоваться иерархическим индексированием. Эту тему мы обсудим в следующем разделе, она лежит в основе многих продвинутых механизмов обработки данных в pandas.

Есть много способов сконструировать объект DataFrame, один из самых распространенных – на основе словаря списков одинаковой длины или массивов NumPy:

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

Получившемуся DataFrame автоматически будет назначен индекс, как и в случае Series, и столбцы расположатся в порядке, определяемом порядком

ключей в `data` (который зависит от того, в каком порядке ключи вставлялись в словарь):

```
In [50]: frame
Out[50]:
   state  year  pop
0  Ohio   2000  1.5
1  Ohio   2001  1.7
2  Ohio   2002  3.6
3  Nevada 2001  2.4
4  Nevada 2002  2.9
5  Nevada 2003  3.2
```



В Jupyter-блокноте объекты `DataFrame` отображаются в виде HTML-таблицы, более удобной для браузера. Пример см. на рис. 5.1.

The screenshot shows a Jupyter Notebook interface. The input cell contains the code `In [19]: frame`. The output cell displays the DataFrame as an HTML table with the following content:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Рис. 5.1. Вид объектов `pandas DataFrame` в Jupyter

Для больших объектов `DataFrame` метод `head` отбирает только первые пять строк:

```
In [51]: frame.head()
Out[51]:
   state  year  pop
0  Ohio   2000  1.5
1  Ohio   2001  1.7
2  Ohio   2002  3.6
3  Nevada 2001  2.4
4  Nevada 2002  2.9
```

Аналогично `tail` возвращает последние пять строк:

```
In [52]: frame.tail()
Out[52]:
   state  year  pop
1  Ohio   2001  1.7
2  Ohio   2002  3.6
```

```
3 Nevada 2001 2.4
4 Nevada 2002 2.9
5 Nevada 2003 3.2
```

Если задать последовательность столбцов, то столбцы DataFrame расположатся строго в указанном порядке:

```
In [53]: pd.DataFrame(data, columns=["year", "state", "pop"])
Out[53]:
   year  state  pop
0 2000   Ohio  1.5
1 2001   Ohio  1.7
2 2002   Ohio  3.6
3 2001 Nevada  2.4
4 2002 Nevada  2.9
5 2003 Nevada  3.2
```

Если запросить столбец, которого нет в `data`, то он будет заполнен значениями `NaN`:

```
In [54]: frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])

In [55]: frame2
Out[55]:
   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
5 2003 Nevada  3.2   NaN
```

```
In [56]: frame2.columns
Out[56]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

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

```
In [57]: frame2["state"]
Out[57]:
0    Ohio
1    Ohio
2    Ohio
3  Nevada
4  Nevada
5  Nevada
Name: state, dtype: object

In [58]: frame2.year
Out[58]:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```



Возможность доступа к столбцам как к атрибутам (например, `frame2.year`) и автозавершение имен столбцов по нажатию **Tab** предоставляются в IPython для удобства.

Синтаксис `frame2[column]` работает для любого имени столбца, а `frame2.column` – только когда имя столбца – допустимое имя переменной Python, не конфликтующее с именами методов DataFrame. Например, если имя столбца содержит пробелы или еще какие-то специальные символы, отличные от знака подчеркивания, то употреблять его в качестве имени атрибута после точки нельзя.

Отметим, что возвращенный объект Series имеет тот же индекс, что и DataFrame, а его атрибут `name` установлен соответствующим образом.

Строки также можно извлечь по позиции или по имени с помощью специальных атрибутов `iloc` и `loc` (подробнее об этом см. в разделе «Выборка из DataFrame с помощью `loc` и `iloc`» ниже):

```
In [59]: frame2.loc[1]
Out[59]:
year      2001
state     Ohio
pop        1.7
debt      NaN
Name: 1, dtype: object
```

```
In [60]: frame2.iloc[2]
Out[60]:
year      2002
state     Ohio
pop        3.6
debt      NaN
Name: 2, dtype: object
```

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

```
In [61]: frame2["debt"] = 16.5
```

```
In [62]: frame2
Out[62]:
```

```
   year  state pop debt
0  2000   Ohio  1.5  16.5
1  2001   Ohio  1.7  16.5
2  2002   Ohio  3.6  16.5
3  2001 Nevada  2.4  16.5
4  2002 Nevada  2.9  16.5
5  2003 Nevada  3.2  16.5
```

```
In [63]: frame2["debt"] = np.arange(6.)
```

```
In [64]: frame2
Out[64]:
```

```

year  state pop debt
0 2000   Ohio  1.5  0.0
1 2001   Ohio  1.7  1.0
2 2002   Ohio  3.6  2.0
3 2001 Nevada  2.4  3.0
4 2002 Nevada  2.9  4.0
5 2003 Nevada  3.2  5.0

```

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

```

In [65]: val = pd.Series([-1.2, -1.5, -1.7], index=["two", "four", "five"])

In [66]: frame2["debt"] = val

In [67]: frame2
Out[67]:
   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
5 2003 Nevada  3.2 NaN

```

Присваивание несуществующему столбцу приводит к созданию нового столбца.

Для удаления столбцов служит ключевое слово `del`, как и в обычном словаре. Для демонстрации работы `del` я сначала добавлю новый столбец булевых признаков, показывающих, находится ли в столбце `state` значение "Ohio":

```

In [68]: frame2["eastern"] = frame2["state"] == "Ohio"

In [69]: frame2
Out[69]:
   year  state pop debt eastern
0 2000   Ohio  1.5 NaN    True
1 2001   Ohio  1.7 NaN    True
2 2002   Ohio  3.6 NaN    True
3 2001 Nevada  2.4 NaN   False
4 2002 Nevada  2.9 NaN   False
5 2003 Nevada  3.2 NaN   False

```



Новый столбец нельзя создать, пользуясь синтаксисом `frame2.eastern`.

Затем для удаления этого столбца я воспользуюсь методом `del`:

```

In [70]: del frame2["eastern"]

In [71]: frame2.columns
Out[71]: 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 явно вызвать метод `copy` объекта Series.

Еще одна распространенная форма данных – словарь словарей:

```
In [72]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},
.....:                  "Nevada": {2001: 2.4, 2002: 2.9}}
```

Если передать вложенный словарь объекту DataFrame, то pandas интерпретирует ключи внешнего словаря как столбцы, а ключи внутреннего словаря – как индексы строк:

```
In [73]: frame3 = pd.DataFrame(populations)
```

```
In [74]: frame3
Out[74]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

Объект DataFrame можно транспонировать (переставить местами строки и столбцы), воспользовавшись таким же синтаксисом, как для словарей NumPy:

```
In [75]: frame3.T
Out[75]:
```

	2000	2001	2002
Ohio	1.5	1.7	3.6
Nevada	NaN	2.4	2.9



Отметим, что операция транспонирования стирает типы данных в столбцах, если не все столбцы имеют один и тот же тип данных. Поэтому при двукратном транспонировании прежняя информация о типах может быть потеряна. В этом случае столбцы становятся массивами стандартных объектов Python.

Ключи внутренних словарей объединяются для образования индекса результата. Однако этого не происходит, если индекс задан явно:

```
In [76]: pd.DataFrame(populations, index=[2001, 2002, 2003])
Out[76]:
```

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9
2003	NaN	NaN

Словари объектов Series интерпретируются очень похоже:

```
In [77]: pdata = {"Ohio": frame3["Ohio"][: -1],
.....:            "Nevada": frame3["Nevada"][: 2]}
In [78]: pd.DataFrame(pdata)
Out[78]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4

Полный перечень возможных аргументов конструктора `DataFrame` приведен в табл. 5.1.

Таблица 5.1. Аргументы конструктора `DataFrame`

Тип	Примечания
Двумерный <code>ndarray</code>	Матрица данных, дополнительно можно передать метки строк и столбцов
Словарь массивов, списков или кортежей	Каждая последовательность становится столбцом объекта <code>DataFrame</code> . Все последовательности должны быть одинаковой длины
Структурированный массив NumPy	Интерпретируется так же, как «словарь массивов»
Словарь объектов <code>Series</code>	Каждое значение становится столбцом. Если индекс явно не задан, то индексы объектов <code>Series</code> объединяются и образуют индекс строк результата
Словарь словарей	Каждый внутренний словарь становится столбцом. Ключи объединяются и образуют индекс строк, как в случае «словаря объектов <code>Series</code> »
Список словарей или объектов <code>Series</code>	Каждый элемент списка становится строкой объекта <code>DataFrame</code> . Объединение ключей словаря или индексов объектов <code>Series</code> становится множеством меток столбцов <code>DataFrame</code>
Список списков или кортежей	Интерпретируется так же, как «двумерный <code>ndarray</code> »
Другой объект <code>DataFrame</code>	Используются индексы <code>DataFrame</code> , если явно не заданы другие индексы
Объект NumPy <code>MaskedArray</code>	Как «двумерный <code>ndarray</code> », с тем отличием, что замаскированные значения становятся отсутствующими в результирующем объекте <code>DataFrame</code>

Если у объектов, возвращаемых при обращении к атрибутам `index` и `columns` объекта `DataFrame`, имеется атрибут `name`, то он также выводится:

```
In [79]: frame3.index.name = "year"

In [80]: frame3.columns.name = "state"

In [81]: frame3
Out[81]:
state  Ohio Nevada
year
2000   1.5      NaN
2001   1.7      2.4
2002   3.6      2.9
```

В отличие от `Series`, объект `DataFrame` не имеет атрибута `name`. Метод `to_numpy` объекта `DataFrame` возвращает хранящиеся в нем данные в виде двумерного массива `ndarray`:

```
In [82]: frame3.to_numpy()
Out[82]:
array([[1.5, nan],
       [1.7, 2.4],
       [3.6, 2.9]])
```

Если у столбцов DataFrame разные типы данных, то тип данных массива `values` будет выбран так, чтобы охватить все столбцы:

```
In [83]: frame2.to_numpy()
Out[83]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, nan],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, nan],
       [2002, 'Nevada', 2.9, nan],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

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 [84]: obj = pd.Series(np.arange(3), index=["a", "b", "c"])

In [85]: index = obj.index

In [86]: index
Out[86]: Index(['a', 'b', 'c'], dtype='object')

In [87]: index[1:]
Out[87]: Index(['b', 'c'], dtype='object')
```

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

```
index[1] = "d" # TypeError
```

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

```
In [88]: labels = pd.Index(np.arange(3))

In [89]: labels
Out[89]: Int64Index([0, 1, 2], dtype='int64')

In [90]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [91]: obj2
Out[91]:
0    1.5
1   -2.5
2    0.0
dtype: float64
```

```
In [92]: obj2.index is labels
Out[92]: True
```



Некоторые пользователи редко используют преимущества, предоставляемые индексами, но поскольку существуют операции, которые возвращают результат, содержащий индексированные данные, важно понимать, как индексы работают.

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

```
In [93]: frame3
Out[93]:
state Ohio Nevada
year
2000    1.5    NaN
2001    1.7    2.4
2002    3.6    2.9

In [94]: frame3.columns
Out[94]: Index(['Ohio', 'Nevada'], dtype='object', name='state')

In [95]: "Ohio" in frame3.columns
Out[95]: True

In [96]: 2003 in frame3.index
Out[96]: False
```

В отличие от множеств Python, индекс в pandas может содержать повторяющиеся метки:

```
In [97]: pd.Index(["foo", "foo", "bar", "bar"])
Out[97]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Если при выборке указана повторяющаяся метка, то будут выбраны все элементы с такой меткой.

У любого объекта Index есть ряд свойств и методов для ответа на типичные вопросы о хранящихся в нем данных. Наиболее полезные перечислены в табл. 5.2.

Таблица 5.2. Некоторые методы и свойства объекта Index

Метод	Описание
<code>append()</code>	Конкатенирует с дополнительными индексными объектами, порождая новый объект Index
<code>difference()</code>	Вычисляет теоретико-множественную разность, представляя ее в виде индексного объекта
<code>intersection()</code>	Вычисляет теоретико-множественное пересечение
<code>union()</code>	Вычисляет теоретико-множественное объединение
<code>isin()</code>	Вычисляет булев массив, показывающий, содержится ли каждое значение индекса в переданной коллекции

Метод	Описание
<code>delete()</code>	Вычисляет новый индексный объект, получающийся после удаления элемента с индексом <code>i</code>
<code>drop()</code>	Вычисляет новый индексный объект, получающийся после удаления переданных значений
<code>insert()</code>	Вычисляет новый индексный объект, получающийся после вставки элемента в позицию с индексом <code>i</code>
<code>is_monotonic()</code>	Возвращает True, если каждый элемент больше или равен предыдущему
<code>is_unique()</code>	Возвращает True, если в индексе нет повторяющихся значений
<code>unique()</code>	Вычисляет массив уникальных значений в индексе

5.2. БАЗОВАЯ ФУНКЦИОНАЛЬНОСТЬ

В этом разделе мы рассмотрим фундаментальные основы взаимодействия с данными, хранящимися в объектах `Series` и `DataFrame`. В последующих главах 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 путем чтения онлайн-документации.

Переиндексация

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

```
In [98]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])
```

```
In [99]: obj
Out[99]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Если вызвать `reindex` для этого объекта `Series`, то данные будут реорганизованы в соответствии с новым индексом, а если каких-то из имеющихся в этом индексе значений раньше не было, то вместо них будут подставлены значения `NaN`:

```
In [100]: obj2 = obj.reindex(["a", "b", "c", "d", "e"])
```

```
In [101]: obj2
Out[101]:
a   -5.3
b    7.2
```

```
c    3.6
d    4.5
e    NaN
dtype: float64
```

Для упорядоченных данных, например временных рядов, иногда желательно произвести интерполяцию, или восполнение отсутствующих значений в процессе переиндексации. Это позволяет сделать параметр `method`; так, если задать для него значение `ffill`, то вместо отсутствующих значений будут подставлены значения из ближайшей предшествующей строки, имеющейся в индексе:

```
In [102]: obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])

In [103]: obj3
Out[103]:
0    blue
2    purple
4    yellow
dtype: object

In [104]: obj3.reindex(np.arange(6), method="ffill")
Out[104]:
0    blue
1    blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

В случае объекта `DataFrame` метод `reindex` может изменять строки, столбцы или то и другое. Если передать ему просто последовательность, то в результирующем объекте переиндексируются строки:

```
In [105]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
.....:                        index=["a", "c", "d"],
.....:                        columns=["Ohio", "Texas", "California"])

In [106]: frame
Out[106]:
   Ohio Texas California
a     0     1         2
c     3     4         5
d     6     7         8

In [107]: frame2 = frame.reindex(index=["a", "b", "c", "d"])

In [108]: frame2
Out[108]:
   Ohio Texas California
a   0.0   1.0         2.0
b   NaN   NaN         NaN
c   3.0   4.0         5.0
d   6.0   7.0         8.0
```

Столбцы можно переиндексировать, задав ключевое слово `columns`:

```
In [109]: states = ["Texas", "Utah", "California"]
```

```
In [110]: frame.reindex(columns=states)
```

```
Out[110]:
```

```
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Поскольку "Ohio" отсутствует в `states`, данные этого столбца удаляются из результата. Другой способ переиндексации по конкретной оси состоит в том, чтобы передать метки новой оси в качестве позиционного аргумента, а затем задать переиндексируемую ось с помощью именованного аргумента `axis`:

```
In [111]: frame.reindex(states, axis="columns")
```

```
Out[111]:
```

```
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

В табл. 5.3 приведены более полные сведения об аргументах `reindex`.

Таблица 5.3. Аргументы функции `reindex`

Аргумент	Описание
<code>labels</code>	Новая последовательность, которая будет использоваться в качестве индекса. Может быть экземпляром <code>Index</code> или любой другой похожей на последовательность структуры данных Python. Экземпляр <code>Index</code> будет использован «как есть», без копирования
<code>index</code>	Использовать переданную последовательность в качестве новых меток индекса
<code>columns</code>	Использовать переданную последовательность в качестве новых меток столбцов
<code>axis</code>	По какой оси переиндексировать: "index" (строки) или "columns" (столбцы). По умолчанию подразумевается "index". Можно чередовать <code>reindex(index=new_labels)</code> и <code>reindex(columns=new_labels)</code>
<code>method</code>	Метод интерполяции (восполнения): "ffill" (прямое восполнение – по предыдущей строке) или "bfill" (прямое восполнение – по следующей строке)
<code>fill_value</code>	Значение, которое должно подставляться вместо отсутствующих значений, появляющихся в результате переиндексации. Если нужно, чтобы для отсутствующих меток принималось значение NaN, то задавайте <code>fill_value="missing"</code>
<code>limit</code>	При прямом или обратном восполнении максимальная длина восполняемой лакуны (выраженная числом элементов)
<code>tolerance</code>	При прямом или обратном восполнении максимальная длина восполняемой лакуны (выраженная абсолютным расстоянием)
<code>level</code>	Сопоставить с простым объектом <code>Index</code> на указанном уровне <code>MultiIndex</code> , иначе выбрать подмножество
<code>copy</code>	Если <code>True</code> , то всегда копировать данные, даже если новый индекс эквивалентен старому. Если <code>False</code> , то не копировать данные, когда индексы эквивалентны

Ниже, в разделе «Выборка из DataFrame с помощью loc и iloc», мы узнаем, что переиндексировать можно также с помощью оператора `loc`, и многие предпочитают именно этот способ. Он работает, только если все метки нового индекса уже присутствуют в объекте DataFrame (тогда как `reindex` вставляет отсутствующие данные для новых меток):

```
In [112]: frame.loc[["a", "d", "c"], ["California", "Texas"]]
Out[112]:
   California  Texas
a           2      1
d           8      7
c           5      4
```

Удаление элементов из оси

Удалить один или несколько элементов из оси просто, если имеется индексный массив или список, не содержащий этих значений, поскольку можно воспользоваться методом `reindex` или индексированием на базе `.loc`. Чтобы избежать нас от манипулирования данными и теоретико-множественных операций, метод `drop` возвращает новый объект, в котором указанные значения удалены из оси:

```
In [113]: obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])
```

```
In [114]: obj
Out[114]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

```
In [115]: new_obj = obj.drop("c")
```

```
In [116]: new_obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

```
In [117]: obj.drop(["d", "c"])
Out[117]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

В случае DataFrame указанные в аргументе `index` значения можно удалить из любой оси. Для иллюстрации сначала создадим объект DataFrame:

```
In [118]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
.....:                        index=["Ohio", "Colorado", "Utah", "New York"],
.....:                        columns=["one", "two", "three", "four"])
```

```
In [119]: data
Out[119]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Если вызвать `drop`, указав последовательность меток, то будут удалены строки с такими метками (ось 0):

```
In [120]: data.drop(index=["Colorado", "Ohio"])
Out[120]:
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

Чтобы удалить столбцы, используйте именованный аргумент `'columns'`:

```
In [121]: data.drop(columns=["two"])
Out[121]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

Столбцы можно удалить также, передав параметры `axis=1` (как в NumPy) или `axis="columns"`:

```
In [122]: data.drop("two", axis=1)
Out[122]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
In [123]: data.drop(["two", "four"], axis="columns")
Out[123]:
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
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 [124]: obj = pd.Series(np.arange(4.), index=["a", "b", "c", "d"])

In [125]: obj
Out[125]:
```

a	0.0
b	1.0


```
c    2.0
d    3.0
dtype: float64

In [126]: obj["b"]
Out[126]: 1.0

In [127]: obj[1]
Out[127]: 1.0

In [128]: obj[2:4]
Out[128]:
c    2.0
d    3.0
dtype: float64

In [129]: obj[["b", "a", "d"]]
Out[129]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [130]: obj[[1, 3]]
Out[130]:
b    1.0
d    3.0
dtype: float64

In [131]: obj[obj < 2]
Out[131]:
a    0.0
b    1.0
dtype: float64
```

Хотя выбирать данные по метке можно и так, предпочтительнее выбирать значения из индекса с помощью специального оператора `loc`:

```
In [132]: obj.loc[["b", "a", "d"]]
Out[132]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

Использовать `loc` лучше, потому что целые числа обрабатываются при этом не так, как при индексировании с помощью `[]`. В последнем случае целые числа трактуются как метки, если индекс содержит целые числа, поэтому поведение зависит от типа данных в индексе. Например:

```
In [133]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])

In [134]: obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [135]: obj1
Out[135]:
2    1
```

```
0 2
1 3
dtype: int64
```

```
In [136]: obj2
```

```
Out[136]:
```

```
a 1
b 2
c 3
dtype: int64
```

```
In [137]: obj1[[0, 1, 2]]
```

```
Out[137]:
```

```
0 2
1 3
2 1
dtype: int64
```

```
In [138]: obj2[[0, 1, 2]]
```

```
Out[138]:
```

```
a 1
b 2
c 3
dtype: int64
```

При использовании `loc` выражение `obj.loc[[0, 1, 2]]` приведет к ошибке, если индекс содержит не целые числа:

```
In [134]: obj2.loc[[0, 1]]
```

```
-----
KeyError                                Traceback (most recent call last)
/tmp/ipykernel_804589/4185657903.py in <module>
----> 1 obj2.loc[[0, 1]]
```

```
^ LONG EXCEPTION ABBREVIATED ^
```

```
KeyError: "None of [Int64Index([0, 1], dtype='int64')] are in the [index]"
```

Поскольку оператор `loc` индексирует только метками, имеется также оператор `iloc`, который индексирует только целыми числами, чтобы обеспечить единообразную работу вне зависимости от того, содержит индекс целые числа или нет:

```
In [139]: obj1.iloc[[0, 1, 2]]
```

```
Out[139]:
```

```
2 1
0 2
1 3
dtype: int64
```

```
In [140]: obj2.iloc[[0, 1, 2]]
```

```
Out[140]:
```

```
a 1
b 2
c 3
dtype: int64
```



С помощью меток можно также производить вырезание, но оно отличается от обычного вырезания в Python тем, что конечная точка включается:

```
In [141]: obj2.loc["b":"c"]
Out[141]:
b      2
c      3
dtype: int64
```

Присваивание с помощью этих методов модифицирует соответствующий участок Series:

```
In [142]: obj2.loc["b":"c"] = 5
```

```
In [143]: obj2
Out[143]:
a      1
b      5
c      5
dtype: int64
```



Начинающие часто допускают ошибку: пытаются вызывать `loc` или `iloc` как функции вместо использования синтаксиса индексирования с квадратными скобками. Эта нотация применяется, чтобы можно было использовать операции вырезания и индексировать по нескольким осям при работе с объектами DataFrame.

Обращение по индексу к DataFrame предназначено для получения одного или нескольких столбцов путем задания одного значения или последовательности:

```
In [144]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
.....: index=["Ohio", "Colorado", "Utah", "New York"],
.....: columns=["one", "two", "three", "four"])
```

```
In [145]: data
Out[145]:
```

	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 [146]: data["two"]
Out[146]:
Ohio      1
Colorado   5
Utah       9
New York  13
Name: two, dtype: int64
```

```
In [147]: data[["three", "one"]]
Out[147]:
```

	three	one
Ohio	2	0

Colorado	6	4
Utah	10	8
New York	14	12

У доступа по индексу есть несколько частных случаев. Во-первых, выборка или вырезание данных с помощью булева массива:

```
In [148]: data[:2]
Out[148]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
In [149]: data[data["three"] > 5]
Out[149]:
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Для удобства предоставляется специальный синтаксис выборки строк `data[:2]`. Если передать один элемент или список оператору `[]`, то выбираются столбцы.

Еще одна возможность – доступ по индексу с помощью булева `DataFrame`, например порожденного в результате скалярного сравнения. Рассмотрим `DataFrame`, содержащий только булевы значения, порожденные сравнением со скаляром:

```
In [150]: data < 5
Out[150]:
```

	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

Мы можем использовать этот `DataFrame`, чтобы присвоить значение 0 всем элементам, которым соответствует значение `True`:

```
In [151]: data[data < 5] = 0
In [152]: data
Out[152]:
```

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7

Выборка из `DataFrame` с помощью `loc` и `iloc`

Как и `Series`, объект `DataFrame` имеет специальные атрибуты `loc` и `iloc` для индексирования метками и целыми числами соответственно. Поскольку объект `DataFrame` двумерный, мы можем выбрать подмножество строк и столбцов `DataFrame` с применением нотации NumPy, используя либо метки строк (`loc`), либо целые числа (`iloc`).

В качестве вступительного примера выберем одну строку по метке:

```
In [153]: data
Out[153]:
```

	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

```
In [154]: data.loc["Colorado"]
Out[154]:
```

one	0
two	5
three	6
four	7

Name: Colorado, dtype: int64

Результатом выбора одной строки является объект Series с индексом, содержащим метки столбцов DataFrame. Чтобы выбрать несколько строк и тем самым создать новый объект DataFrame, передадим последовательность меток:

```
In [155]: data.loc[["Colorado", "New York"]]
Out[155]:
```

	one	two	three	four
Colorado	0	5	6	7
New York	12	13	14	15

Оператор `loc` позволяет выбрать одновременно строки и столбцы – нужно только разделить указания тех и других запятой:

```
In [156]: data.loc["Colorado", ["two", "three"]]
Out[156]:
```

two	5
three	6

Name: Colorado, dtype: int64

Затем произведем аналогичную выборку, но уже по целочисленным индексам с помощью `iloc`:

```
In [157]: data.iloc[2]
Out[157]:
```

one	8
two	9
three	10
four	11

Name: Utah, dtype: int64

```
In [158]: data.iloc[[2, 1]]
Out[158]:
```

	one	two	three	four
Utah	8	9	10	11
Colorado	0	5	6	7

```
In [159]: data.iloc[2, [3, 0, 1]]
Out[159]:
```

four	11
one	8
two	9

```
Name: Utah, dtype: int64
```

```
In [160]: data.iloc[[1, 2], [3, 0, 1]]
```

```
Out[160]:
```

```
         four one two
Colorado    7   0   5
Utah       11   8   9
```

Обе функции индексирования работают не только с одиночными метками или списками меток, но и со срезами:

```
In [161]: data.loc["Utah", "two"]
```

```
Out[161]:
```

```
Ohio      0
Colorado   5
Utah      9
Name: two, dtype: int64
```

```
In [162]: data.iloc[:, :3][data.three > 5]
```

```
Out[162]:
```

```
         one two three
Colorado    0   5    6
Utah        8   9   10
New York   12  13   14
```

Таким образом, существует много способов выборки и реорганизации данных, содержащихся в объекте `pandas`. Для `DataFrame` краткая сводка многих из них приведена в табл. 5.4. Позже мы увидим, что при работе с иерархическими индексами есть ряд дополнительных возможностей.

Таблица 5.4. Варианты доступа по индексу для объекта `DataFrame`

Вариант	Примечание
<code>df[column]</code>	Выбрать один столбец или последовательность столбцов из <code>DataFrame</code> . Частные случаи: булев массив (фильтрация строк), срез (вырезание строк) или булев <code>DataFrame</code> (установка значений в позициях, удовлетворяющих некоторому критерию)
<code>df.loc[rows]</code>	Выбрать одну строку или подмножество строк из <code>DataFrame</code> по метке
<code>obj.loc[:, cols]</code>	Выбрать один столбец или подмножество столбцов по метке
<code>df.iloc[rows, cols]</code>	Выбрать строки и столбцы по метке
<code>df.iloc[rows]</code>	Выбрать одну строку или подмножество строк из <code>DataFrame</code> по целочисленной позиции
<code>obj.iloc[:, cols]</code>	Выбрать один столбец или подмножество столбцов по целочисленной позиции
<code>df.loc[rows, cols]</code>	Выбрать строки и столбцы по целочисленной позиции
<code>df.at[row, col]</code>	Выбрать одно скалярное значение по меткам строки и столбца
<code>df.iat[row, col]</code>	Выбрать одно скалярное значение по целочисленным позициям строки и столбца
метод <code>reindex</code>	Выбрать строки или столбцы по меткам

Подвохи целочисленного индексирования

Начинающие часто испытывают затруднения при работе с объектами pandas, индексированными целыми числами, из-за различий с семантикой индексирования встроенных в Python структур данных, таких как списки и кортежи. Например, вряд ли вы ожидаете столкнуться с ошибкой в следующем коде:

```
In [164]: ser = pd.Series(np.arange(3.))

In [165]: ser
Out[165]:
0    0.0
1    1.0
2    2.0
dtype: float64

In [166]: ser[-1]
-----
ValueError                                Traceback (most recent call last)
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/indexes/range.p
y in get_loc(self, key, method, tolerance)
    384         try:
--> 385             return self._range.index(new_key)
    386         except ValueError as err:
ValueError: -1 is not in range
The above exception was the direct cause of the following exception:
KeyError                                Traceback (most recent call last)
<ipython-input-166-44969a759c20> in <module>
----> 1 ser[-1]
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/series.py in __
getitem__(self, key)
    956
    957     elif key_is_scalar:
--> 958         return self._get_value(key)
    959
    960     if is_hashable(key):
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/series.py in _g
et_value(self, label, takeable)
   1067
   1068     # Similar to Index.get_value, but we do not fall back to positional
-> 1069     loc = self.index.get_loc(label)
   1070     return self.index._get_values_for_loc(self, loc, label)
   1071
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/indexes/range.p
y in get_loc(self, key, method, tolerance)
    385         return self._range.index(new_key)
    386         except ValueError as err:
--> 387             raise KeyError(key) from err
    388     self._check_indexing_error(key)
    389     raise KeyError(key)
KeyError: -1
```

В данном примере pandas могла бы «откатиться» к целочисленному индексированию, но в общем случае это трудно сделать, не внося тонкие ошибки в пользовательский код. Здесь мы имеем индекс, содержащий 0, 1, 2, но понять, чего хочет пользователь (индексировать по метке или по позиции), трудно:

```
In [167]: ser
Out[167]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

С другой стороны, если индекс не целочисленный, то никакой неоднозначности не возникает:

```
In [168]: ser2 = pd.Series(np.arange(3.), index=["a", "b", "c"])

In [169]: ser2[-1]
Out[169]: 2.0
```

Если индекс по какой-то оси содержит целые числа, то выборка данных всегда производится по метке. Как было сказано выше, использование `loc` (для меток) или `iloc` (для целых) всегда приводит к желаемому результату:

```
In [170]: ser.iloc[-1]
Out[170]: 2.0
```

С другой стороны, срез с применением целых чисел всегда ориентирован на целые:

```
In [171]: ser[:2]
Out[171]:
0    0.0
1    1.0
dtype: float64
```

Из-за всех этих тонкостей рекомендуется всегда использовать индексирование с помощью `loc` и `iloc`, избежав тем самым двусмысленности.

Подвохи цепного индексирования

В предыдущем разделе мы видели, как можно гибко выбирать данные из объекта `DataFrame` с помощью атрибутов `loc` и `iloc`. Их можно использовать также для изменения `DataFrame` на месте, но делать это следует осторожно.

Например, в предыдущем примере `DataFrame` мы можем присвоить значение строке или столбцу по метке или по целому числу:

```
In [172]: data.loc[:, "one"] = 1
```

```
In [173]: data
Out[173]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	1	5	6	7
Utah	1	9	10	11
New York	1	13	14	15

```
In [174]: data.iloc[2] = 5
```

```
In [175]: data
Out[175]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	1	5	6	7
Utah	1	9	10	11
New York	1	13	14	15

Ohio	1	0	0	0
Colorado	1	5	6	7
Utah	5	5	5	5
New York	1	13	14	15

```
In [176]: data.loc[data["four"] > 5] = 3
```

```
In [177]: data
```

```
Out[177]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	5	5
New York	3	3	3	3

Типичная ошибка начинающих пользователей pandas – сцеплять операции выборки при присваивании, например:

```
In [177]: data.loc[data.three == 5]["three"] = 6
<ipython-input-11-0ed1cf2155d5>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

В зависимости от данных может быть напечатано специальное предупреждение `SettingWithCopyWarning`, в котором говорится, что вы пытаетесь модифицировать временное значение (непустой результат `data.loc[data.three == 5]`), а не данные `data` исходного объекта `DataFrame`, как, вероятно, намеревались. В данном случае `data` не изменилось:

```
In [179]: data
```

```
Out[179]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	5	5
New York	3	3	3	3

В таких случаях нужно переписать цепное присваивание, заменив его одной операцией `loc`:

```
In [180]: data.loc[data.three == 5, "three"] = 6
```

```
In [181]: data
```

```
Out[181]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	6	5
New York	3	3	3	3

Рекомендуется вообще избегать цепного индексирования при выполнении присваиваний. Есть и другие ситуации, связанные с цепным индексированием, когда pandas выдает предупреждение `SettingWithCopyWarning`. Отсылаю вас к онлайн-документации.

Арифметические операции и выравнивание данных

pandas может сильно упростить работу с объектами, имеющими различные индексы. Так, если при сложении двух объектов в одном индексе обнаруживается значение, отсутствующее в другом, то результирующий индекс будет объединением исходных. Рассмотрим пример:

```
In [182]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=["a", "c", "d", "e"])
```

```
In [183]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
.....: index=["a", "c", "e", "f", "g"])
```

```
In [184]: s1
```

```
Out[184]:
```

```
a    7.3
```

```
c   -2.5
```

```
d    3.4
```

```
e    1.5
```

```
dtype: float64
```

```
In [185]: s2
```

```
Out[185]:
```

```
a   -2.1
```

```
c    3.6
```

```
e   -1.5
```

```
f    4.0
```

```
g    3.1
```

```
dtype: float64
```

Сложение этих объектов дает:

```
In [186]: s1 + s2
```

```
Out[186]:
```

```
a    5.2
```

```
c    1.1
```

```
d   NaN
```

```
e    0.0
```

```
f   NaN
```

```
g   NaN
```

```
dtype: float64
```

Вследствие внутреннего выравнивания данных образуются значения NaN в позициях, для которых не нашлось соответственной пары. Эти значения распространяются на последующие операции.

В случае DataFrame выравнивание производится как для строк, так и для столбцов:

```
In [187]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),
.....: index=["Ohio", "Texas", "Colorado"])
```

```
In [188]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
.....: index=["Utah", "Ohio", "Texas", "Oregon"])
```

```
In [189]: df1
```

```
Out[189]:
```

```
      b    c    d
Ohio  0.0  1.0  2.0
```

```
Texas    3.0  4.0  5.0
Colorado 6.0  7.0  8.0
```

```
In [190]: df2
Out[190]:
```

```
      b    d    e
Utah   0.0  1.0  2.0
Ohio   3.0  4.0  5.0
Texas   6.0  7.0  8.0
Oregon  9.0 10.0 11.0
```

При сложении этих объектов получается DataFrame, индекс и столбцы которого являются объединениями индексов и столбцов слагаемых:

```
In [191]: df1 + df2
Out[191]:
```

```
      b    c    d    e
Colorado NaN NaN NaN NaN
Ohio     3.0 NaN  6.0 NaN
Oregon   NaN NaN NaN NaN
Texas    9.0 NaN 12.0 NaN
Utah     NaN NaN NaN NaN
```

Поскольку столбцы 'c' и 'e' не встречаются в обоих объектах DataFrame сразу, то в результирующем объекте они помечены как отсутствующие. То же самое относится к строкам с метками, не встречающимися в обоих объектах.

Если сложить объекты DataFrame, не имеющие общих столбцов и строк, то результат будет содержать только значения NaN.

```
In [192]: df1 = pd.DataFrame({"A": [1, 2]})
```

```
In [193]: df2 = pd.DataFrame({"B": [3, 4]})
```

```
In [194]: df1
Out[194]:
```

```
  A
0  1
1  2
```

```
In [195]: df2
Out[195]:
```

```
  B
0  3
1  4
```

```
In [196]: df1 + df2
Out[196]:
```

```
   A  B
0 NaN NaN
1 NaN NaN
```

Восполнение значений в арифметических методах

При выполнении арифметических операций с объектами, проиндексированными по-разному, иногда желательно поместить специальное значение, например 0, в позиции операнда, которым в другом операнде соответствует от-

сутствующая позиция. В примере ниже мы записываем в элемент специальный признак отсутствия, присваивая ему значение `np.nan`:

```
In [197]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
.....: columns=list("abcd"))
```

```
In [198]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
.....: columns=list("abcde"))
```

```
In [199]: df2.loc[1, "b"] = np.nan
```

```
In [200]: df1
```

```
Out[200]:
```

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

```
In [201]: df2
```

```
Out[201]:
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Сложение этих объектов порождает отсутствующие значения в позициях, которые имеются не в обоих операндах:

```
In [202]: df1 + df2
```

```
Out[202]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

Теперь я вызову метод `add` объекта `df1` и передам ему объект `df2` и значение параметра `fill_value`. В результате все отсутствующие значения будут заменены переданным:

```
In [203]: df1.add(df2, fill_value=0)
```

```
Out[203]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

В табл. 5.5 приведен перечень арифметических методов объектов `Series` и `DataFrame`. У каждого из них имеется вариант, начинающийся буквой `r`, в котором аргументы переставлены местами. Поэтому следующие два предложения эквивалентны:

```
In [204]: 1 / df1
```

```
Out[204]:
```

	a	b	c	d

```
0    inf 1.000000 0.500000 0.333333
1  0.250 0.200000 0.166667 0.142857
2  0.125 0.111111 0.100000 0.090909
```

```
In [205]: df1.rdiv(1)
Out[205]:
      a      b      c      d
0    inf 1.000000 0.500000 0.333333
1  0.250 0.200000 0.166667 0.142857
2  0.125 0.111111 0.100000 0.090909
```

Точно так же, выполняя переиндексацию объекта Series или DataFrame, можно указать восполняемое значение:

```
In [206]: df1.reindex(columns=df2.columns, fill_value=0)
Out[206]:
      a      b      c      d      e
0  0.0  1.0  2.0  3.0  0
1  4.0  5.0  6.0  7.0  0
2  8.0  9.0 10.0 11.0  0
```

Таблица 5.5. Гибкие арифметические методы

Метод	Описание
<code>add, radd</code>	Сложение (+)
<code>sub, rsub</code>	Вычитание (-)
<code>div, rdiv</code>	Деление (/)
<code>floordiv, rfloordiv</code>	Деление с отсечением (//)
<code>mul, rmul</code>	Умножение (*)
<code>pow, grow</code>	Возведение в степень (**)

Операции между DataFrame и Series

Как и в случае массивов NumPy, арифметические операции между DataFrame и Series корректно определены. В качестве пояснительного примера рассмотрим вычисление разности между двумерным массивом и одной из его строк:

```
In [207]: arr = np.arange(12.).reshape((3, 4))
```

```
In [208]: arr
Out[208]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
In [209]: arr[0]
Out[209]: array([0., 1., 2., 3.])
```

```
In [210]: arr - arr[0]
Out[210]:
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

Когда мы вычитаем `arr[0]` из `arr`, операция производится один раз для каждой строки. Это называется укладыванием и подробно объясняется в приложении А, поскольку имеет отношение к массивам NumPy вообще. Операции между DataFrame и Series аналогичны:

```
In [211]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
.....:                        columns=list("bde"),
.....:                        index=["Utah", "Ohio", "Texas", "Oregon"])
```

```
In [212]: series = frame.iloc[0]
```

```
In [213]: frame
```

```
Out[213]:
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [214]: series
```

```
Out[214]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

По умолчанию при выполнении арифметических операций между DataFrame и Series индекс Series сопоставляется со столбцами DataFrame, а укладываются строки:

```
In [215]: frame - series
```

```
Out[215]:
```

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

Если какой-нибудь индекс не найден либо в столбцах DataFrame, либо в индексе Series, то объекты переиндексируются для образования объединения:

```
In [216]: series2 = pd.Series(np.arange(3), index=["b", "e", "f"])
```

```
In [217]: series2
```

```
Out[217]:
b    0
e    1
f    2
dtype: int64
```

```
In [218]: frame + series2
```

```
Out[218]:
```

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

Если вы хотите вместо этого сопоставлять строки, а укладывать столбцы, то должны будете воспользоваться каким-нибудь арифметическим методом, указав сопоставление по индексу. Например:

```
In [219]: series3 = frame["d"]

In [220]: frame
Out[220]:
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```

In [221]: series3
Out[221]:
Utah      1.0
Ohio      4.0
Texas     7.0
Oregon    10.0
Name: d, dtype: float64

In [222]: frame.sub(series3, axis="index")
Out[222]:
```

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

Передаваемый номер оси – это ось, *вдоль которой производится сопоставление*. В данном случае мы хотим сопоставлять с индексом строк DataFrame (`axis="index"`) и укладывать поперек.

Применение функций и отображение

Универсальные функции NumPy (поэлементные методы массивов) отлично работают и с объектами pandas:

```
In [223]: frame = pd.DataFrame(np.random.standard_normal((4, 3)),
.....:                        columns=list("bde"),
.....:                        index=["Utah", "Ohio", "Texas", "Oregon"])

In [224]: frame
Out[224]:
```

	b	d	e
Utah	-0.204708	0.478943	-0.519439
Ohio	-0.555730	1.965781	1.393406
Texas	0.092908	0.281746	0.769023
Oregon	1.246435	1.007189	-1.296221

```

In [225]: np.abs(frame)
Out[225]:
```

	b	d	e
Utah	0.204708	0.478943	0.519439
Ohio	0.555730	1.965781	1.393406

```
Texas    0.092908  0.281746  0.769023
Oregon   1.246435  1.007189  1.296221
```

Еще одна часто встречающаяся операция – применение функции, определенной для одномерных массивов, к каждому столбцу или строке. Именно это и делает метод `apply` объекта `DataFrame`:

```
In [226]: def f1(x):
.....:     return x.max() - x.min()

In [227]: frame.apply(f1)
Out[227]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

Здесь функция `f`, вычисляющая разность между максимальным и минимальным значениями `Series`, вызывается один раз для каждого столбца `frame`. В результате получается объект `Series`, для которого индексом являются столбцы `frame`.

Если передать методу `apply` аргумент `axis="columns"`, то функция будет вызываться по одному разу для каждой строки:

```
In [228]: frame.apply(f1, axis="columns")
Out[228]:
Utah    0.998382
Ohio    2.521511
Texas   0.676115
Oregon  2.542656
dtype: float64
```

Многие из наиболее распространенных статистик массивов (например, `sum` и `mean`) – методы `DataFrame`, поэтому применять `apply` в этом случае необязательно.

Функция, передаваемая методу `apply`, не обязана возвращать скалярное значение, она может вернуть и объект `Series`, содержащий несколько значений:

```
In [229]: def f2(x):
.....:     return pd.Series([x.min(), x.max()], index=["min", "max"])

In [230]: frame.apply(f2)
Out[230]:
      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 [231]: def my_format(x):
.....:     return f"{x:.2f}"

In [232]: frame.applymap(my_format)
Out[232]:
```


	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 [233]: frame["e"].map(my_format)
Out[233]:
Utah      -0.52
Ohio       1.39
Texas       0.77
Oregon    -1.30
Name: e, dtype: object
```

Сортировка и ранжирование

Сортировка набора данных по некоторому критерию – еще одна важная встроенная операция. Для лексикографической сортировки по индексу служит метод `sort_index`, который возвращает новый отсортированный объект:

```
In [234]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])

In [235]: obj
Out[235]:
d    0
a    1
b    2
c    3
dtype: int64

In [236]: obj.sort_index()
Out[236]:
a    1
b    2
c    3
d    0
dtype: int64
```

В случае `DataFrame` можно сортировать по индексу, ассоциированному с любой осью:

```
In [237]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
.....:                        index=["three", "one"],
.....:                        columns=["d", "a", "b", "c"])

In [238]: frame
Out[238]:
   d a b c
three 0 1 2 3
one   4 5 6 7

In [239]: frame.sort_index()
Out[239]:
   d a b c
```

```

one    4 5 6 7
three  0 1 2 3

In [240]: frame.sort_index(axis="columns")
Out[240]:
   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 [241]: frame.sort_index(axis="columns", ascending=False)
Out[241]:
   d c b a
three 0 3 2 1
one    4 7 6 5

```

Для сортировки Series по значениям служит метод `sort_values`:

```

In [242]: obj = pd.Series([4, 7, -3, 2])

In [243]: obj.sort_values()
Out[243]:
2    -3
3     2
0     4
1     7
dtype: int64

```

Значения NaN по умолчанию оказываются в конце Series:

```

In [244]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [245]: obj.sort_values()
Out[245]:
4    -3.0
5     2.0
0     4.0
2     7.0
1    NaN
3    NaN
dtype: float64

```

Значения NaN можно поместить в начало, а не в конец, добавив именованный параметр `na_position`:

```

In [246]: obj.sort_values(na_position="first")
Out[246]:
1    NaN
3    NaN
4    -3.0
5     2.0
0     4.0
2     7.0
dtype: float64

```

Объект DataFrame можно сортировать по значениям в одном или нескольких столбцах. Для этого передайте имя столбца методу `sort_values`:

```
In [247]: frame = pd.DataFrame({"b": [4, 7, -3, 2], "a": [0, 1, 0, 1]})
```

```
In [248]: frame
```

```
Out[248]:
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1
```

```
In [249]: frame.sort_values("b")
```

```
Out[249]:
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

Для сортировки по нескольким столбцам следует передать список имен:

```
In [250]: frame.sort_values(["a", "b"])
```

```
Out[250]:
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

Ранжирование заключается в присваивании рангов – от единицы до числа присутствующих в массиве элементов, начиная с наименьшего значения. Для ранжирования применяется метод `rank` объектов `Series` и `DataFrame`; по умолчанию `rank` обрабатывает равные ранги, присваивая каждой группе средний ранг:

```
In [251]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [252]: obj.rank()
```

```
Out[252]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ранги можно также присваивать в соответствии с порядком появления в данных:

```
In [253]: obj.rank(method="first")
```

```
Out[253]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Здесь элементам 0 и 2 присвоен не средний ранг 6.5, а ранги 6 и 7, потому что в данных метка 0 предшествует метке 2.

Можно ранжировать и в порядке убывания:

```
In [254]: obj.rank(ascending=False)
Out[254]:
0    1.5
1    7.0
2    1.5
3    3.5
4    5.0
5    6.0
6    3.5
dtype: float64
```

В табл. 5.6 перечислены способы обработки равных рангов.

DataFrame умеет вычислять ранги как по строкам, так и по столбцам:

```
In [255]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],
.....: "c": [-2, 5, 8, -2.5]})
```

```
In [256]: frame
Out[256]:
   b  a  c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5
```

```
In [257]: frame.rank(axis="columns")
Out[257]:
   b  a  c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

Таблица 5.6. Способы обработки равных рангов

Способ	Описание
"average"	По умолчанию: всем элементам группы присвоить средний ранг
"min"	Всем элементам группы присвоить минимальный ранг
"max"	Всем элементам группы присвоить максимальный ранг
"first"	Присваивать ранги в порядке появления значений в наборе данных
"dense"	Как <code>method="min"</code> , но при переходе к следующей группе элементов с одинаковым рангом ранг всегда увеличивается на 1, а не на количество элементов в группе

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`) требу-

ется уникальность меток, в общем случае это необязательно. Рассмотрим небольшой объект Series с повторяющимися индексами:

```
In [258]: obj = pd.Series(np.arange(5), index=["a", "a", "b", "b", "c"])

In [259]: obj
Out[259]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

О том, являются метки уникальными или нет, можно узнать, опросив свойство `is_unique`:

```
In [260]: obj.index.is_unique
Out[260]: False
```

Выборка данных – одна из основных операций, поведение которых меняется в зависимости от наличия или отсутствия дубликатов. Если метка встречается несколько раз, то возвращается объект Series, а если только один раз, то скалярное значение:

```
In [261]: obj["a"]
Out[261]:
a    0
a    1
dtype: int64

In [262]: obj["c"]
Out[262]: 4
```

Это иногда усложняет код, поскольку тип, получающийся в результате индексирования, может зависеть от того, повторяется метка или нет.

Такое же правило действует для доступа к строкам (или столбцам) в DataFrame:

```
In [263]: df = pd.DataFrame(np.random.standard_normal((5, 3)),
.....:                      index=["a", "a", "b", "b", "c"])

In [264]: df
Out[264]:
```

	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
c	-0.577087	0.124121	0.302614

```

In [265]: df.loc["b"]
Out[265]:
```

	0	1	2
b	1.669025	-0.438570	-0.539741
b	0.476985	3.248944	-1.021228

```
In [266]: df.loc["c"]
Out[266]:
0    -0.577087
1     0.124121
2     0.302614
Name: c, dtype: float64
```

5.3. Редукция и вычисление описательных статистик

Объекты `pandas` оснащены набором стандартных математических и статистических методов. Большая их часть попадает в категорию *редукций*, или *сводных статистик*, – методов, которые вычисляют единственное значение (например, сумму или среднее) для `Series` или объект `Series` – для строк либо столбцов `DataFrame`. По сравнению с эквивалентными методами массивов `NumPy`, они имеют встроенные средства обработки отсутствующих данных. Рассмотрим небольшой объект `DataFrame`:

```
In [267]: df = pd.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 [268]: df
Out[268]:
   one  two
a  1.40 NaN
b  7.10 -4.5
c  NaN  NaN
d  0.75 -1.3
```

Метод `sum` объекта `DataFrame` возвращает объект `Series`, содержащий суммы по столбцам:

```
In [269]: df.sum()
Out[269]:
one    9.25
two   -5.80
dtype: float64
```

Если передать параметр `axis="columns"` или `axis=1`, то суммирование будет производиться по строкам:

```
In [270]: df.sum(axis="columns")
Out[270]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

Если строка или столбец целиком состоит из значений `NaN`, то сумма равна 0. В противном случае значения `NaN` пропускаются при вычислении суммы. Этот режим можно изменить, задав параметр `skipna=False`, тогда при наличии хотя бы одного значения `NaN` в строке или в столбце сумма будет равна `NaN`:

```
In [271]: df.sum(axis="index", skipna=False)
Out[271]:
one    NaN
two    NaN
dtype: float64

In [272]: df.sum(axis="columns", skipna=False)
Out[272]:
a      NaN
b     2.60
c      NaN
d    -0.55
dtype: float64
```

Некоторым агрегатным функциям, например `mean`, для получения результата требуется, чтобы по крайней мере одно значение было отлично от NaN, поэтому в данном случае мы имеем:

```
In [273]: df.mean(axis="columns")
Out[273]:
a     1.400
b     1.300
c      NaN
d    -0.275
dtype: float64
```

Перечень часто употребляемых параметров методов редукции приведен в табл. 5.7.

Таблица 5.7. Параметры методов редукции

Метод	Описание
<code>axis</code>	Ось, по которой производится редуцирование. В случае DataFrame " <code>index</code> " означает строки, " <code>columns</code> " – 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 (MultiIndex)

Некоторые методы, например `idxmin` и `idxmax`, возвращают косвенные статистики, скажем индекс, при котором достигается минимум или максимум:

```
In [274]: df.idxmax()
Out[274]:
one    b
two    d
dtype: object
```

Существуют также *аккумулирующие* методы:

```
In [275]: df.cumsum()
Out[275]:
      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 аккумулярующим. Примером может служить метод `describe`, который возвращает несколько сводных статистик за одно обращение:

```
In [276]: df.describe()
Out[276]:
```

	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 [277]: obj = pd.Series(["a", "a", "b", "c"] * 4)

In [278]: obj.describe()
Out[278]:
```

count	16
unique	3
top	a
freq	8

dtype: object

Полный список сводных статистик и родственных методов приведен в табл. 5.8.

Таблица 5.8. Описательные и сводные статистики

Метод	Описание
<code>count</code>	Количество значений, исключая отсутствующие
<code>describe</code>	Вычисляет набор сводных статистик
<code>min</code> , <code>max</code>	Вычисляет минимальное или максимальное значение
<code>argmin</code> , <code>argmax</code>	Вычисляет позиции в индексе (целые числа), в которых достигается минимальное или максимальное значение соответственно
<code>idxmin</code> , <code>idxmax</code>	Вычисляет метки индекса, для которых достигается минимальное или максимальное значение соответственно
<code>quantile</code>	Вычисляет выборочный квантиль в диапазоне от 0 до 1 (по умолчанию 0.5)
<code>sum</code>	Сумма значений
<code>mean</code>	Среднее значение
<code>median</code>	Медиана (50%-ный квантиль)
<code>mad</code>	Среднее абсолютное отклонение от среднего

Метод	Описание
<code>prod</code>	Произведение значений
<code>var</code>	Выборочная дисперсия
<code>std</code>	Выборочное стандартное отклонение
<code>skew</code>	Асимметрия (третий момент)
<code>kurt</code>	Куртозис (четвертый момент)
<code>cumsum</code>	Нарастающая сумма
<code>cummin</code> , <code>cummax</code>	Нарастающий минимум или максимум соответственно
<code>cumprod</code>	Нарастающее произведение
<code>diff</code>	Первая арифметическая разность (полезно для временных рядов)
<code>pct_change</code>	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. Эти данные имеются в pickle-файлах, размещенных на сайте книги:

```
In [279]: price = pd.read_pickle("examples/yahoo_price.pkl")
In [280]: volume = pd.read_pickle("examples/yahoo_volume.pkl")
```

Теперь вычислим процентное изменение цен (эта операция над временными рядами рассматривается подробнее в главе 11):

```
In [281]: returns = price.pct_change()

In [282]: returns.tail()
Out[282]:
```

	AAPL	GOOG	IBM	MSFT
Date				
2016-10-17	-0.000680	0.001837	0.002072	-0.003483
2016-10-18	-0.000681	0.019616	-0.026168	0.007690
2016-10-19	-0.002979	0.007846	0.003583	-0.002255
2016-10-20	-0.000512	-0.005652	0.001719	-0.004867
2016-10-21	-0.003930	0.003011	-0.012474	0.042096

Метод `corr` объекта `Series` вычисляет корреляцию перекрывающихся, отличных от NaN, выровненных по индексу значений в двух объектах `Series`. Соответственно, метод `cov` вычисляет ковариацию:

```
In [283]: returns["MSFT"].corr(returns["IBM"])
Out[283]: 0.49976361144151144

In [284]: returns["MSFT"].cov(returns["IBM"])
Out[284]: 8.870655479703546e-05
```

Поскольку `MSFT` – допустимое имя атрибута Python, те же столбцы можно выбрать более лаконично:

```
In [285]: returns["MSFT"].corr(returns["IBM"])
Out[285]: 0.49976361144151144
```

С другой стороны, методы `corr` и `cov` объекта `DataFrame` возвращают соответственно полную корреляционную или ковариационную матрицу в виде `DataFrame`:

```
In [286]: returns.corr()
Out[286]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.407919	0.386817	0.389695
GOOG	0.407919	1.000000	0.405099	0.465919
IBM	0.386817	0.405099	1.000000	0.499764
MSFT	0.389695	0.465919	0.499764	1.000000

```
In [287]: returns.cov()
Out[287]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.000277	0.000107	0.000078	0.000095
GOOG	0.000107	0.000251	0.000078	0.000108
IBM	0.000078	0.000078	0.000146	0.000089
MSFT	0.000095	0.000108	0.000089	0.000215

С помощью метода `corrwith` объекта `DataFrame` можно вычислить попарные корреляции между столбцами или строками `DataFrame` и другим объектом `Series` или `DataFrame`. Если передать ему объект `Series`, то будет возвращен `Series`, содержащий значения корреляции, вычисленные для каждого столбца:

```
In [288]: returns.corrwith(returns["IBM"])
Out[288]:
```

AAPL	0.386817
GOOG	0.405099
IBM	1.000000
MSFT	0.499764

dtype: float64

Если передать объект `DataFrame`, то будут вычислены корреляции столбцов с соответственными именами. Ниже я вычисляю корреляцию процентных изменений с объемом сделок:

```
In [289]: returns.corrwith(volume)
Out[289]:
```

AAPL	-0.075565
GOOG	-0.007067
IBM	-0.204849
MSFT	-0.092950

dtype: float64

Если передать `axis="columns"`, то будут вычислены корреляции строк. Во всех случаях перед началом вычислений данные выравниваются по меткам.

Уникальные значения, счетчики значений и членство

Еще один класс методов служит для извлечения информации о значениях, хранящихся в одномерном объекте `Series`. Для иллюстрации рассмотрим пример:

```
In [290]: obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
```

Метод `unique` возвращает массив уникальных значений в Series:

```
In [291]: uniques = obj.unique()
```

```
In [292]: uniques
```

```
Out[292]: 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 [293]: obj.value_counts()
```

```
Out[293]:
```

```
c    3
a    3
b    2
d    1
dtype: int64
```

Для удобства этот объект отсортирован по значениям в порядке убывания. Функция `value_counts` может быть также вызвана как метод pandas верхнего уровня и в таком случае применима к массивам NumPy и другим последовательностям Python:

```
In [294]: pd.value_counts(obj.to_numpy(), sort=False)
```

```
Out[294]:
```

```
c    3
a    3
d    1
b    2
dtype: int64
```

Метод `isin` вычисляет булев вектор членства в множестве и может быть полезен для фильтрации набора данных относительно подмножества значений в объекте Series или столбце DataFrame:

```
In [295]: obj
```

```
Out[295]:
```

```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

```
In [296]: mask = obj.isin(["b", "c"])
```

```
In [297]: mask
```

```
Out[297]:
```

```
0    True
1    False
```

```

2 False
3 False
4 False
5 True
6 True
7 True
8 True
dtype: bool

In [298]: obj[mask]
Out[298]:
0 c
5 b
6 b
7 c
8 c
dtype: object

```

C `isin` тесно связан метод `Index.get_indexer`, который возвращает массив индексов, описывающий соответствие между массивом потенциально повторяющихся значений и массивом, содержащим только различные значения:

```

In [299]: to_match = pd.Series(["c", "a", "b", "b", "c", "a"])

In [300]: unique_vals = pd.Series(["c", "b", "a"])

In [301]: indices = pd.Index(unique_vals).get_indexer(to_match)

In [302]: indices
Out[302]: array([0, 2, 1, 1, 0, 2])

```

Справочная информация по этим методам приведена в табл. 5.9.

Таблица 5.9. Уникальные значения, счетчики значений и методы «раскладывания»

Метод	Описание
<code>isin</code>	Вычисляет булев массив, показывающий, содержится ли каждое принадлежащее Series или DataFrame значение в переданной последовательности
<code>get_indexer</code>	Вычисляет для каждого значения массива целочисленный индекс в другом массиве неповторяющихся значений; полезен, когда нужно выровнять данные и выполнить операции, подобные соединению
<code>unique</code>	Вычисляет массив уникальных значений в Series и возвращает их в порядке появления
<code>value_counts</code>	Возвращает объект Series, который содержит уникальное значение в качестве индекса и его частоту в качестве значения. Результат отсортирован в порядке убывания частот

Иногда требуется вычислить гистограмму нескольких взаимосвязанных столбцов в DataFrame. Приведем пример:

```

In [303]: data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],
.....:                       "Qu2": [2, 3, 1, 2, 3],
.....:                       "Qu3": [1, 5, 2, 4, 4]})

```

```
In [304]: data
Out[304]:
   Qu1 Qu2 Qu3
0    1   2   1
1    3   3   5
2    4   1   2
3    3   2   4
4    4   3   4
```

Счетчики значений в одном столбце вычисляются следующим образом:

```
In [305]: data["Qu1"].value_counts().sort_index()
Out[305]:
1    1
3    2
4    2
Name: Qu1, dtype: int64
```

А для вычисления счетчиков по всем столбцам нужно передать `pandas.value_counts` методу `apply` объекта `DataFrame`:

```
In [306]: result = data.apply(pd.value_counts).fillna(0)

In [307]: result
Out[307]:
   Qu1 Qu2 Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

Здесь метками строк результирующего объекта являются неповторяющиеся значения, встречающиеся в столбцах. Значениями же являются счетчики значений в столбце.

Существует также метод `DataFrame.value_counts`, но он при вычислении счетчиков рассматривает каждую строку `DataFrame` как кортеж, чтобы определить количество вхождений различных строк:

```
In [308]: data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})

In [309]: data
Out[309]:
   a  b
0  1  0
1  1  0
2  1  1
3  2  0
4  2  0

In [310]: data.value_counts()
Out[310]:
a  b
1  0    2
2  0    2
1  1    1
dtype: int64
```

В этом случае разные строки представлены в результате иерархическим индексом; эту тему мы будем подробнее обсуждать в главе 8.

5.4. ЗАКЛЮЧЕНИЕ

В следующей главе мы обсудим имеющиеся в `pandas` средства чтения (или *загрузки*) и записи наборов данных. Затем займемся вопросами очистки, трансформации, анализа и визуализации данных.

Чтение и запись данных, форматы файлов

Чтение данных и обеспечение их доступности программе (этот процесс часто называют *загрузкой данных*) – обязательный первый шаг применения большинства описанных в книге инструментов. Иногда для описания загрузки текстовых данных и их интерпретации как таблиц и других типов данных употребляется термин *разбор* (parsing). Я буду рассматривать в основном ввод и вывод с помощью объектов pandas, хотя, разумеется, и в других библиотеках нет недостатка в соответствующих средствах.

Обычно средства ввода-вывода относят к нескольким категориям: чтение файлов в текстовом или каком-то более эффективном двоичном формате, загрузка из баз данных и взаимодействие с сетевыми источниками, например API доступа к вебу.

6.1. ЧТЕНИЕ И ЗАПИСЬ ДАННЫХ В ТЕКСТОВОМ ФОРМАТЕ

В библиотеке pandas имеется ряд функций для чтения табличных данных, представленных в виде объекта DataFrame. Некоторые из них перечислены в табл. 6.1, хотя в этой книге чаще всего используется функция `read_csv`. Двоичные форматы данных рассматриваются в разделе 6.2.

Таблица 6.1. Функции загрузки текстовых и двоичных данных в pandas

Функция	Описание
<code>read_csv</code>	Загружает данные из файла, URL-адреса или файлоподобного объекта. По умолчанию разделителем является запятая
<code>read_fwf</code>	Читает данные в формате с фиксированной шириной столбцов (без разделителей)
<code>read_clipboard</code>	Вариант <code>read_csv</code> , который читает данные из буфера обмена. Полезно для преобразования в таблицу данных на веб-странице
<code>read_excel</code>	Читает табличные данные из файлов Excel в формате XLS или XLSX
<code>read_hdf</code>	Читает HDF5-файлы, записанные pandas
<code>read_html</code>	Читает все таблицы, обнаруженные в HTML-документе

Окончание табл. 6.1

Функция	Описание
<code>read_json</code>	Читает данные из строки в формате JSON (JavaScript Object Notation)
<code>read_feather</code>	Читает данные в двоичном формате Feather
<code>read_orc</code>	Читает данные в двоичном формате Apache ORC
<code>read_parquet</code>	Читает данные в двоичном формате Apache Parquet
<code>read_pickle</code>	Читает объект, сохраненный pandas в формате Python pickle
<code>read_sas</code>	Читает набор данных в одном из пользовательских форматов хранения системы SAS
<code>read_sql</code>	Читает результаты SQL-запроса (применяя пакет SQLAlchemy) в объект pandas DataFrame
<code>read_spss</code>	Читает данные, созданные системой SPSS
<code>read_sql</code>	Читает результаты выполнения SQL-запроса (с использованием SQLAlchemy)
<code>read_sql_table</code>	Читает SQL-таблицу целиком (с использованием SQLAlchemy); эквивалентно чтению с помощью <code>read_sql</code> результатов запроса, который выбирает из таблицы все данные
<code>read_stata</code>	Читает набор данных из файла в формате Stata
<code>read_xml</code>	Читает табличные данные из XML-файла

Я дам краткий обзор этих функций, которые служат для преобразования текстовых данных в объект DataFrame. Их необязательные параметры можно отнести к нескольким категориям:

Индексирование

Какие столбцы рассматривать как индекс возвращаемого DataFrame и откуда брать имена столбцов: из файла, из заданных вами аргументов или вообще ниоткуда.

Выведение типа и преобразование данных

Включает определенные пользователем преобразования значений и список маркеров отсутствия 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 верхнего или нижнего колонтитула, комментариев и другие мелочи, например обработка числовых данных, в которых тройки разрядов разделены запятыми.

В связи с тем, что в реальности данные могут быть очень грязными, некоторые функции загрузки (в особенности `pandas.read_csv`) со временем обросли гигантским количеством опций (так, у `pandas.read_csv` их уже больше 50). В онлайн-документации есть много примеров их применения, не исключено, что вы найдете подходящий, сражаясь с конкретным файлом.

Некоторые функции производят *выведение типа*, поскольку информация о типах данных в столбцах не входит в формат данных. Это означает, что не обязательно явно указывать, какие столбцы являются числовыми, целыми, булевыми или строками. Есть и такие форматы, например HDF5, ORC и Parquet, в которых сведения о типах данных хранятся явно.

Для обработки дат и нестандартных типов требуется больше усилий.

Начнем с текстового файла, содержащего короткий список данных через запятую (формат CSV):

```
In [10]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```



Здесь я воспользовался командой Unix `cat`, которая печатает содержимое файла на экране без какого-либо форматирования. Если вы работаете в Windows, можете с тем же успехом использовать команду `type` в терминале.

Поскольку данные разделены запятыми, мы можем прочитать их в `DataFrame` с помощью функции `pandas.read_csv`:

```
In [11]: df = pd.read_csv("examples/ex1.csv")

In [12]: df
Out[12]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

В файле не всегда есть строка-заголовок. Рассмотрим такой файл:

```
In [13]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Прочитать его можно двумя способами. Можно поручить `pandas` выбрать имена столбцов по умолчанию, а можно задать их самостоятельно:

```
In [14]: pd.read_csv("examples/ex2.csv", header=None)
Out[14]:
```

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [15]: pd.read_csv("examples/ex2.csv", names=["a", "b", "c", "d", "message"])
Out[15]:
```

	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 [16]: names = ["a", "b", "c", "d", "message"]

In [17]: pd.read_csv("examples/ex2.csv", names=names, index_col="message")
Out[17]:
```

	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 (см. раздел 8.1), то просто передайте список их номеров или имен:

```
In [18]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [19]: parsed = pd.read_csv("examples/csv_mindex.csv",
....:                        index_col=["key1", "key2"])

In [20]: parsed
Out[20]:
```

		value1	value2
one	key2		
	a	1	2
	b	3	4
	c	5	6
two	d	7	8
	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. Рассмотрим такой текстовый файл:

```
In [21]: !cat examples/ex3.txt
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
```

В данном случае поля разделены переменным числом пробелов и, хотя можно было бы переформатировать данные вручную, проще передать функции `pandas.read_csv` регулярное выражение `\s+` в качестве разделителя:

```
In [22]: result = pd.read_csv("examples/ex3.txt", sep="\s+")
```

```
In [23]: result
```

```
Out[23]:
```

	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, `pandas.read_csv` делает вывод, что в данном частном случае первый столбец должен быть индексом DataFrame.

У функций разбора много дополнительных аргументов, которые помогают справиться с широким разнообразием файловых форматов (см. табл. 6.2). Например, параметр `skiprows` позволяет пропустить первую, третью и четвертую строки файла:

```
In [24]: !cat examples/ex4.csv
# привет!
a,b,c,d,message
# хотелось немного усложнить тебе жизнь
# да кто вообще читает CSV-файлы на компьютере?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

```
In [25]: pd.read_csv("examples/ex4.csv", skiprows=[0, 2, 3])
```

```
Out[25]:
```

	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 [26]: !cat examples/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 [27]: result = pd.read_csv("examples/ex5.csv")
```

```
In [28]: result
```

```
Out[28]:
something  a    b    c    d message
0         one  1    2  3.0    4      NaN
1         two  5    6  NaN    8    world
2        three  9   10 11.0   12      foo
```

Напомним, что pandas выводит отсутствующие значения как NaN, так что в result у нас два отсутствующих значения:

```
In [29]: pd.isna(result)
Out[29]:
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 принимает последовательность строк, добавляемых в список строк, которые по умолчанию рассматриваются как маркеры отсутствия значений:

```
In [30]: result = pd.read_csv("examples/ex5.csv", na_values=["NULL"])
```

```
In [31]: result
Out[31]:
something  a    b    c    d message
0         one  1    2   3.0    4      NaN
1         two  5    6  NaN    8    world
2        three  9   10 11.0   12      foo
```

pandas.read_csv поддерживает длинный список представлений отсутствующих значений, но все это по умолчанию можно отключить, задав параметр keep_default_na:

```
In [32]: result2 = pd.read_csv("examples/ex5.csv", keep_default_na=False)
```

```
In [33]: result2
Out[33]:
something  a    b    c    d message
0         one  1    2   3.0    4      NA
1         two  5    6      8    world
2        three  9   10 11.0   12      foo
```

```
In [34]: result2.isna()
Out[34]:
something  a    b    c    d message
0      False False False False False  False
1      False False False False False  False
2      False False False False False  False
```

```
In [35]: result3 = pd.read_csv("examples/ex5.csv", keep_default_na=False,
.....:                          na_values=["NA"])
```

```
In [36]: result3
Out[36]:
something  a    b    c    d message
0         one  1    2    3    4      NaN
1         two  5    6      8    world
```

```
2      three 9 10 11 12      foo

In [37]: result3.isna()
Out[37]:
something      a      b      c      d  message
0      False False False False False      True
1      False False False False False      False
2      False False False False False      False
```

Если в разных столбцах применяются разные маркеры, то их можно задать с помощью словаря:

```
In [38]: sentinels = {"message": ["foo", "NA"], "something": ["two"]}

In [39]: pd.read_csv("examples/ex5.csv", na_values=sentinels,
....:                keep_default_na=False)
Out[39]:
something  a  b  c  d  message
0      one  1  2  3  4      NaN
1      NaN  5  6      8  world
2     three  9 10 11 12      NaN
```

В табл. 6.2 перечислены некоторые часто используемые аргументы функции `pandas.read_csv`.

Таблица 6.2. Некоторые аргументы функции `read_csv`

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 отсутствия данных
<code>keep_default_na</code>	Использовать ли подразумеваемый по умолчанию список маркеров отсутствия значения (по умолчанию <code>True</code>)
<code>comment</code>	Один или несколько символов, начинающих комментарий, который продолжается до конца строки

Аргумент	Описание
<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>
<code>converters</code>	Словарь, содержащий отображение номеров или имен столбцов на функции. Например, <code>{"foo": f}</code> означает, что нужно применить функцию <code>f</code> ко всем значениям в столбце <code>foo</code>
<code>dayfirst</code>	При разборе потенциально неоднозначных дат предполагать международный формат (т. е. 7/6/2012 означает «7 июня 2012»). По умолчанию <code>False</code>
<code>date_parser</code>	Функция, применяемая для разбора дат
<code>nrows</code>	Количество читаемых строк от начала файла
<code>iterator</code>	Возвращает объект <code>TextFileReader</code> для чтения файла порциями
<code>chunksize</code>	Размер порции при итерировании
<code>skip_footer</code>	Сколько строк в конце файла игнорировать
<code>verbose</code>	Печатать разного рода информацию о ходе разбора, например количество отсутствующих значений, помещенных в нечисловые столбцы
<code>encoding</code>	Кодировка текста в случае Unicode. Например, <code>"utf-8"</code> означает, что текст представлен в кодировке UTF-8. Если <code>None</code> , то по умолчанию подразумевается <code>"utf-8"</code>
<code>squeeze</code>	Если в результате разбора данных оказалось, что имеется только один столбец, вернуть объект <code>Series</code>
<code>thousands</code>	Разделитель тысяч, например <code>","</code> или <code>"."</code> . По умолчанию <code>None</code>
<code>decimal</code>	Десятичный разделитель, например <code>"."</code> или <code>","</code> . По умолчанию <code>"."</code>
<code>engine</code>	Какой движок разбора CSV и преобразования использовать. Может принимать значения <code>"c"</code> , <code>"python"</code> или <code>"pyarrow"</code> . По умолчанию подразумевается <code>"c"</code> , хотя более современный движок <code>"pyarrow"</code> разбирает некоторые файлы гораздо быстрее. Движок <code>"python"</code> медленнее, но поддерживает ряд возможностей, отсутствующих у других движков

Чтение текстовых файлов порциями

Для обработки очень больших файлов или для того чтобы определить правильный набор аргументов, необходимых для обработки большого файла, иногда требуется прочитать небольшой фрагмент файла или последовательно читать файл небольшими порциями.

Прежде чем приступить к обработке большого файла, попросим pandas отображать меньше данных:

```
In [40]: pd.options.display.max_rows = 10
```

Теперь имеем:

```
In [41]: result = pd.read_csv("examples/ex6.csv")
```

```
In [42]: result
```

```
Out[42]:
```

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q
...	...	...	...	...	..
9995	2.311896	-0.417070	-1.409599	-0.515821	L
9996	-0.479893	-0.650419	0.745152	-0.646038	E
9997	0.523331	0.787112	0.486066	1.093156	K
9998	-0.362559	0.598894	-1.843201	0.887292	G
9999	-0.096376	-1.012999	-0.657431	-0.573315	0

```
[10000 rows x 5 columns]
```

Многоточие ... означает, что строки в середине DataFrame опущены.

Чтобы прочитать только небольшое число строк (а не весь файл), нужно задать это число в параметре `nrows`:

```
In [43]: pd.read_csv("examples/ex6.csv", nrows=5)
```

```
Out[43]:
```

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q

Для чтения файла порциями задайте с помощью параметра `chunksize` размер порции в строках:

```
In [44]: chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)
```

```
In [45]: type(chunker)
```

```
Out[45]: pandas.io.parsers.readers.TextFileReader
```

Объект `TextParser`, возвращаемый функцией `pandas.read_csv`, позволяет читать файл порциями размера `chunksize`. Например, можно таким образом итеративно читать файл `ex6.csv`, агрегируя счетчики значений в столбце "key":

```
chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)

tot = pd.Series([], dtype='int64')
for piece in chunker:
    tot = tot.add(piece["key"].value_counts(), fill_value=0)
tot = tot.sort_values(ascending=False)
```

Имеем:

```
In [47]: tot[:10]
```

```
Out[47]:
```

E	368.0
X	364.0

```
L 346.0
O 343.0
Q 340.0
M 338.0
J 337.0
F 335.0
K 334.0
H 330.0
dtype: float64
```

У объекта `TextParser` имеется также метод `get_chunk`, который позволяет читать куски произвольного размера.

Вывод данных в текстовом формате

Данные можно экспортировать в формате с разделителями. Рассмотрим одну из приведенных выше операций чтения CSV-файла:

```
In [48]: data = pd.read_csv("examples/ex5.csv")
```

```
In [49]: data
```

```
Out[49]:
  something  a  b   c  d message
0      one  1  2  3.0  4      NaN
1      two  5  6  NaN  8    world
2     three  9 10 11.0 12     foo
```

С помощью метода `to_csv` объекта `DataFrame` мы можем вывести данные в файл через запятую:

```
In [50]: data.to_csv("examples/out.csv")
```

```
In [51]: !cat examples/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 [52]: import sys
```

```
In [53]: 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 указать какой-нибудь другой маркер:

```
In [54]: 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 [55]: 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 [56]: data.to_csv(sys.stdout, index=False, columns=["a", "b", "c"])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Обработка данных в других форматах с разделителями

Как правило, табличные данные можно загрузить с диска с помощью функции `pandas.read_csv` и родственных ей. Но иногда требуется ручная обработка. Не так уж необычно встретить файл, в котором одна или несколько строк сформированы неправильно, что сбивает `pandas.read_csv`. Для иллюстрации базовых средств рассмотрим небольшой CSV-файл:

```
In [57]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

Для любого файла с односимвольным разделителем можно воспользоваться стандартным модулем Python `csv`. Для этого передайте открытый файл или файлоподобный объект методу `csv.reader`:

```
In [58]: import csv

In [59]: f = open("examples/ex7.csv")

In [60]: reader = csv.reader(f)
```

Обход файла с помощью объекта `reader` дает кортежи значений в каждой строке после удаления кавычек:

```
In [61]: for line in reader:
.....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']

In [62]: f.close()
```

Далее можно произвести любые манипуляции, необходимые для преобразования данных к нужному виду. Пойдем по шагам. Сначала прочитаем файл в список строк:

```
In [63]: with open("examples/ex7.csv") as f:
.....:     lines = list(csv.reader(f))
```

Затем отделим строку-заголовок от остальных:

```
In [64]: header, values = lines[0], lines[1:]
```

Далее мы можем создать словарь столбцов, применив словарное включение и выражение `zip(*values)`, которое транспонирует строки и столбцы (помните, что для больших файлов эта операция потребляет много памяти):

```
In [65]: data_dict = {h: v for h, v in zip(header, zip(*values))}
```

```
In [66]: data_dict
```

```
Out[66]: {'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 создадим простой подкласс класса `csv.Dialect`:

```
class my_dialect(csv.Dialect):
    lineterminator = "\n"
    delimiter = ","
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

Параметры диалекта CSV можно задать также в виде именованных параметров `csv.reader`, не определяя подкласса:

```
reader = csv.reader(f, delimiter=",")
```

Возможные атрибуты `csv.Dialect` вместе с назначением каждого описаны в табл. 6.3.

Таблица 6.3. Параметры диалекта CSV

Аргумент	Описание
<code>delimiter</code>	Односимвольная строка, определяющая разделитель полей. По умолчанию <code>,</code>
<code>lineterminator</code>	Завершитель строк при выводе, по умолчанию <code>'\\r\\n'</code> . Объект <code>reader</code> игнорирует этот параметр, используя вместо него платформенное соглашение о концах строк
<code>quotechar</code>	Символ закавычивания для полей, содержащих специальные символы (например, разделитель). По умолчанию <code>'\"'</code>
<code>quoting</code>	Соглашение об употреблении кавычек. Допустимые значения: <code>csv.QUOTE_ALL</code> (заключать в кавычки все поля), <code>csv.QUOTE_MINIMAL</code> (только поля, содержащие специальные символы, например разделитель), <code>csv.QUOTE_NONNUMERIC</code> и <code>csv.QUOTE_NON</code> (не закрывать в кавычки). Полное описание см. в документации. По умолчанию <code>QUOTE_MINIMAL</code>
<code>skipinitialspace</code>	Игнорировать пробелы после каждого разделителя. По умолчанию <code>False</code>
<code>doublequote</code>	Как обрабатывать символ кавычки внутри поля. Если <code>True</code> , добавляется второй символ кавычки. Полное описание см. в документации.
<code>escapechar</code>	Строка для экранирования разделителя в случае, когда <code>quoting</code> равно <code>csv.QUOTE_NONE</code> . 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`. По счастью, функция `pandas.read_csv` умеет делать почти все необходимое – нужно только задать правильные параметры, – поэтому вам редко придется разбирать файлы вручную.

Для записи файлов с разделителями вручную можно использовать функцию `csv.writer`. Она 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",
 "cities_lived": ["Akron", "Nashville", "New York", "San Francisco"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars", "soccer"]},
               {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}]}
"""
```

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, булевы значения и `null`. Ключ любого объекта должен быть строкой. На Python существует несколько библиотек для чтения и записи JSON-данных. Здесь я воспользуюсь модулем `json`, потому что он входит в стандартную библиотеку Python. Для преобразования JSON-строки в объект Python служит метод `json.loads`:

```
In [68]: import json
```

```
In [69]: result = json.loads(obj)
```

```
In [70]: result
```

```
Out[70]:
```

```
{'name': 'Wes',
 'cities_lived': ['Akron', 'Nashville', 'New York', 'San Francisco'],
 'pet': None,
```

```
'siblings': [{'name': 'Scott',
              'age': 34,
              'hobbies': ['guitars', 'soccer']}],
{'name': 'Katie', 'age': 42, 'hobbies': ['diving', 'art']}]}
```

Напротив, метод `json.dumps` преобразует объект Python в формат JSON:

```
In [71]: asjson = json.dumps(result)
```

```
In [72]: asjson
```

```
Out[72]: '{"name": "Wes", "cities_lived": ["Akron", "Nashville", "New York", "San Francisco"], "pet": null, "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars", "soccer"]}, {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}]}'
```

Как именно преобразовывать объект JSON или список таких объектов в DataFrame или еще какую-то структуру данных для анализа, решать вам. Для удобства предлагается возможность передать список словарей (которые раньше были объектами JSON) конструктору DataFrame и выбрать подмножество полей данных:

```
In [73]: siblings = pd.DataFrame(result["siblings"], columns=["name", "age"])
```

```
In [74]: siblings
```

```
Out[74]:
   name age
0  Scott  34
1  Katie  42
```

Функция `pandas.read_json` умеет автоматически преобразовать наборы данных определенного вида в формате JSON в объекты Series или DataFrame. Например:

```
In [75]: !cat examples/example.json
```

```
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

Подразумеваемые по умолчанию параметры `pandas.read_json` предполагают, что каждый объект в JSON-массиве – строка таблицы:

```
In [76]: data = pd.read_json("examples/example.json")
```

```
In [77]: data
```

```
Out[77]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Более полный пример чтения и манипулирования данными в формате JSON (включая и вложенные записи) приведен при рассмотрении базы данных о продуктах питания USDA в главе 13.

Чтобы экспортировать данные из pandas в формате JSON, можно воспользоваться методами `to_json` объектов Series и DataFrame:

```
In [78]: data.to_json(sys.stdout)
```

```
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}
```

```
In [79]: data.to_json(sys.stdout, orient="records")
[{"a":1,"b":2,"c":3},{ "a":4,"b":5,"c":6},{ "a":7,"b":8,"c":9}]
```

XML и HTML: разбор веб-страниц

На Python написано много библиотек для чтения и записи данных в вездесущих форматах HTML и XML. В частности, библиотеки `lxml`, `Beautiful Soup` и `html5lib`. Хотя `lxml` в общем случае работает гораздо быстрее остальных, другие библиотеки лучше справляются с неправильно сформированными HTML- и XML-файлами.

В `pandas` имеется функция `pandas.read_html`, которая использует все вышеупомянутые библиотеки автоматического выделения таблиц из HTML-файлов и представления их в виде объектов `DataFrame`. Чтобы продемонстрировать, как это работает, я скачал из Федеральной корпорации страхования вкладов США HTML-файл (он упоминается в документации по `pandas`) со сведениями о банкротствах банков⁴. Сначала необходимо установить дополнительные библиотеки, необходимые функции `read_html`:

```
conda install lxml beautifulsoup4 html5lib
```

Если вы не пользуетесь `conda`, то можно с тем же успехом выполнить команду `pip install lxml`.

У функции `pandas.read_html` имеется много параметров, но по умолчанию она ищет и пытается разобрать все табличные данные внутри тегов `<table>`. Результатом является список объектов `DataFrame`:

```
In [80]: tables = pd.read_html("examples/fdic_failed_bank_list.html")
```

```
In [81]: len(tables)
Out[81]: 1
```

```
In [82]: failures = tables[0]
```

```
In [83]: failures.head()
Out[83]:
```

	Bank Name	City	ST	CERT	\
0	Allied Bank	Mulberry	AR	91	
1	The Woodbury Banking Company	Woodbury	GA	11297	
2	First CornerStone Bank	King of Prussia	PA	35312	
3	Trust Company Bank	Memphis	TN	9956	
4	North Milwaukee State Bank	Milwaukee	WI	20364	
	Acquiring Institution	Closing Date	Updated Date		
0	Today's Bank	September 23, 2016	November 17, 2016		
1	United Bank	August 19, 2016	November 17, 2016		
2	First-Citizens Bank & Trust Company	May 6, 2016	September 6, 2016		
3	The Bank of Fayette County	April 29, 2016	September 6, 2016		
4	First-Citizens Bank & Trust Company	March 11, 2016	June 16, 2016		

Поскольку в объекте `failures` много столбцов, `pandas` вставляет знак разрыва строки `\`.

Как будет показано в следующих главах, дальше мы можем произвести очистку и анализ данных, например посчитать количество банкротств по годам:

⁴ Полный список см. по адресу <https://www.fdic.gov/bank/individual/failed/banklist.html>.

```
In [84]: close_timestamps = pd.to_datetime(failures["Closing Date"])
```

```
In [85]: close_timestamps.dt.year.value_counts()
```

```
Out[85]:
```

```
2010    157
2009    140
2011     92
2012     51
2008     25
```

```
...
```

```
2004     4
2001     4
2007     3
2003     3
2000     2
```

```
Name: Closing Date, Length: 15, dtype: int64
```

Разбор XML с помощью lxml.objectify

XML (расширяемый язык разметки) – еще один популярный формат представления структурированных данных, поддерживающий иерархически вложенные данные, снабженные метаданными. Текст этой книги на самом деле представляет собой набор больших XML-документов.

Выше я продемонстрировал работу функции `pandas.read_html`, которая пользуется библиотекой `lxml` или `Beautiful Soup` для разбора HTML-файлов. Форматы XML и HTML структурно похожи, но XML более общий. Ниже я покажу, как с помощью `lxml` разбирать данные в формате XML.

Управление городского транспорта Нью-Йорка (MTA) публикует временные ряды с данными о работе автобусов и электричек. Мы сейчас рассмотрим данные о качестве обслуживания, хранящиеся в виде XML-файлов. Для каждой автобусной и железнодорожной компании существует свой файл (например, `Performance_MNR.xml` для компании MetroNorth Railroad), содержащий данные за 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`:

```
In [86]: from lxml import objectify

In [87]: path = "datasets/mta_perf/Performance_MNR.xml"

In [88]: with open(path) as f:
.....:     parsed = objectify.parse(f)

In [89]: 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 [91]: perf = pd.DataFrame(data)
```

```
In [92]: perf.head()
Out[92]:
```

	AGENCY_NAME	INDICATOR_NAME \		DESCRIPTION \	
0	Metro-North Railroad On-Time Performance (West of Hudson)				
1	Metro-North Railroad On-Time Performance (West of Hudson)				
2	Metro-North Railroad On-Time Performance (West of Hudson)				
3	Metro-North Railroad On-Time Performance (West of Hudson)				
4	Metro-North Railroad On-Time Performance (West of Hudson)				
	PERIOD_YEAR	PERIOD_MONTH	CATEGORY	FREQUENCY	INDICATOR_UNIT \
0	2008	1	Service Indicators	M	%
1	2008	2	Service Indicators	M	%
2	2008	3	Service Indicators	M	%
3	2008	4	Service Indicators	M	%
4	2008	5	Service Indicators	M	%
	YTD_TARGET	YTD_ACTUAL	MONTHLY_TARGET	MONTHLY_ACTUAL	
0	95.0	96.9	95.0	96.9	
1	95.0	96.0	95.0	95.0	
2	95.0	96.3	95.0	96.9	
3	95.0	96.8	95.0	98.3	
4	95.0	96.6	95.0	95.8	

Функция `pandas.read_xml` позволяет заменить этот код однострочным выражением:

```
In [93]: perf2 = pd.read_xml(path)

In [94]: perf2.head()
Out[94]:
```

	INDICATOR_SEQ	PARENT_SEQ	AGENCY_NAME \
0	28445	NaN	Metro-North Railroad
1	28445	NaN	Metro-North Railroad
2	28445	NaN	Metro-North Railroad
3	28445	NaN	Metro-North Railroad
4	28445	NaN	Metro-North Railroad

	INDICATOR_NAME \
0	On-Time Performance (West of Hudson)
1	On-Time Performance (West of Hudson)
2	On-Time Performance (West of Hudson)
3	On-Time Performance (West of Hudson)
4	On-Time Performance (West of Hudson)

	DESCRIPTION \
0	Percent of commuter trains that arrive at their destinations within 5 m...
1	Percent of commuter trains that arrive at their destinations within 5 m...
2	Percent of commuter trains that arrive at their destinations within 5 m...
3	Percent of commuter trains that arrive at their destinations within 5 m...
4	Percent of commuter trains that arrive at their destinations within 5 m...

	PERIOD_YEAR	PERIOD_MONTH	CATEGORY	FREQUENCY	DESIRED_CHANGE \
0	2008	1	Service Indicators	M	U
1	2008	2	Service Indicators	M	U
2	2008	3	Service Indicators	M	U
3	2008	4	Service Indicators	M	U
4	2008	5	Service Indicators	M	U

	INDICATOR_UNIT	DECIMAL_PLACES	YTD_TARGET	YTD_ACTUAL	MONTHLY_TARGET \
0	%	1	95.00	96.90	95.00
1	%	1	95.00	96.00	95.00
2	%	1	95.00	96.30	95.00
3	%	1	95.00	96.80	95.00
4	%	1	95.00	96.60	95.00

	MONTHLY_ACTUAL
0	96.90
1	95.00
2	96.90
3	98.30
4	95.80

Про обращение с более сложными XML-документами можно прочитать в строке документации функции `pandas.read_xml`, где описывается, как производить выборку и фильтрацию для извлечения конкретной таблицы.

6.2. Двоичные форматы данных

Один из самых простых способов эффективного хранения данных в двоичном формате – воспользоваться встроенным в Python методом *сериализации* `pickle()` (NumPy). У всех объектов `pandas` имеется метод `to_pickle`, который сохраняет данные на диске в виде pickle-файла:


```
In [95]: frame = pd.read_csv("examples/ex1.csv")
```

```
In [96]: frame
```

```
Out[96]:
```

```
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12    foo
```

```
In [97]: frame.to_pickle("examples/frame_pickle")
```

Вообще говоря, pickle-файлы можно прочитать только из Python-программы. Любой объект в этом формате можно прочитать с помощью встроенной функции `pickle` или – еще удобнее – с помощью функции `pandas.read_pickle`:

```
In [98]: pd.read_pickle("examples/frame_pickle")
```

```
Out[98]:
```

```
   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. При разработке `pandas` были приложены все усилия к тому, чтобы такого не случилось, но, возможно, наступит момент, когда придется «поломать» формат `pickle`.

В `pandas` встроена поддержка еще нескольких открытых двоичных форматов, в т. ч. `HDF5`, `ORC` и `Apache Parquet`. Например, если установить пакет `pyarrow` (`conda install pyarrow`), то можно будет читать файлы в формате `Parquet` с помощью функции `pandas.read_parquet`:

```
In [100]: fec = pd.read_parquet('datasets/fec/fec.parquet')
```

Ниже я приведу несколько примеров работы с `HDF5`, но призываю вас самостоятельно исследовать другие форматы, чтобы оценить их эффективность и пригодность для анализа в вашей задаче.

Чтение файлов Microsoft Excel

В `pandas` имеется также поддержка для чтения табличных данных в формате `Excel 2003` (и более поздних версий) с помощью класса `pandas.ExcelFile` или функции `pandas.read_excel`. На внутреннем уровне `ExcelFile` пользуется пакетами `xlrd` и `openpyxl` для чтения файлов в формате `XLS` и `XLSX` соответственно. Эти пакеты нужно устанавливать отдельно от `pandas` командой `pip` или `conda`:

```
conda install openpyxl xlrd
```

Для работы с классом `pandas.ExcelFile` создайте его экземпляр, передав конструктору путь к файлу с расширением `xls` или `xlsx`:

```
In [101]: xlsx = pd.ExcelFile("examples/ex1.xlsx")
```

Этот объект может показать список рабочих листов в файле:

```
In [102]: xlsx.sheet_names
Out[102]: ['Sheet1']
```

Прочитать данные из рабочего листа в объект DataFrame позволяет метод `parse`:

```
In [103]: xlsx.parse(sheet_name="Sheet1")
Out[103]:
   Unnamed: 0  a  b  c  d message
0           0  0  1  2  3  4  hello
1           1  1  5  6  7  8  world
2           2  2  9 10 11 12   foo
```

В этой таблице Excel имеется индексный столбец, поэтому мы можем указать его в аргументе `index_col`:

```
In [104]: xlsx.parse(sheet_name="Sheet1", index_col=0)
Out[104]:
   a  b  c  d message
0  1  2  3  4  hello
1  5  6  7  8  world
2  9 10 11 12   foo
```

Если вы собираетесь читать несколько рабочих листов из файла, то быстрее создать объект `pandas.ExcelFile`, но можно просто передать имя файла функции `pandas.read_excel`:

```
In [105]: frame = pd.read_excel("examples/ex1.xlsx", sheet_name="Sheet1")

In [106]: frame
Out[106]:
   Unnamed: 0  a  b  c  d message
0           0  0  1  2  3  4  hello
1           1  1  5  6  7  8  world
2           2  2  9 10 11 12   foo
```

Для записи данных pandas в файл формата Excel следует сначала создать объект `ExcelWriter`, а затем записать в него данные, пользуясь методом `to_excel` объектов pandas:

```
In [107]: writer = pd.ExcelWriter("examples/ex2.xlsx")

In [108]: frame.to_excel(writer, "Sheet1")

In [109]: writer.save()
```

Можно вместо этого передать методу `to_excel` путь к файлу, избежав тем самым создания `ExcelWriter`:

```
In [110]: frame.to_excel("examples/ex2.xlsx")
```

Формат HDF5

HDF5 – хорошо зарекомендовавший себя файловый формат для хранения больших объемов научных данных в виде массивов. Для работы с ним используется библиотека, написанная на C и имеющая интерфейсы ко многим языкам,

в том числе Java, Julia, Python и MATLAB. Акроним «HDF» в ее названии означает *hierarchical data format* (иерархический формат данных). Каждый HDF5-файл содержит внутри себя структуру узлов, напоминающую файловую систему, которая позволяет хранить несколько наборов данных вместе с относящимися к ним метаданными. В отличие от более простых форматов, HDF5 поддерживает сжатие на лету с помощью различных алгоритмов сжатия, что позволяет более эффективно хранить повторяющиеся комбинации данных. Для наборов данных, которые не помещаются в память, HDF5 – отличный выбор, потому что дает возможность эффективно читать и записывать небольшие участки гораздо больших массивов.

Чтобы начать работу с HDF5 в pandas, необходимо сначала установить пакет PyTables:

```
conda install pytables
```



Отметим, что в архиве PyPI пакет PyTables называется «tables», поэтому если будете устанавливать его с помощью pip, то выполняйте команду `pip install tables`.

Хотя библиотеки PyTables и h5py позволяют работать с файлами HDF5 непосредственно, pandas предоставляет высокоуровневый интерфейс, который упрощает сохранение объектов Series и DataFrame. Класс `HDFStore` работает как словарь и отвечает за низкоуровневые детали:

```
In [113]: frame = pd.DataFrame({"a": np.random.standard_normal(100)})
```

```
In [114]: store = pd.HDFStore("examples/mydata.h5")
```

```
In [115]: store["obj1"] = frame
```

```
In [116]: store["obj1_col"] = frame["a"]
```

```
In [117]: store
```

```
Out[117]:
<class 'pandas.io.pytables.HDFStore'>
File path: examples/mydata.h5
```

Объекты из HDF5-файла можно извлекать, как из словаря:

```
In [118]: store["obj1"]
```

```
Out[118]:
      a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

`HDFStore` поддерживает две схемы хранения: `'fixed'` (по умолчанию) и `'table'`. Последняя, вообще говоря, медленнее, но поддерживает запросы в специальном синтаксисе:

```
In [119]: store.put("obj2", frame, format="table")

In [120]: store.select("obj2", where=["index >= 10 and index <= 15"])
Out[120]:
      a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429
In [121]: store.close()
```

Метод `put` – явный вариант синтаксиса `store['obj2'] = frame`, но он позволяет задавать и другие параметры, например формат хранения.

Функция `pandas.read_hdf` дает лаконичный способ доступа к этой функциональности:

```
In [122]: frame.to_hdf("examples/mydata.h5", "obj3", format="table")

In [123]: pd.read_hdf("examples/mydata.h5", "obj3", where=["index < 5"])
Out[123]:
      a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

При желании можно удалить созданный HDF5-файл:

```
In [124]: import os

In [125]: os.remove("examples/mydata.h5")
```



Если вы работаете с данными, которые хранятся на удаленных серверах, например Amazon S3 или HDFS, то может оказаться более подходящим какой-нибудь другой двоичный формат, разработанный специально для распределенных хранилищ, например Apache Parquet (<http://parquet.apache.org/>).

Если вы собираетесь работать с очень большими объемами данных локально, 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.

6.3. ВЗАИМОДЕЙСТВИЕ С HTML И WEB API

Многие сайты предоставляют открытый API для получения данных в формате JSON или каком-то другом. Получить доступ к таким API из Python можно разными способами; я рекомендую простой пакет `requests` (<http://docs.python-requests.org>), который можно установить с помощью `pip` или `conda`:

```
conda install requests
```

Чтобы найти последние 30 заявок, касающихся `pandas` на GitHub, мы можем отправить с помощью библиотеки `requests` такой HTTP-запрос `GET`:

```
In [126]: import requests

In [127]: url = "https://api.github.com/repos/pandas-dev/pandas/issues"

In [128]: resp = requests.get(url)

In [129]: resp.raise_for_status()

In [130]: resp
Out[130]: <Response [200]>
```

Я рекомендую всегда вызывать метод `raise_for_status` после обращения к `requests.get`, чтобы проверить ошибки HTTP.

Метод `json` объекта `Response` возвращает объект Python, содержащий разобранные JSON-данные, представленные в виде словаря или списка (в зависимости от того, какой JSON-код был возвращен):

```
In [131]: data = resp.json()

In [132]: data[0]["title"]
Out[132]: 'REF: make copy keyword non-stateful'
```

Поскольку результаты зависят от данных, поступающих в режиме реального времени, при выполнении этого кода вы, скорее всего, увидите совершенно другую картину.

Каждый элемент в списке `data` – словарь, содержащий все данные на странице заявки в GitHub (кроме комментариев). Список `data` можно передать конструктору `DataFrame` и выделить интересующие нас поля:

```
In [133]: issues = pd.DataFrame(data, columns=["number", "title",
.....:                                     "labels", "state"])

In [134]: issues
Out[134]:
  number \
0    48062
1    48061
```

```

2  48060
3  48059
4  48058
..  ...
25 48032
26 48030
27 48028
28 48027
29 48026

                                title \
0                                REF: make copy keyword non-stateful
1                                STYLE: upgrade flake8
2  DOC: "Creating a Python environment" in "Creating a development environ...
3                                REGR: Avoid overflow with groupby sum
4  REGR: fix reset_index (Index.insert) regression with custom Index subcl...
..                                ...
25                                BUG: Union of multi index with EA types can lose EA dtype
26                                ENH: Add rolling.prod()
27                                CLN: Refactor groupby's _make_wrapper
28                                ENH: Support masks in groupby prod
29                                DEP: Add pip to environment.yml

                                labels \
0                                []
1  [{ 'id': 106935113, 'node_id': 'MDU6TGFiZWw3MDY5MzUxMTM=', 'url': 'https...
2  [{ 'id': 134699, 'node_id': 'MDU6TGFiZWw3MzQ2OTk=', 'url': 'https://api....
3  [{ 'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
4  [{ 'id': 32815646, 'node_id': 'MDU6TGFiZWwzMjgxNTY0Ng==', 'url': 'https:...
..                                ...
25 [{ 'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...
26 [{ 'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url': 'https://api.g...
27 [{ 'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
28 [{ 'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
29 [{ 'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...

state
0  open
1  open
2  open
3  open
4  open
..  ...
25 open
26 open
27 open
28 open
29 open
[30 rows x 4 columns]

```

Немного попотев, вы сможете создать высокоуровневые интерфейсы к распределенным API работы с вебом, которые возвращают объекты DataFrame для дальнейшего анализа.

6.4. ВЗАИМОДЕЙСТВИЕ С БАЗАМИ ДАННЫХ

В корпоративных системах большая часть данных хранится не в текстовых или Excel-файлах. Широко используются реляционные базы данных на осно-

ве SQL (например, SQL Server, PostgreSQL и MySQL), а равно альтернативные базы данных, быстро набирающие популярность. Выбор базы данных обычно диктуется производительностью, необходимостью поддержания целостности данных и потребностями приложения в масштабируемости.

В pandas есть несколько функций для упрощения загрузки результатов SQL-запроса в DataFrame. В качестве примера я создам базу данных SQLite3, целиком размещающуюся в памяти, и драйвер `sqlite3`, включенный в стандартную библиотеку Python:

```
In [135]: import sqlite3

In [136]: query = """
.....: CREATE TABLE test
.....: (a VARCHAR(20), b VARCHAR(20),
.....:  c REAL, d INTEGER
.....: );"""

In [137]: con = sqlite3.connect("mydata.sqlite")

In [138]: con.execute(query)
Out[138]: <sqlite3.Cursor at 0x7fdfd73b69c0>

In [139]: con.commit()
```

Затем вставлю несколько строк в таблицу:

```
In [140]: data = [("Atlanta", "Georgia", 1.25, 6),
.....: ("Tallahassee", "Florida", 2.6, 3),
.....: ("Sacramento", "California", 1.7, 5)]

In [141]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [142]: con.executemany(stmt, data)
Out[142]: <sqlite3.Cursor at 0x7fdfd73a00c0>

In [143]: con.commit()
```

Большинство драйверов SQL, имеющихся в Python, при выборе данных из таблицы возвращают список кортежей:

```
In [144]: cursor = con.execute("SELECT * FROM test")

In [145]: rows = cursor.fetchall()

In [146]: rows
Out[146]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

Этот список кортежей можно передать конструктору DataFrame, но необходимы еще имена столбцов, содержащиеся в атрибуте курсора `description`. Отметим, что в случае SQLite3 атрибут `description` дает только имена столбцов (прочие поля, упомянутые в спецификации API доступа к базам данных из Python, равны `None`), но некоторые другие драйверы баз данных возвращают больше информации о столбцах:

```

In [147]: cursor.description
Out[147]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [148]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[148]:
   a      b    c  d
0  Atlanta  Georgia  1.25  6
1  Tallahassee  Florida  2.60  3
2  Sacramento  California  1.70  5

```

Такое переформатирование не хочется выполнять при каждом запросе к базе данных. Проект SQLAlchemy (www.sqlalchemy.org) – популярная библиотека на Python, абстрагирующая многие различия между базами данных SQL. В pandas имеется функция `read_sql`, которая позволяет без труда читать данные из соединения, открытого SQLAlchemy. Для установки SQLAlchemy выполните команду

```
conda install sqlalchemy
```

Далее подключимся к той же самой базе SQLite с помощью SQLAlchemy и прочитаем данные из ранее созданной таблицы:

```

In [149]: import sqlalchemy as sqla

In [150]: db = sqla.create_engine("sqlite:///mydata.sqlite")

In [151]: pd.read_sql("SELECT * FROM test", db)
Out[151]:
   a      b    c  d
0  Atlanta  Georgia  1.25  6
1  Tallahassee  Florida  2.60  3
2  Sacramento  California  1.70  5

```

6.5. ЗАКЛЮЧЕНИЕ

Получение доступа к данным часто является первым шагом процесса анализа данных. В этой главе мы рассмотрели ряд полезных средств, позволяющих приступить к делу. В следующих главах мы более детально рассмотрим переформатирование данных, визуализацию, анализ временных рядов и другие вопросы.

Очистка и подготовка данных

Значительная часть времени программиста, занимающегося анализом и моделированием данных, уходит на подготовку данных: загрузку, очистку, преобразование и реорганизацию. Часто говорят, что это составляет 80 и даже более процентов работы аналитика. Иногда способ хранения данных в файлах или в базе не согласуется с алгоритмом обработки. Многие предпочитают писать преобразования данных из одной формы в другую на каком-нибудь универсальном языке программирования типа Python, Perl, R или Java либо с помощью имеющихся в UNIX средств обработки текста типа `sed` или `awk`. По счастью, `pandas` дополняет стандартную библиотеку Python высокоуровневыми, гибкими и производительными базовыми преобразованиями и алгоритмами, которые позволяют переформатировать данные без особых проблем.

Если вы наткнетесь на манипуляцию, которой нет ни в этой книге, ни вообще в библиотеке `pandas`, не стесняйтесь внести предложение в списке расылки или на сайте GitHub. Вообще, многое в `pandas` – в части как проектирования, так и реализации – обусловлено потребностями реальных приложений.

В этой главе мы обсудим средства работы с отсутствующими и повторяющимися данными, средства обработки строк и некоторые другие преобразования данных, применяемые в процессе анализа. А следующая глава будет посвящена различным способам комбинирования и реорганизации наборов данных.

7.1. ОБРАБОТКА ОТСУТСТВУЮЩИХ ДАННЫХ

Отсутствующие данные – типичное явление в большинстве аналитических приложений. При проектировании `pandas` в качестве одной из целей ставилась задача сделать работу с отсутствующими данными как можно менее болезненной. Например, при вычислении всех описательных статистик для объектов `pandas` отсутствующие данные не учитываются.

Способ представления отсутствующих данных в объектах `pandas` не идеален, но для большинства практических применений его достаточно. Для представления отсутствующих данных типа `float64` в `pandas` используется значение с плавающей точкой `NaN` (не число), которое мы будем называть *маркером*.

```
In [14]: float_data = pd.Series([1.2, -3.5, np.nan, 0])
```

```
In [15]: float_data
```

```
Out[15]:
```

```
0    1.2
```

```
1   -3.5
```

```

2    NaN
3    0.0
dtype: float64

```

Метод `isna` дает булев объект Series, в котором элементы `True` соответствуют отсутствующим значениям:

```

In [16]: float_data.isna()
Out[16]:
0    False
1    False
2     True
3    False
dtype: bool

```

В pandas мы приняли соглашение, заимствованное из языка программирования R, – обозначать отсутствующие данные NA – *not available* (недоступны). В статистических приложениях NA может означать, что данные не существуют или существуют, но не наблюдаемы (например, из-за сложностей сбора данных). В процессе очистки данных зачастую важно анализировать сами отсутствующие данные, чтобы выявить проблемы, относящиеся к их сбору, или потенциальное смещение, вызванное отсутствием данных.

Встроенное в Python значение `None` также рассматривается как NA в массивах объектов:

```

In [17]: string_data = pd.Series(["aardvark", np.nan, None, "avocado"])

```

```

In [18]: string_data
Out[18]:
0    aardvark
1         NaN
2         None
3     avocado
dtype: object

```

```

In [19]: string_data.isna()
Out[19]:
0    False
1     True
2     True
3    False
dtype: bool

```

```

In [20]: float_data = pd.Series([1, 2, None], dtype='float64')

```

```

In [21]: float_data
Out[21]:
0    1.0
1    2.0
2     NaN
dtype: float64

```

```

In [22]: float_data.isna()
Out[22]:
0    False
1    False

```

```
2 True
dtype: bool
```

В проекте `pandas` предпринята попытка сделать работу с отсутствующими данными максимально независимой от типа данных. Функции типа `pandas.isna` абстрагируют многие досаждающие детали. В табл. 7.1 приведен перечень некоторых функций, относящихся к обработке отсутствующих данных.

Таблица 7.1. Методы обработки отсутствующих данных

Метод	Описание
<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
<code>fillna</code>	Восполняет отсутствующие данные указанным значением или использует какой-нибудь метод интерполяции, например <code>"ffill"</code> или <code>"bfill"</code>
<code>isna</code>	Возвращает объект, содержащий булевы значения, которые показывают, какие значения отсутствуют
<code>notna</code>	Логическое отрицание <code>isna</code> ; возвращает <code>True</code> для присутствующих и <code>False</code> для отсутствующих значений

Фильтрация отсутствующих данных

Существует несколько способов фильтрации отсутствующих данных. Конечно, можно сделать это и вручную с помощью функции `pandas.isna` и булева индексирования, но часто бывает полезен метод `dropna`. Для `Series` он возвращает другой объект `Series`, содержащий только данные и значения индекса, отличные от `NA`:

```
In [23]: data = pd.Series([1, np.nan, 3.5, np.nan, 7])

In [24]: data.dropna()
Out[24]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

Это эквивалентно такому коду:

```
In [25]: data[data.notna()]
Out[25]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

В случае объектов `DataFrame` есть несколько способов избавиться от отсутствующих данных. 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 [26]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],
.....:                       [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])
```

```
In [27]: data
Out[27]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
In [28]: data.dropna()
Out[28]:
```

	0	1	2
0	1.0	6.5	3.0

Если передать параметр `how='all'`, то будут отброшены строки, которые целиком состоят из отсутствующих значений:

```
In [29]: data.dropna(how="all")
Out[29]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

Имейте в виду, что эти функции по умолчанию возвращают новые объекты, а не модифицируют содержимое исходного.

Для подобного же отбрасывания столбцов достаточно передать параметр `axis=1`:

```
In [30]: data[4] = np.nan

In [31]: data
Out[31]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
In [32]: data.dropna(axis="columns", how="all")
Out[32]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Допустим, требуется оставить только строки, содержащие определенное количество отсутствующих наблюдений. Этот порог можно задать с помощью аргумента `thresh`:

```
In [33]: df = pd.DataFrame(np.random.standard_normal((7, 3)))

In [34]: df.iloc[:4, 1] = np.nan

In [35]: df.iloc[:2, 2] = np.nan

In [36]: df
```

```
Out[36]:
```

	0	1	2
0	-0.204708	NaN	NaN
1	-0.555730	NaN	NaN
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [37]: df.dropna()
```

```
Out[37]:
```

	0	1	2
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [38]: df.dropna(thresh=2)
```

```
Out[38]:
```

	0	1	2
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Восполнение отсутствующих данных

Иногда отсутствующие данные желательно не отфильтровывать (и потенциально вместе с ними отбрасывать полезные данные), а каким-то способом заполнить «дыры». В большинстве случаев для этой цели можно использовать метод `fillna`. Ему передается константа, подставляемая вместо отсутствующих значений:

```
In [39]: df.fillna(0)
```

```
Out[39]:
```

	0	1	2
0	-0.204708	0.000000	0.000000
1	-0.555730	0.000000	0.000000
2	0.092908	0.000000	0.769023
3	1.246435	0.000000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Если передать методу `fillna` словарь, то можно будет подставлять вместо отсутствующих данных значение, зависящее от столбца:

```
In [40]: df.fillna({1: 0.5, 2: 0})
```

```
Out[40]:
```

	0	1	2
0	-0.204708	0.500000	0.000000
1	-0.555730	0.500000	0.000000
2	0.092908	0.500000	0.769023
3	1.246435	0.500000	-1.296221
4	0.274992	0.228913	1.352917

```
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Те же методы интерполяции, что применяются для переиндексации, годятся и для `fillna` (см. табл. 5.3):

```
In [41]: df = pd.DataFrame(np.random.standard_normal((6, 3)))
```

```
In [42]: df.iloc[2:, 1] = np.nan
```

```
In [43]: df.iloc[4:, 2] = np.nan
```

```
In [44]: df
```

```
Out[44]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	NaN	1.343810
3	-0.713544	NaN	-2.370232
4	-1.860761	NaN	NaN
5	-1.265934	NaN	NaN

```
In [45]: df.fillna(method="ffill")
```

```
Out[45]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	0.124121	-2.370232
5	-1.265934	0.124121	-2.370232

```
In [46]: df.fillna(method="ffill", limit=2)
```

```
Out[46]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	NaN	-2.370232
5	-1.265934	NaN	-2.370232

`fillna` умеет еще много чего, например восполнять данные, подставляя среднее или медиану:

```
In [47]: data = pd.Series([1., np.nan, 3.5, np.nan, 7])
```

```
In [48]: data.fillna(data.mean())
```

```
Out[48]:
```

0	1.000000
1	3.833333
2	3.500000
3	3.833333
4	7.000000

dtype: float64

Справочная информация об аргументах метода `fillna` приведена в табл. 7.2.

Таблица 7.2. Аргументы метода `fillna`

Аргумент	Описание
<code>value</code>	Скалярное значение или похожий на словарь объект для восполнения отсутствующих значений
<code>method</code>	Метод интерполяции: <code>"bfill"</code> (обратное восполнение) или <code>"ffill"</code> (прямое восполнение). По умолчанию <code>None</code>
<code>axis</code>	Ось, по которой производится восполнение (<code>"index"</code> или <code>"columns"</code>); по умолчанию <code>axis="index"</code>
<code>limit</code>	Для прямого и обратного восполнений максимальное количество непрерывно следующих друг за другом промежутков, подлежащих восполнению

7.2. ПРЕОБРАЗОВАНИЕ ДАННЫХ

До сих пор мы в этой главе занимались реорганизацией данных. Фильтрация, очистка и прочие преобразования составляют еще один, не менее важный класс операций.

Устранение дубликатов

Строки-дубликаты могут появиться в объекте `DataFrame` по разным причинам. Приведем пример:

```
In [49]: data = pd.DataFrame({"k1": ["one", "two"] * 3 + ["two"],
.....:                      "k2": [1, 1, 2, 3, 3, 4, 4]})
```

```
In [50]: data
```

```
Out[50]:
```

```
   k1 k2
```

```
0  one  1
```

```
1  two  1
```

```
2  one  2
```

```
3  two  3
```

```
4  one  3
```

```
5  two  4
```

```
6  two  4
```

Метод `duplicated` объекта `DataFrame` возвращает булев объект `Series`, который для каждой строки показывает, является она дубликатом или нет (все значения в столбцах в точности равны соответственным значениям в ранее встречавшейся строке):

```
In [51]: data.duplicated()
```

```
Out[51]:
```

```
0    False
```

```
1    False
```

```
2    False
```

```
3    False
```

```
4    False
```

```
5    False
```

```
6     True
```

```
dtype: bool
```

А метод `drop_duplicates` возвращает DataFrame, содержащий только строки, которым в массиве, возвращенном методом `duplicated`, соответствует значение `False`:

```
In [52]: data.drop_duplicates()
Out[52]:
   k1 k2
0  one  1
1  two  1
2  one  2
3  two  3
4  one  3
5  two  4
```

По умолчанию оба метода принимают во внимание все столбцы, но можно указать произвольное подмножество столбцов, которые необходимо исследовать на наличие дубликатов. Допустим, есть еще один столбец значений, и мы хотим отфильтровать строки, которые содержат повторяющиеся значения только в столбце `'k1'`:

```
In [53]: data["v1"] = range(7)

In [54]: data
Out[54]:
   k1 k2 v1
0  one  1  0
1  two  1  1
2  one  2  2
3  two  3  3
4  one  3  4
5  two  4  5
6  two  4  6
In [55]: data.drop_duplicates(subset=["k1"])

Out[55]:
   k1 k2 v1
0  one  1  0
1  two  1  1
```

По умолчанию методы `duplicated` и `drop_duplicates` оставляют первую встретившуюся строку с данной комбинацией значений. Но если задать параметр `keep="last"`, то будет оставлена последняя строка:

```
In [56]: data.drop_duplicates(["k1", "k2"], keep="last")
Out[56]:
   k1 k2 v1
0  one  1  0
1  two  1  1
2  one  2  2
3  two  3  3
4  one  3  4
6  two  4  6
```


Преобразование данных с помощью функции или отображения

Часто бывает необходимо произвести преобразование набора данных, исходя из значений в некотором массиве, объекте Series или столбце объекта DataFrame. Рассмотрим гипотетические данные о сортах мяса:

```
In [57]: data = pd.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 [58]: data
Out[58]:
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 (см. раздел «Применение функций и отображение» главы 5) принимает функцию или похожий на словарь объект, содержащий отображение, которое реализует преобразование значений:

```
In [60]: data["animal"] = data["food"].map(meat_to_animal)
```

```
In [61]: data
Out[61]:
   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 [62]: def get_animal(x):
.....:     return meat_to_animal[x]

In [63]: data["food"].map(get_animal)
Out[63]:
0      pig
1      pig
2      pig
3      cow
4      cow
5      pig
6      cow
7      pig
8  salmon
Name: food, dtype: object
```

Метод `map` – удобное средство выполнения поэлементных преобразований и других операций очистки.

Замена значений

Восполнение отсутствующих данных методом `fillna` можно рассматривать как частный случай более общей замены значений. Если метод `map`, как мы только что видели, позволяет модифицировать подмножество значений, хранящихся в объекте, то метод `replace` предлагает для этого более простой и гибкий интерфейс. Рассмотрим такой объект Series:

```
In [64]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [65]: data
Out[65]:
0      1.0
1    -999.0
2      2.0
3    -999.0
4   -1000.0
5      3.0
dtype: float64
```

Значение `-999` могло бы быть маркером отсутствия данных. Чтобы заменить все такие значения теми, которые понимает pandas, воспользуемся методом `replace`, порождающим новый объект Series:

```
In [66]: data.replace(-999, np.nan)
Out[66]:
0      1.0
1      NaN
2      2.0
3      NaN
4   -1000.0
5      3.0
dtype: float64
```

Чтобы заменить сразу несколько значений, нужно передать их список и заменяющее значение:

```
In [67]: data.replace([-999, -1000], np.nan)
Out[67]:
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

Если для каждого заменяемого значения нужно свое заменяющее, передайте список замен:

```
In [68]: data.replace([-999, -1000], [np.nan, 0])
Out[68]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

В аргументе можно передавать также словарь:

```
In [69]: data.replace({-999: np.nan, -1000: 0})
Out[69]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```



Метод `data.replace` – не то же самое, что метод `data.str.replace`, который выполняет поэлементную замену строки. Методы работы со строками будут рассмотрены при обсуждении объекта `Series` ниже в этой главе.

Переименование индексов осей

Как и значения в объекте `Series`, метки осей можно преобразовывать с помощью функции или отображения, порождающего новые объекты с другими метками. Оси можно также модифицировать на месте, не создавая новую структуру данных. Вот простой пример:

```
In [70]: data = pd.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 [71]: def transform(x):
.....:     return x[:4].upper()
```

```
In [72]: data.index.map(transform)
Out[72]: Index(['OHIO', 'COLO', 'NEW'], dtype='object')
```

Атрибуту `index` можно присваивать значение, т. е. модифицировать `DataFrame` на месте:

```
In [73]: data.index = data.index.map(transform)
```

```
In [74]: data
```

```
Out[74]:
```

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Если требуется создать преобразованный вариант набора данных, не меняя оригинал, то будет полезен метод `rename`:

```
In [75]: data.rename(index=str.title, columns=str.upper)
```

```
Out[75]:
```

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

Интересно, что `rename` можно использовать в сочетании с похожим на словарь объектом, который предоставляет новые значения для подмножества меток оси:

```
In [76]: data.rename(index={"OHIO": "INDIANA"},
.....:               columns={"three": "peekaboo"})
```

```
Out[76]:
```

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Метод `rename` избавляет от необходимости копировать объект `DataFrame` вручную и присваивать значения его атрибутам `index` и `columns`.

Дискретизация и группировка по интервалам

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 по ящикам, соответствующим возрасту – дискретной величине:

```
In [77]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Разобьем множество возрастов на интервалы: от 18 до 25, от 26 до 35, от 35 до 60 и наконец от 61 и старше. Для этой цели в `pandas` есть функция `pandas.cut`:

```
In [78]: bins = [18, 25, 35, 60, 100]
```

```
In [79]: age_categories = pd.cut(ages, bins)
```

```
In [80]: age_categories
```

```
Out[80]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

pandas возвращает специальный объект `Categorical`. Показанный выше результат – интервалы, вычисленные методом `pandas.cut`. Каждый интервал можно рассматривать как специальное (уникальное для pandas) значение, содержащее нижнюю и верхнюю границы:

```
In [81]: age_categories.codes
Out[81]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [82]: age_categories.categories
Out[82]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval
[int64, right]')

In [83]: age_categories.categories[0]
Out[83]: Interval(18, 25, closed='right')

In [84]: pd.value_counts(age_categories)
Out[84]:
(18, 25]    5
(25, 35]    3
(35, 60]    3
(60, 100]   1
dtype: int64
```

Заметим, что `pd.value_counts(cats)` – счетчики значений в каждом интервале, вычисленном методом `pandas.cut`.

В строковом представлении интервала круглая скобка означает, что соответствующий конец не включается (интервал *открыт*), а квадратная – что включается (интервал *замкнут*). Чтобы сделать открытым правый конец, следует задать параметр `right=False`:

```
In [85]: pd.cut(ages, bins, right=False)
Out[85]:
[(18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25, 35), [60, 100), [35,
60), [35, 60), [25, 35]]
Length: 12
Categories (4, interval[int64, left]): [(18, 25) < [25, 35) < [35, 60) < [60, 100))
```

Можно также самостоятельно задать имена интервалов, передав список или массив в параметре `labels`:

```
In [86]: group_names = ["Youth", "YoungAdult", "MiddleAged", "Senior"]

In [87]: pd.cut(ages, bins, labels=group_names)
Out[87]:
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senior', '
MiddleAged', 'MiddleAged', 'YoungAdult']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

Если передать методу `pandas.cut` целое число интервалов, а не явно заданные границы, то он разобьет данные на интервалы равной длины, исходя из мини-

мального и максимального значений. Рассмотрим группировку равномерно распределенных данных по четырем интервалам равной длины:

```
In [88]: data = np.random.uniform(size=20)

In [89]: pd.cut(data, 4, precision=2)
Out[89]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34,
0.55],
(0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64, right]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

Параметр `precision=2` означает, что числа следует выводить с двумя десятичными знаками после точки.

Родственная функция `pandas.qcut` распределяет данные, исходя из выборочных квантилей. Метод `pandas.cut` обычно создает интервалы, содержащие разное число точек, — это всецело определяется распределением данных. А `pandas.qcut` пользуется выборочными квантилями, поэтому по определению получаются интервалы приблизительно равного размера:

```
In [90]: data = np.random.standard_normal(1000)

In [91]: quartiles = pd.qcut(data, 4, precision=2)

In [92]: quartiles
Out[92]:
[(-0.026, 0.62], (0.62, 3.93], (-0.68, -0.026], (0.62, 3.93], (-0.026, 0.62], ...
, (-0.68, -0.026], (-0.68, -0.026], (-2.96, -0.68], (0.62, 3.93], (-0.68, -0.026]
]
Length: 1000
Categories (4, interval[float64, right]): [(-2.96, -0.68] < (-0.68, -0.026] < (-0.026, 0.62] < (0.62, 3.93]]

In [93]: pd.value_counts(quartiles)
Out[93]:
(-2.96, -0.68]      250
(-0.68, -0.026]     250
(-0.026, 0.62]      250
(0.62, 3.93]        250
dtype: int64
```

Как и в случае `pandas.cut`, можно задать величины квантилей (числа от 0 до 1 включительно) самостоятельно:

```
In [94]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.]).value_counts()
Out[94]:
(-2.9499999999999997, -1.187]      100
(-1.187, -0.0265]                  400
(-0.0265, 1.286]                   400
(1.286, 3.928]                     100
dtype: int64
```

Мы еще вернемся к методам `pandas.cut` и `pandas.qcut` ниже в этой главе, когда будем обсуждать агрегирование и групповые операции, поскольку эти функции дискретизации особенно полезны для анализа квантилей и групп.

Обнаружение и фильтрация выбросов

Фильтрация, или преобразование, выбросов – это в основном вопрос применения операций с массивами. Рассмотрим объект `DataFrame` с нормально распределенными данными:

```
In [95]: data = pd.DataFrame(np.random.standard_normal((1000, 4)))
```

```
In [96]: data.describe()
```

```
Out[96]:
```

```
0 1 2 3
```

count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.049091	0.026112	-0.002544	-0.051827
std	0.996947	1.007458	0.995232	0.998311
min	-3.645860	-3.184377	-3.745356	-3.428254
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.525865	2.735527	3.366626

Допустим, что мы хотим найти в одном из столбцов значения, превышающие 3 по абсолютной величине:

```
In [97]: col = data[2]
```

```
In [98]: col[col.abs() > 3]
```

```
Out[98]:
```

```
41    -3.399312
```

```
136    -3.745356
```

```
Name: 2, dtype: float64
```

Чтобы выбрать все строки, в которых встречаются значения, по абсолютной величине превышающие 3, мы можем воспользоваться методом `any` для булева объекта `DataFrame`:

```
In [99]: data[(data.abs() > 3).any(axis="columns")]
```

```
Out[99]:
```

	0	1	2	3
41	0.457246	-0.025907	-3.399312	-0.974657
60	1.951312	3.260383	0.963301	1.201206
136	0.508391	-0.196713	-3.745356	-1.520113
235	-0.242459	-3.056990	1.918403	-0.578828
258	0.682841	0.326045	0.425384	-3.428254
322	1.179227	-3.184377	1.369891	-1.074833
544	-3.548824	1.553205	-2.186301	1.277104
635	-0.578093	0.193299	1.397822	3.366626
782	-0.207434	3.525865	0.283070	0.544635
803	-3.645860	0.255475	-0.549574	-1.907459

Скобки вокруг `data.abs() > 3` необходимы, они означают, что метод `any` применяется к результату операции сравнения.

Можно также присваивать значения данным, удовлетворяющим какому-то критерию. Следующий код срезает значения, выходящие за границы интервала от -3 до 3:

```
In [100]: data[data.abs() > 3] = np.sign(data) * 3

In [101]: data.describe()
Out[101]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.050286	0.025567	-0.001399	-0.051765
std	0.992920	1.004214	0.991414	0.995761
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.000000	2.735527	3.000000

Выражение `np.sign(data)` равно 1 или -1 в зависимости от того, является значение `data` положительным или отрицательным:

```
In [102]: np.sign(data).head()
Out[102]:
```

	0	1	2	3
0	-1.0	1.0	-1.0	1.0
1	1.0	-1.0	1.0	-1.0
2	1.0	1.0	1.0	-1.0
3	-1.0	-1.0	1.0	-1.0
4	-1.0	1.0	-1.0	-1.0

Перестановки и случайная выборка

Переставить (случайным образом переупорядочить) объект `Series` или строки объекта `DataFrame` легко с помощью функции `numpy.random.permutation`. Если передать функции `permutation` длину оси, для которой производится перестановка, то будет возвращен массив целых чисел, описывающий новый порядок:

```
In [103]: df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))

In [104]: df
Out[104]:
```

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

```
In [105]: sampler = np.random.permutation(5)
```

```
In [106]: sampler
Out[106]: array([3, 1, 4, 2, 0])
```

Этот массив затем можно использовать для индексирования на основе `iloc` или, что эквивалентно, передать функции `take`:

```
In [107]: df.take(sampler)
Out[107]:
```

	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27
1	7	8	9	10	11	12	13


```
4 28 29 30 31 32 33 34
2 14 15 16 17 18 19 20
0 0 1 2 3 4 5 6
```

```
In [108]: df.iloc[sampler]
Out[108]:
   0  1  2  3  4  5  6
3 21 22 23 24 25 26 27
1  7  8  9 10 11 12 13
4 28 29 30 31 32 33 34
2 14 15 16 17 18 19 20
0  0  1  2  3  4  5  6
```

Вызвав `take` с параметром `axis="columns"`, мы могли бы также произвести перестановку столбцов:

```
In [109]: column_sampler = np.random.permutation(7)

In [110]: column_sampler
Out[110]: array([4, 6, 3, 2, 1, 0, 5])

In [111]: df.take(column_sampler, axis="columns")
Out[111]:
   4  6  3  2  1  0  5
0  4  6  3  2  1  0  5
1 11 13 10  9  8  7 12
2 18 20 17 16 15 14 19
3 25 27 24 23 22 21 26
4 32 34 31 30 29 28 33
```

Чтобы выбрать случайное подмножество без возвращения, можно использовать метод `sample` объектов `Series` и `DataFrame`:

```
In [112]: df.sample(n=3)
Out[112]:
   0  1  2  3  4  5  6
2 14 15 16 17 18 19 20
4 28 29 30 31 32 33 34
0  0  1  2  3  4  5  6
```

Чтобы сгенерировать выборку *с возвращением* (когда разрешается выбирать один и тот же элемент несколько раз), передайте методу `sample` аргумент `replace=True`:

```
In [113]: choices = pd.Series([5, 7, -1, 6, 4])

In [114]: choices.sample(n=10, replace=True)
Out[114]:
2  -1
0  5
3  6
1  7
4  4
0  5
4  4
0  5
4  4
4  4
dtype: int64
```

Вычисление индикаторных переменных

Еще одно преобразование, часто встречающееся в статистическом моделировании и машинном обучении, – *преобразование категориальной переменной в фиктивную, или индикаторную, матрицу*. Если в столбце объекта DataFrame встречается k различных значений, то можно построить матрицу или объект DataFrame с k столбцами, содержащими только нули и единицы. В библиотеке pandas для этого имеется функция `pandas.get_dummies`, хотя нетрудно написать и свою собственную. Рассмотрим пример DataFrame:

```
In [115]: df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
.....:                      "data1": range(6)})
```

```
In [116]: df
```

```
Out[116]:
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
In [117]: pd.get_dummies(df["key"])
```

```
Out[117]:
```

	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 соединить с другими данными. У функции `pandas.get_dummies` для этой цели предусмотрен аргумент `prefix`:

```
In [118]: dummies = pd.get_dummies(df["key"], prefix="key")
```

```
In [119]: df_with_dummy = df[["data1"]].join(dummies)
```

```
In [120]: df_with_dummy
```

```
Out[120]:
```

	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

О методе `DataFrame.join` мы подробнее поговорим в следующей главе.

Если некоторая строка DataFrame принадлежит нескольким категориям, то к созданию фиктивных переменных придется применить другой подход. Рассмотрим набор данных MovieLens 1M, который будет более подробно исследован в главе 13:

```
In [121]: mnames = ["movie_id", "title", "genres"]

In [122]: movies = pd.read_table("datasets/movielens/movies.dat", sep="::",
.....:                          header=None, names=mnames, engine="python")

In [123]: movies[:10]
Out[123]:
```

	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

В pandas реализован специальный метод объекта Series `str.get_dummies` (методы, имена которых начинаются со `str.`, подробнее обсуждаются в разделе 7.4 ниже) для обработки ситуации, когда принадлежность нескольким группам закодирована строкой с разделителями:

```
In [124]: dummies = movies["genres"].str.get_dummies("|")

In [125]: dummies.iloc[:10, :6]
Out[125]:
```

	Action	Adventure	Animation	Children's	Comedy	Crime
0	0	0	1	1	1	0
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	0
4	0	0	0	0	1	0
5	1	0	0	0	0	1
6	0	0	0	0	1	0
7	0	1	0	1	0	0
8	1	0	0	0	0	0
9	1	1	0	0	0	0

Затем, как и раньше, можно объединить этот результат с `movies`, добавив "Genre_" к именам столбцов в объекте DataFrame `dummies` с помощью метода `add_prefix`:

```
In [126]: movies_windic = movies.join(dummies.add_prefix("Genre_"))

In [127]: movies_windic.iloc[0]
Out[127]:
```

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, dtype: object

```



Для очень больших наборов данных такой способ построения индикаторных переменных для нескольких категорий быстрым не назовешь. Было бы лучше реализовать низкоуровневую функцию, которая пишет напрямую в массив NumPy, а затем обернуть результат объектом DataFrame.

В статистических приложениях бывает полезно сочетать функцию `pandas.get_dummies` с той или иной функцией дискретизации, например `pandas.cut`:

```

In [128]: np.random.seed(12345) # чтобы результат был повторяемым

In [129]: values = np.random.uniform(size=10)
In [130]: values
Out[130]:
array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532,
       0.7489, 0.6536])

In [131]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [132]: pd.get_dummies(pd.cut(values, bins))
Out[132]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
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

```

Мы еще вернемся к функции `pandas.get_dummies` в разделе «Создание индикаторных функций для моделирования» ниже.

7.3. РАСШИРЕНИЕ ТИПОВ ДАННЫХ



Это сравнительно новое дополнение к pandas, которое большинству пользователей вряд ли пригодится, но я включил его для полноты картины, поскольку ниже в нескольких местах буду ссылаться на расширенные типы и использовать их.

Pandas первоначально основывалась на возможностях библиотеки NumPy, которая эффективно реализует операции с массивами и предназначена в первую очередь для работы с числовыми данными. Многие концепции, заложенные в pandas, в частности отсутствующих данных, были реализованы на базе уже имеющегося в NumPy, при этом были приложены все усилия для сохранения максимальной совместимости с библиотеками, которые одновременно использовали NumPy и pandas.

Построение на базе NumPy стало причиной ряда недостатков, в том числе:

- обработка отсутствия данных некоторых типов, например целых и булевых, была неполной. Поэтому при наличии пропусков в таких данных pandas преобразовывала их тип в `float64` и использовала для представления отсутствующих значений `np.nan`. Это приводило к кумулятивному эффекту из-за тонких проблем во многих алгоритмах pandas;
- обработка наборов, содержащих много строковых данных, была вычислительно неэффективной и потребляла много памяти;
- некоторые типы данных, в т. ч. временные интервалы, временные дельты и временные метки с часовыми поясами, невозможно было поддерживать эффективно, не используя массивов объектов Python, операции с которыми обходятся дорого.

Недавно в pandas была добавлена система *расширения типов*, позволяющая добавлять новые типы, не поддерживаемые NumPy. С данными таких типов данных можно работать точно так же, как с данными, происходящими из массивов NumPy.

Рассмотрим пример создания объекта Series, содержащего целые числа с одним отсутствующим значением:

```
In [133]: s = pd.Series([1, 2, 3, None])
```

```
In [134]: s
```

```
Out[134]:
```

```
0    1.0
```

```
1    2.0
```

```
2    3.0
```

```
3    NaN
```

```
dtype: float64
```

```
In [135]: s.dtype
```

```
Out[135]: dtype('float64')
```

Главным образом из соображений обратной совместимости Series сохраняет унаследованное поведение: использование типа `float64` и `np.nan` для представления отсутствующих значений. Этот объект Series можно было бы также создать, указав тип `pandas.Int64Dtype`:

```
In [136]: s = pd.Series([1, 2, 3, None], dtype=pd.Int64Dtype())
```

```
In [137]: s
Out[137]:
0      1
1      2
2      3
3    <NA>
dtype: Int64
```

```
In [138]: s.isna()
Out[138]:
0    False
1    False
2    False
3     True
dtype: bool
```

```
In [139]: s.dtype
Out[139]: Int64Dtype()
```

Здесь маркер `<NA>` означает, что в массиве расширенного типа отсутствует значение. Для этого используется специальное маркерное значение `pandas.NA`:

```
In [140]: s[3]
Out[140]: <NA>
```

```
In [141]: s[3] is pd.NA
Out[141]: True
```

Для задания этого типа можно было бы использовать сокращенную запись `"Int64"` вместо `pd.Int64Dtype()`. Заглавная буква обязательна, иначе считалось бы, что это нерасширенный тип, основанный на `NumPy`:

```
In [142]: s = pd.Series([1, 2, 3, None], dtype="Int64")
```

В `pandas` имеется также расширенный тип специально для строковых данных, в котором не используются массивы `NumPy` (для его работы необходима библиотека `pyarrow`, которую придется устанавливать отдельно):

```
In [143]: s = pd.Series(['one', 'two', None, 'three'], dtype=pd.StringDtype())

In [144]: s
Out[144]:
0    one
1    two
2    <NA>
3   three
dtype: string
```

Такие массивы строк обычно потребляют гораздо меньше памяти и зачастую оказываются вычислительно эффективнее при выполнении операций над большими наборами данных.

`Categorical` – еще один важный расширенный тип, который мы подробнее обсудим в разделе 7.5 ниже. Сравнительно полный перечень расширенных типов на момент написания книги приведен в табл. 7.3.

Расширенные типы можно передавать методу `astype` объекта `Series`, что упрощает преобразования в процессе очистки:

```
In [145]: df = pd.DataFrame({"A": [1, 2, None, 4],
.....:                      "B": ["one", "two", "three", None],
.....:                      "C": [False, None, False, True]})
```

```
In [146]: df
Out[146]:
   A    B    C
0  1.0  one  False
1  2.0  two  None
2  NaN three False
3  4.0  None  True
```

```
In [147]: df["A"] = df["A"].astype("Int64")
```

```
In [148]: df["B"] = df["B"].astype("string")
```

```
In [149]: df["C"] = df["C"].astype("boolean")
```

```
In [150]: df
Out[150]:
   A    B    C
0   1  one  False
1   2  two  <NA>
2  <NA> three False
3   4  <NA>  True
```

Таблица 7.3. Расширенные типы данных в pandas

Расширенный тип	Описание
<code>BooleanDtype</code>	Булев тип, допускающий null; в виде строки передавать <code>"boolean"</code>
<code>CategoricalDtype</code>	Категориальный тип; в виде строки передавать <code>"category"</code>
<code>DatetimeTZDtype</code>	Дата и время с часовым поясом
<code>Float32Dtype</code>	32-разрядный тип с плавающей точкой, допускающий null; в виде строки передавать <code>"Float32"</code>
<code>Float64Dtype</code>	64-разрядный тип с плавающей точкой, допускающий null; в виде строки передавать <code>"Float64"</code>
<code>Int8Dtype</code>	8-разрядный тип целого со знаком, допускающий null; в виде строки передавать <code>"Int8"</code>
<code>Int16Dtype</code>	16-разрядный тип целого со знаком, допускающий null; в виде строки передавать <code>"Int16"</code>
<code>Int32Dtype</code>	32-разрядный тип целого со знаком, допускающий null; в виде строки передавать <code>"Int32"</code>
<code>Int64Dtype</code>	64-разрядный тип целого со знаком, допускающий null; в виде строки передавать <code>"Int64"</code>

Окончание табл. 7.3

Расширенный тип	Описание
<code>UInt8Dtype</code>	8-разрядный тип целого без знака, допускающий null; в виде строки передавать <code>"UInt8"</code>
<code>UInt16Dtype</code>	16-разрядный тип целого без знака, допускающий null; в виде строки передавать <code>"UInt16"</code>
<code>UInt32Dtype</code>	32-разрядный тип целого без знака, допускающий null; в виде строки передавать <code>"UInt32"</code>
<code>UInt64Dtype</code>	64-разрядный тип целого без знака, допускающий null; в виде строки передавать <code>"UInt64"</code>

7.4. 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 `re` расширяет этот инструментарий, позволяя применять методы строк и регулярных выражений к целым массивам и беря на себя возню с отсутствующими значениями.

Встроенные методы строковых объектов

Для многих приложений вполне достаточно встроенных методов работы со строками. Например, строку, в которой данные записаны через запятую, можно разбить на поля с помощью метода `split`:

```
In [151]: val = "a,b, guido"

In [152]: val.split(",")
Out[152]: ['a', 'b', ' guido']
```

Метод `split` часто употребляется вместе с методом `strip`, чтобы убрать пробельные символы (в том числе переходы на новую строку):

```
In [153]: pieces = [x.strip() for x in val.split(",")]

In [154]: pieces
Out[154]: ['a', 'b', 'guido']
```

Чтобы конкатенировать строки, применяя в качестве разделителя двойное двоеточие, можно использовать оператор сложения:

```
In [155]: first, second, third = pieces

In [156]: first + "::" + second + "::" + third
Out[156]: 'a::b::guido'
```

Но это недостаточно общий метод. Быстрее и лучше соответствует духу Python другой способ: передать список или кортеж методу `join` строки `::`:


```
In [157]: "::".join(pieces)
Out[157]: '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 [158]: "guido" in val
Out[158]: True
```

```
In [159]: val.index(",")
Out[159]: 1
```

```
In [160]: val.find(":")
Out[160]: -1
```

Разница между `find` и `index` состоит в том, что `index` возбуждает исключение, если строка не найдена, тогда как `find` возвращает `-1`:

```
In [161]: val.index(":")
-----
ValueError                                Traceback (most recent call last)
<ipython-input-161-bea4c4c30248> in <module>
----> 1 val.index(":")
ValueError: substring not found
```

Метод `count` возвращает количество вхождений подстроки:

```
In [162]: val.count(",")
Out[162]: 2
```

Метод `replace` заменяет вхождения образца указанной строкой. Он же применяется для удаления подстрок – достаточно в качестве заменяющей передать пустую строку:

```
In [163]: val.replace(",", "::")
Out[163]: 'a::b:: guido'
```

```
In [164]: val.replace(", ", "")
Out[164]: 'ab guido'
```

В табл. 7.3 перечислены некоторые методы работы со строками в Python.

Как мы вскоре увидим, во многих таких операциях можно использовать также регулярные выражения.

Таблица 7.3. Встроенные в Python методы строковых объектов

Метод	Описание
<code>count</code>	Возвращает количество неперекрывающихся вхождений подстроки в строку
<code>endswith</code>	Возвращает <code>True</code> , если строка оканчивается указанной подстрокой
<code>startswith</code>	Возвращает <code>True</code> , если строка начинается указанной подстрокой
<code>join</code>	Использовать данную строку как разделитель при конкатенации последовательности других строк

Метод	Описание
<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 первого 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 последнего 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 с обоих концов, в начале или в конце строки
<code>split</code>	Разбивает строку на список подстрок по указанному разделителю
<code>lower</code>	Преобразует буквы в нижний регистр
<code>upper</code>	Преобразует буквы в верхний регистр
<code>casefold</code>	Преобразует символы в нижний регистр, а зависящие от локали комбинации символов – в общую форму, допускающую сравнение
<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 и в других книгах имеется немало отличных пособий и справочных руководств.

Функции из модуля `re` можно отнести к трем категориям: сопоставление с образцом, замена и разбиение. Естественно, все они взаимосвязаны; регулярное выражение описывает образец, который нужно найти в тексте, а затем его уже можно применять для разных целей. Рассмотрим простой пример: требуется разбить строку в тех местах, где имеется сколько-то пробельных символов (пробелов, знаков табуляции и знаков новой строки).

Для сопоставления с одним или несколькими пробельными символами служит регулярное выражение `\s+`:

```
In [165]: import re

In [166]: text = "foo bar\t baz \tqux"

In [167]: re.split(r"\s+", text)
Out[167]: ['foo', 'bar', 'baz', 'qux']
```

При обращении `re.split('\s+', text)` сначала *компилируется* регулярное выражение, а затем его методу `split` передается заданный текст. Можно просто откомпилировать регулярное выражение методом `re.compile`, создав тем самым объект, допускающий повторное использование:

```
In [168]: regex = re.compile(r"\s+")

In [169]: regex.split(text)
Out[169]: ['foo', 'bar', 'baz', 'qux']
```

Чтобы получить список всех подстрок, отвечающих данному регулярному выражению, следует воспользоваться методом `findall`:

```
In [170]: regex.findall(text)
Out[170]: [' ', '\t ', '\t']
```



Чтобы не прибегать к громоздкому экранированию знаков `\` в регулярном выражении, пользуйтесь *простыми* (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, распознающее большинство адресов электронной почты:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com"""
pattern = r"[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
# Флаг re.IGNORECASE делает регулярное выражение нечувствительным к регистру
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Применение метода `findall` к этому тексту порождает список почтовых адресов:

```
In [172]: regex.findall(text)
Out[172]:
['dave@google.com',
230 | Chapter 7: Data Cleaning and Preparation
'steve@gmail.com',
'rob@gmail.com',
'ryan@yahoo.com']
```

Метод `search` возвращает специальный объект соответствия для первого встретившегося в тексте адреса. В нашем случае этот объект может сказать только о начальной и конечной позициях найденного в строке образца:

```
In [173]: m = regex.search(text)

In [174]: m
Out[174]: <re.Match object; span=(5, 20), match='dave@google.com'>

In [175]: text[m.start():m.end()]
Out[175]: 'dave@google.com'
```

Метод `regex.match` возвращает `None`, потому что находит соответствие образцу только в начале строки:

```
In [176]: print(regex.match(text))
None
```

Метод `sub` возвращает новую строку, в которой вхождения образца заменены указанной строкой:

```
In [177]: 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 [178]: pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})"
In [179]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Метод `groups` объекта соответствия, порожденного таким модифицированным регулярным выражением, возвращает кортеж компонентов образца:

```
In [180]: m = regex.match("wesm@bright.net")

In [181]: m.groups()
Out[181]: ('wesm', 'bright', 'net')
```

Если в образце есть группы, то метод `findall` возвращает список кортежей:

```
In [182]: regex.findall(text)
Out[182]:
[('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 [183]: 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. В табл. 7.5 приведена краткая сводка методов.

Таблица 7.5. Методы регулярных выражений

Метод	Описание
<code>findall</code>	Возвращает список всех непересекающихся образцов, найденных в строке
<code>finditer</code>	Аналогичен <code>findall</code> , но возвращает итератор
<code>match</code>	Ищет соответствие образцу в начале строки и факультативно выделяет в образце группы. Если образец найден, возвращает объект соответствия, иначе <code>None</code>
<code>search</code>	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
<code>split</code>	Разбивает строку на части в местах вхождения образца
<code>sub</code> , <code>subn</code>	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</code> , <code>\2</code> , ...

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 [184]: data = {"Dave": "dave@google.com", "Steve": "steve@gmail.com",
.....:          "Rob": "rob@gmail.com", "Wes": np.nan}
```

```
In [185]: data = pd.Series(data)
```

```
In [186]: data
Out[186]:
Dave    dave@google.com
Steve   steve@gmail.com
Rob      rob@gmail.com
Wes    NaN
dtype: object
```

```
In [187]: data.isna()
Out[187]:
Dave    False
Steve   False
Rob      False
Wes      True
dtype: bool
```

Методы строк и регулярных выражений можно применить к каждому значению с помощью метода `data.map` (которому передается лямбда или другая

функция), но для отсутствующих значений они «грохнутся». Чтобы справиться с этой проблемой, в классе Series есть методы для операций со строками, которые пропускают отсутствующие значения. Доступ к ним производится через атрибут `str`; например, вот как можно было бы с помощью метода `str.contains` проверить, содержит ли каждый почтовый адрес подстроку `'gmail'`:

```
In [188]: data.str.contains("gmail")
Out[188]:
Dave      False
Steve     True
Rob       True
Wes       NaN
dtype: object
```

Обратите внимание, что результат этой операции имеет тип `object`. В pandas имеются расширенные типы для специализированной обработки строк, целых чисел и булевых данных; до недавнего времени работа с этими типами вызывала некоторые трудности, если часть данных отсутствовала:

```
In [189]: data_as_string_ext = data.astype('string')

In [190]: data_as_string_ext
Out[190]:
Dave      dave@google.com
Steve     steve@gmail.com
Rob       rob@gmail.com
Wes       <NA>
dtype: string

In [191]: data_as_string_ext.str.contains("gmail")
Out[191]:
Dave      False
Steve     True
Rob       True
Wes       <NA>
dtype: boolean
```

Расширенные типы более подробно обсуждаются в разделе 7.3 выше.

Регулярные выражения тоже можно так использовать, равно как и их флаги типа `IGNORECASE`:

```
In [192]: pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"

In [193]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[193]:
Dave      [(dave, google, com)]
Steve     [(steve, gmail, com)]
Rob       [(rob, gmail, com)]
Wes       NaN
dtype: object
```

Существует два способа векторной выборки элементов: `str.get` или доступ к атрибуту `str` по индексу:

```
In [194]: matches = data.str.findall(pattern, flags=re.IGNORECASE).str[0]

In [195]: matches
```

```
Out[195]:
Dave      (dave, google, com)
Steve     (steve, gmail, com)
Rob       (rob, gmail, com)
Wes       NaN
dtype: object
```

```
In [196]: matches.str.get(1)
Out[196]:
Dave      google
Steve     gmail
Rob       gmail
Wes       NaN
dtype: object
```

Аналогичный синтаксис позволяет вырезать строки:

```
In [197]: data.str[:5]
Out[197]:
Dave      dave@
Steve     steve
Rob       rob@g
Wes       NaN
dtype: object
```

Метод `str.extract` возвращает запомненные группы регулярного выражения в виде объекта `DataFrame`:

```
In [198]: data.str.extract(pattern, flags=re.IGNORECASE)
Out[198]:
           0      1      2
Dave      dave  google  com
Steve     steve  gmail  com
Rob       rob   gmail  com
Wes      NaN   NaN   NaN
```

В табл. 7.6 перечислены дополнительные методы строк в `pandas`.

Таблица 7.6. Неполный перечень векторных методов строковых объектов

Метод	Описание
<code>cat</code>	Поэлементно конкатенирует строки с необязательным разделителем
<code>contains</code>	Возвращает булев массив, показывающий, содержит ли каждая строка указанный образец
<code>count</code>	Подсчитывает количество вхождений образца
<code>extract</code>	Использует регулярное выражение с группами, чтобы выделить одну или несколько строк из объекта <code>Series</code> , содержащего строки; результатом является <code>DataFrame</code> , содержащий по одному столбцу на каждую группу
<code>endswith</code>	Эквивалентно <code>x.endswith(pattern)</code> для каждого элемента
<code>startswith</code>	Эквивалентно <code>x.startswith(pattern)</code> для каждого элемента
<code>findall</code>	Возвращает список всех вхождений образца для каждой строки

Окончание табл. 7.6

Метод	Описание
<code>get</code>	Доступ по индексу ко всем элементам (выбрать <code>i</code> -й элемент)
<code>isalnum</code>	Эквивалентно встроенному методу <code>str.isalnum</code>
<code>isalpha</code>	Эквивалентно встроенному методу <code>str.isalpha</code>
<code>isdecimal</code>	Эквивалентно встроенному методу <code>str.isdecimal</code>
<code>isdigit</code>	Эквивалентно встроенному методу <code>str.isdigit</code>
<code>islower</code>	Эквивалентно встроенному методу <code>str.islower</code>
<code>isnumeric</code>	Эквивалентно встроенному методу <code>str.isnumeric</code>
<code>isupper</code>	Эквивалентно встроенному методу <code>str.isupper</code>
<code>join</code>	Объединяет строки в каждом элементе Series, вставляя между ними указанный разделитель
<code>len</code>	Вычисляет длину каждой строки
<code>lower</code> , <code>upper</code>	Преобразование регистра; эквивалентно <code>x.lower()</code> или <code>x.upper()</code> для каждого элемента
<code>match</code>	Вызывает <code>re.match</code> с указанным регулярным выражением для каждого элемента, возвращает список выделенных групп
<code>pad</code>	Дополняет строки пробелами слева, справа или с обеих сторон
<code>center</code>	Эквивалентно <code>pad(side='both')</code>
<code>repeat</code>	Дублирует значения; например, <code>s.str.repeat(3)</code> эквивалентно <code>x * 3</code> для каждой строки
<code>replace</code>	Заменяет вхождения образца указанной строкой
<code>slice</code>	Вырезает каждую строку в объекте Series
<code>split</code>	Разбивает строки по разделителю или по регулярному выражению
<code>strip</code>	Убирает пробельные символы, в т. ч. знак новой строки, с обеих сторон строки
<code>rstrip</code>	Убирает пробельные символы справа
<code>lstrip</code>	Убирает пробельные символы слева

7.5. КАТЕГОРИАЛЬНЫЕ ДАННЫЕ

В этом разделе мы познакомимся с типом pandas `Categorical`. Я покажу, как с его помощью повысить производительность и снизить потребление памяти при выполнении некоторых операций. Кроме того, я представлю инструменты, помогающие использовать категориальные данные в статистике и машинном обучении.

Для чего это нужно

Часто бывает, что столбец таблицы содержит небольшое множество повторяющихся значений. Мы уже встречались с функциями `unique` и `value_counts`, которые позволяют соответственно получить различные значения, встречающиеся в массиве, и подсчитать их частоты:

```
In [199]: values = pd.Series(['apple', 'orange', 'apple',
.....:                       'apple'] * 2)
```

```
In [200]: values
```

```
Out[200]:
```

```
0    apple
```

```
1    orange
```

```
2    apple
```

```
3    apple
```

```
4    apple
```

```
5    orange
```

```
6    apple
```

```
7    apple
```

```
dtype: object
```

```
In [201]: pd.unique(values)
```

```
Out[201]: array(['apple', 'orange'], dtype=object)
```

```
In [202]: pd.value_counts(values)
```

```
Out[202]:
```

```
apple    6
```

```
orange    2
```

```
dtype: int64
```

Во многих системах (для организации хранилищ данных, статистических расчетов и т. д.) разработаны специальные подходы к представлению данных с повторяющимися значениями с целью более эффективного хранения и вычислений. В хранилищах данных принято использовать так называемые таблицы измерений, которые содержат различные значения и на которые главные таблицы ссылаются по целочисленным ключам:

```
In [203]: values = pd.Series([0, 1, 0, 0] * 2)
```

```
In [204]: dim = pd.Series(['apple', 'orange'])
```

```
In [205]: values
```

```
Out[205]:
```

```
0    0
```

```
1    1
```

```
2    0
```

```
3    0
```

```
4    0
```

```
5    1
```

```
6    0
```

```
7    0
```

```
dtype: int64
```

```
In [206]: dim
```

```
Out[206]:
```

```
0   apple
1   orange
dtype: object
```

Метод `take` позволяет восстановить исходный объект `Series`, содержащий строки:

```
In [207]: dim.take(values)
Out[207]:
0   apple
1   orange
0   apple
0   apple
0   apple
1   orange
0   apple
0   apple
dtype: object
```

Это представление целыми числами называется *категориальным*, или *словарным*. Массив различных значений может называться *категориями*, *слова-рем* или *уровнями* данных. В этой книге употребляются термины *категориальное* и *категории*. Целые значения, ссылающиеся на категории, будем называть *кодами категорий*, или просто *кодами*.

Категориальное представление может дать заметное повышение производительности в аналитических приложениях. Мы также можем преобразовывать категории, не меняя их коды. Вот два примера преобразований, выполнение которых обходится сравнительно дешево:

- переименование категорий;
- добавление новой категории без изменения порядка или положения существующих.

Расширенный тип `Categorical` в `pandas`

В `pandas` имеется расширенный тип `Categorical`, специально предназначенный для хранения данных, представленных целочисленными категориями, т. е. в некоторой *кодировке*. Это популярная техника сжатия, применяемая, когда данные содержат много одинаковых значений. Она может дать значительный прирост производительности и одновременно снизить потребление памяти, особенно в случае строковых данных.

Рассмотрим уже встречавшийся ранее объект `Series`:

```
In [208]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2

In [209]: N = len(fruits)

In [210]: rng = np.random.default_rng(seed=12345)

In [211]: df = pd.DataFrame({'fruit': fruits,
.....:                      'basket_id': np.arange(N),
.....:                      'count': rng.integers(3, 15, size=N),
.....:                      'weight': rng.uniform(0, 4, size=N)},
.....:                      columns=['basket_id', 'fruit', 'count', 'weight'])
```

```
In [212]: df
Out[212]:
```

	basket_id	fruit	count	weight
0	0	apple	11	1.564438
1	1	orange	5	1.331256
2	2	apple	12	2.393235
3	3	apple	6	0.746937
4	4	apple	5	2.691024
5	5	orange	12	3.767211
6	6	apple	10	0.992983
7	7	apple	11	3.795525

Здесь `df['fruit']` – массив строковых объектов. Его можно следующим образом преобразовать в категориальную форму:

```
In [213]: fruit_cat = df['fruit'].astype('category')
```

```
In [214]: fruit_cat
Out[214]:
```

0	apple
1	orange
2	apple
3	apple
4	apple
5	orange
6	apple
7	apple

```
Name: fruit, dtype: category
Categories (2, object): ['apple', 'orange']
```

Теперь значением `fruit_cat` является экземпляр типа `pandas.Categorical`, который можно получить с помощью атрибута `.array`:

```
In [215]: c = fruit_cat.array

In [216]: type(c)
Out[216]: pandas.core.arrays.categorical.Categorical
Объект Categorical имеет атрибуты categories и codes:
In [217]: c.categories
Out[217]: Index(['apple', 'orange'], dtype='object')

In [218]: c.codes
Out[218]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

К ним можно обратиться проще, воспользовавшись аксессуаром `cat`, о котором я расскажу ниже в разделе «Категориальные методы».

Для получения отображения между кодами и категориями есть полезный прием:

```
In [219]: dict(enumerate(c.categories))
Out[219]: {0: 'apple', 1: 'orange'}
```

Столбец DataFrame можно преобразовать в категориальную форму с помощью такого присваивания:

```
In [220]: df['fruit'] = df['fruit'].astype('category')

In [221]: df["fruit"]
```

```
Out[221]:
0    apple
1    orange
2    apple
3    apple
4    apple
5    orange
6    apple
7    apple
Name: fruit, dtype: category
Categories (2, object): ['apple', 'orange']
```

Объект типа `pandas.Categorical` можно также создать непосредственно из других типов последовательностей Python:

```
In [222]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])

In [223]: my_categories
Out[223]:
['foo', 'bar', 'baz', 'foo', 'bar']
Categories (3, object): ['bar', 'baz', 'foo']
```

Если категориальные данные получены из другого источника, то можно воспользоваться альтернативным конструктором `from_codes`:

```
In [224]: categories = ['foo', 'bar', 'baz']

In [225]: codes = [0, 1, 2, 0, 0, 1]

In [226]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [227]: my_cats_2
Out[227]:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo', 'bar', 'baz']
```

Если явно не оговорено противное, при преобразовании данных в категориальную форму конкретный порядок категорий не предполагается. Поэтому порядок элементов в массиве `categories` может зависеть от того, как упорядочены входные данные. Но при использовании `from_codes` или любого другого конструктора можно указать, что порядок категорий имеет значение:

```
In [228]: ordered_cat = pd.Categorical.from_codes(codes, categories,
.....:                                           ordered=True)

In [229]: ordered_cat
Out[229]:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo' < 'bar' < 'baz']
```

Результат `[foo < bar < baz]` показывает, что 'foo' предшествует 'bar' и т. д. Неупорядоченный категориальный объект можно сделать упорядоченным с помощью метода `as_ordered`:

```
In [230]: my_cats_2.as_ordered()
Out[230]:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo' < 'bar' < 'baz']
```

И последнее замечание: категориальные данные необязательно должны быть строковыми, хотя во всех примерах выше я только такие и показывал. Элементы категориального массива могут иметь любой неизменяемый тип.

Вычисления с объектами `Categorical`

Объекты `Categorical` в `pandas`, вообще говоря, ведут себя так же, как незакодированные данные (скажем, массив строк). Но некоторые части `pandas`, например функция `groupby`, с категориальными данными работают быстрее. Для некоторых функций также имеет значение упорядоченность.

Возьмем случайные числовые данные и воспользуемся функцией распределения по интервалам `pandas.qcut`. Она возвращает объект `pandas.Categorical`; мы уже встречались с этой функцией раньше, но опустили детали работы с категориальными данными:

```
In [231]: rng = np.random.default_rng(seed=12345)

In [232]: draws = rng.standard_normal(1000)

In [233]: draws[:5]
Out[233]: array([-1.4238,  1.2637, -0.8707, -0.2592, -0.0753])
```

Вычислим квартильное распределение этих данных по интервалам и получим некоторые статистики:

```
In [234]: bins = pd.qcut(draws, 4)

In [235]: bins
Out[235]:
[(-3.121, -0.675], (0.687, 3.211], (-3.121, -0.675], (-0.675, 0.0134], (-0.675, 0.0134], ..., (0.0134, 0.687], (0.0134, 0.687], (-0.675, 0.0134], (0.0134, 0.687], (-0.675, 0.0134]]
Length: 1000
Categories (4, interval[float64, right]): [(-3.121, -0.675] < (-0.675, 0.0134] < (0.0134, 0.687] < (0.687, 3.211]]
```

Хотя точные выборочные квартили и полезны, но не так, как отчет, содержащий имена квартилей. Для получения такого отчета можно передать функции `qcut` аргумент `labels`:

```
In [236]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [237]: bins
Out[237]:
['Q1', 'Q4', 'Q1', 'Q2', 'Q2', ..., 'Q3', 'Q3', 'Q2', 'Q3', 'Q2']
Length: 1000
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']

In [238]: bins.codes[:10]
Out[238]: array([0, 3, 0, 1, 1, 0, 0, 2, 2, 0], dtype=int8)
```

Категориальный объект `bins` с метками не содержит информацию о границах интервалов, поэтому для получения сводных статистик воспользуемся методом `groupby`:

```
In [239]: bins = pd.Series(bins, name='quartile')
```

```
In [240]: results = (pd.Series(draws)
.....:                .groupby(bins)
.....:                .agg(['count', 'min', 'max'])
.....:                .reset_index())
```

```
In [241]: results
```

```
Out[241]:
```

	quartile	count	min	max
0	Q1	250	-3.119609	-0.678494
1	Q2	250	-0.673305	0.008009
2	Q3	250	0.018753	0.686183
3	Q4	250	0.688282	3.211418

В столбце результата `'quartile'` сохранена исходная категориальная информация из `bins`, включая упорядочение:

```
In [242]: results['quartile']
Out[242]:
```

0	Q1
1	Q2
2	Q3
3	Q4

```
Name: quartile, dtype: category
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

Повышение производительности с помощью перехода к категориальным данным

В начале этого раздела я сказал, что категориальные типы могут улучшить производительность и потребление памяти, теперь приведу несколько примеров. Рассмотрим объект Series, содержащий 10 млн элементов и небольшое число различных категорий:

```
In [243]: N = 10_000_000
```

```
In [244]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Теперь преобразуем `labels` в категориальную форму:

```
In [245]: categories = labels.astype('category')
```

Заметим, что `labels` занимает гораздо меньше памяти, чем `categories`:

```
In [246]: labels.memory_usage(deep=True)
Out[246]: 600000128
```

```
In [247]: categories.memory_usage(deep=True)
Out[247]: 10000540
```

Разумеется, переход к категориям обходится не бесплатно, но это одноразовые затраты:

```
In [248]: %time _ = labels.astype('category')
CPU times: user 469 ms, sys: 106 ms, total: 574 ms
Wall time: 577 ms
```

Переход к категориальной форме может значительно ускорить операции группировки, потому что лежащие в их основе алгоритмы теперь работают с массивом целочисленных кодов, а не строк. Ниже приведено сравнение быстродействия функции `value_counts()`, в которой используется механизм группировки:

```
In [249]: %timeit labels.value_counts()
840 ms +- 10.9 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [250]: %timeit categories.value_counts()
30.1 ms +- 549 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Категориальные методы

Объект `Series`, содержащий категориальные данные, имеет специальные методы для работы со строками, похожие на `Series.str`. Он также предоставляет удобный доступ к категориям и кодам. Рассмотрим следующий объект `Series`:

```
In [251]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [252]: cat_s = s.astype('category')
```

```
In [253]: cat_s
Out[253]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']
```

Специальный *акцессор* `cat` открывает доступ к категориальным методам:

```
In [254]: cat_s.cat.codes
Out[254]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [255]: cat_s.cat.categories
Out[255]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Допустим, что нам известно, что множество категорий для этих данных не ограничивается четырьмя наблюдаемыми в конкретном наборе значениями. Чтобы изменить множество категорий, воспользуемся методом `set_categories`:

```

In [256]: actual_categories = ['a', 'b', 'c', 'd', 'e']

In [257]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [258]: cat_s2
Out[258]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): ['a', 'b', 'c', 'd', 'e']

```

Хотя, на первый взгляд, данные не изменились, новые категории найдут отражения в операциях. Например, метод `value_counts` учитывает все категории:

```

In [259]: cat_s.value_counts()
Out[259]:
a    2
b    2
c    2
d    2
dtype: int64

In [260]: cat_s2.value_counts()
Out[260]:
a    2
b    2
c    2
d    2
e    0
dtype: int64

```

В больших наборах категориальные данные часто используются для экономии памяти и повышения производительности. После фильтрации большого объекта `DataFrame` или `Series` многих категорий в данных может не оказаться. Метод `remove_unused_categories` убирает ненаблюдаемые категории:

```

In [261]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [262]: cat_s3
Out[262]:
0    a
1    b
4    a
5    b
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

In [263]: cat_s3.cat.remove_unused_categories()
Out[263]:
0    a
1    b

```



```
4 a
5 b
dtype: category
Categories (2, object): ['a', 'b']
```

Перечень имеющихся категориальных методов приведен в табл. 7.7.

Таблица 7.7. Категориальные методы класса Series в pandas

Метод	Описание
<code>add_categories</code>	Добавить новые (невстречающиеся) категории в конец списка существующих категорий
<code>as_ordered</code>	Упорядочить категории
<code>as_unordered</code>	Не упорядочивать категории
<code>remove_categories</code>	Удалить категории, подставив вместо всех удаленных значений null
<code>remove_unused_categories</code>	Удалить категории, не встречающиеся в данных
<code>rename_categories</code>	Заменить имена категорий; количество категорий при этом должно остаться тем же
<code>reorder_categories</code>	Ведет себя так же, как <code>rename_categories</code> , но может и изменить результат, чтобы упорядочить категории
<code>set_categories</code>	Заменить старое множество категорий новым; при этом категории можно добавлять или удалять

Создание индикаторных переменных для моделирования

При работе с инструментами статистики или машинного обучения категориальные данные часто преобразуются в *индикаторные переменные*; этот процесс называется также *унитарным кодированием*. При этом создается объект DataFrame, содержащий по одному столбцу для каждой категории; в этих столбцах 1 обозначает присутствие данной категории, а 0 – ее отсутствие.

Рассмотрим прежний пример:

```
In [264]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

Как уже отмечалось выше в этой главе, функция `pandas.get_dummies` преобразует эти одномерные категориальные данные в объект DataFrame, содержащий индикаторную переменную:

```
In [265]: pd.get_dummies(cat_s)
Out[265]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

7.6. ЗАКЛЮЧЕНИЕ

Эффективные средства подготовки данных способны значительно повысить продуктивность, поскольку оставляют больше времени для анализа данных. В этой главе мы рассмотрели целый ряд инструментов, но наше изложение было далеко не полным. В следующей главе мы изучим имеющиеся в `pandas` средства соединения и группировки.

Переформатирование данных: соединение, комбинирование и изменение формы

Во многих приложениях бывает, что данные разбросаны по многим файлам или базам данных либо организованы так, что их трудно проанализировать. Эта глава посвящена средствам комбинирования, соединения и реорганизации данных.

Сначала мы познакомимся с концепцией *иерархического индексирования* в `pandas`, которая широко применяется в некоторых из описываемых далее операций. А затем перейдем к деталям конкретных манипуляций данными. В главе 13 будут продемонстрированы различные применения этих средств.

8.1. ИЕРАРХИЧЕСКОЕ ИНДЕКСИРОВАНИЕ

Иерархическое индексирование – важная особенность `pandas`, позволяющая организовать несколько (два и более) *уровней* индексирования по одной оси. По-другому можно сказать, что это способ работать с многомерными данными, представив их в форме меньшей размерности. Начнем с простого примера – создадим объект `Series` с индексом в виде списка списков или массивов:

```
In [11]: data = pd.Series(np.random.uniform(size=9),
.....: index=[["a", "a", "a", "b", "b", "c", "c", "d", "d"],
.....: [1, 2, 3, 1, 3, 1, 2, 2, 3]])
```

```
In [12]: data
Out[12]:
a 1    0.929616
  2    0.316376
  3    0.183919
b 1    0.204560
  3    0.567725
c 1    0.595545
  2    0.964515
d 2    0.653177
  3    0.748907
dtype: float64
```

Здесь мы видим отформатированное представление `Series` с мультииндексом (`MultiIndex`). «Разрывы» в представлении индекса означают «взять значение вышестоящей метки».

```
In [13]: data.index
Out[13]:
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

К иерархически индексированному объекту возможен доступ по так называемому частичному индексу, что позволяет лаконично записывать выборку подмножества данных:

```
In [14]: data["b"]
Out[14]:
1    0.204560
3    0.567725
dtype: float64

In [15]: data["b":"c"]
Out[15]:
b 1    0.204560
   3    0.567725
c 1    0.595545
   2    0.964515
dtype: float64

In [16]: data.loc[["b", "d"]]
Out[16]:
b 1    0.204560
   3    0.567725
d 2    0.653177
   3    0.748907
dtype: float64
```

В некоторых случаях возможна даже выборка с «внутреннего» уровня. Ниже я выбрал все элементы со значением 2 на втором уровне индекса:

```
In [17]: data.loc[:, 2]
Out[17]:
a    0.316376
c    0.964515
d    0.653177
dtype: float64
```

Иерархическое индексирование играет важнейшую роль в изменении формы данных и групповых операциях, в том числе построении сводных таблиц. Например, эти данные можно было бы преобразовать в DataFrame с помощью метода `unstack`:

```
In [18]: data.unstack()
Out[18]:
```

1	2	3

```

a  0.929616  0.316376  0.183919
b  0.204560      NaN  0.567725
c  0.595545  0.964515      NaN
d      NaN  0.653177  0.748907

```

Обратной к `unstack` операцией является `stack`:

```

In [19]: data.unstack().stack()
Out[19]:
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
dtype: float64

```

Методы `stack` и `unstack` будут подробно рассмотрены в разделе 8.3.

В случае DataFrame иерархический индекс может существовать для любой оси:

```

In [20]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
.....:                        index=[["a", "a", "b", "b"], [1, 2, 1, 2]],
.....:                        columns=[["Ohio", "Ohio", "Colorado"],
.....:                                ["Green", "Red", "Green"]])

In [21]: frame
Out[21]:
      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:

```

In [22]: frame.index.names = ["key1", "key2"]

In [23]: frame.columns.names = ["state", "color"]

In [24]: frame
Out[24]:
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

```

Эти имена замещают атрибут `name`, используемый только для одноуровневых индексов.



Обратите внимание, что имена индексов `"state"` и `"color"` не являются частями меток строк (значений `frame.index`).

Количество уровней индекса показывает атрибут `nlevels`:

```
In [25]: frame.index.nlevels
Out[25]: 2
```

Доступ по частичному индексу позволяет также выбирать группы столбцов:

```
In [26]: frame["Ohio"]
Out[26]:
```

	color	Green	Red
key1 key2			
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

Мультииндекс можно создать отдельно, а затем использовать повторно; в показанном выше объекте `DataFrame` столбцы с именами уровней можно было бы создать так:

```
pd.MultiIndex.from_arrays([["Ohio", "Ohio", "Colorado"],
                           ["Green", "Red", "Green"]],
                           names=["state", "color"])
```

Переупорядочение и уровни сортировки

Иногда требуется изменить порядок уровней на оси или отсортировать данные по значениям на одном уровне. Метод `swaplevel` принимает номера или имена двух уровней и возвращает новый объект, в котором эти уровни переставлены (но во всех остальных отношениях данные не изменяются):

```
In [27]: frame.swaplevel("key1", "key2")
Out[27]:
```

	state	Ohio	Colorado
color		Green	Red
key2 key1			Green
1	a	0	1
2	a	3	4
1	b	6	7
2	b	9	10

Метод `sort_index` по умолчанию сортирует данные лексикографически, используя все уровни индексов, но мы можем указать только один уровень или подмножество уровней, задав аргумент `level`. Например:

```
In [28]: frame.sort_index(level=1)
Out[28]:
```

	state	Ohio	Colorado
color		Green	Red
key1 key2			Green
a	1	0	1
b	1	6	7

```

a      2      3  4      5
b      2      9 10     11

In [29]: frame.swaplevel(0, 1).sort_index(level=0)
Out[29]:
state      Ohio      Colorado
color      Green Red      Green
key2 key1
1      a      0  1      2
      b      6  7      8
2      a      3  4      5
      b      9 10     11

```



Производительность выборки данных из иерархически индексированных объектов будет гораздо выше, если индекс отсортирован лексикографически, начиная с самого внешнего уровня, т. е. в результате вызова `sort_index(level=0)` или `sort_index()`.

Сводная статистика по уровню

У многих методов объектов `DataFrame` и `Series`, вычисляющих сводные и описательные статистики, имеется параметр `level` для задания уровня, на котором требуется производить агрегирование по конкретной оси. Рассмотрим тот же объект `DataFrame`, что и выше; мы можем суммировать по уровню для строк или для столбцов:

```

In [30]: frame.groupby(level="key2").sum()
Out[30]:
state Ohio      Colorado
color Green Red      Green
key2
1      6  8      10
2     12 14      16

In [31]: frame.groupby(level="color", axis="columns").sum()
Out[31]:
color      Green Red
key1 key2
a      1      2  1
      2      8  4
b      1     14  7
      2     20 10

```

Мы гораздо подробнее рассмотрим механизм `groupby` в главе 10.

Индексирование 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 [32]: frame = pd.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 [33]: frame
Out[33]:
```

	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 [34]: frame2 = frame.set_index(["c", "d"])

In [35]: frame2
Out[35]:
```

	a	b
one 0	0	7
1 1	1	6
2 2	2	5
two 0	3	4
1 1	4	3
2 2	5	2
3 3	6	1

По умолчанию столбцы удаляются из `DataFrame`, хотя их можно и оставить, передав методу `set_index` аргумент `drop=False`:

```
In [36]: frame.set_index(["c", "d"], drop=False)
Out[36]:
```

	a	b	c	d
one 0	0	7	one	0
1 1	1	6	one	1
2 2	2	5	one	2
two 0	3	4	two	0
1 1	4	3	two	1
2 2	5	2	two	2
3 3	6	1	two	3

Есть также метод `reset_index`, который делает прямо противоположное `set_index`; уровни иерархического индекса перемещаются в столбцы:

```
In [37]: frame2.reset_index()
Out[37]:
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

8.2. КОМБИНИРОВАНИЕ И СЛИЯНИЕ НАБОРОВ ДАННЫХ

Данные, хранящиеся в объектах `pandas`, можно комбинировать различными способами.

`pandas.merge`

Соединяет строки объектов `DataFrame` по одному или нескольким ключам. Эта операция хорошо знакома пользователям реляционных баз данных.

`pandas.concat`

Конкатенирует объекты, располагая их в стопку вдоль оси.

`combine_first`

Сращивает 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 `pandas.merge` – портал ко всем алгоритмам такого рода.

Начнем с простого примера:

```
In [38]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "a", "b"],
.....:                      "data1": pd.Series(range(7), dtype="Int64")})
```

```
In [39]: df2 = pd.DataFrame({"key": ["a", "b", "d"],
.....:                      "data2": pd.Series(range(3), dtype="Int64")})
```

```
In [40]: df1
Out[40]:
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

```
In [41]: df2
Out[41]:
```

	key	data2
0	a	0
1	b	1
2	d	2

Здесь я использовал расширенный тип `Int64` для целых, допускающих `null`; он обсуждается в разделе 7.3.

Это пример соединения типа *многие к одному*; в объекте `df1` есть несколько строк с метками `a` и `b`, а в `df2` – только одна строка для каждого значения в столбце `key`. Вызов `pandas.merge` для таких объектов дает:

```
In [42]: pd.merge(df1, df2)
```

```
Out[42]:
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

Обратите внимание, что я не указал, по какому столбцу производить соединение. В таком случае `merge` использует в качестве ключей столбцы с одинаковыми именами. Однако рекомендуется все же указывать столбцы явно:

```
In [43]: pd.merge(df1, df2, on="key")
```

```
Out[43]:
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

В общем случае порядок столбцов в результате операции `pandas.merge` не определен.

Если имена столбцов в объектах различаются, то можно задать их порознь:

```
In [44]: df3 = pd.DataFrame({"lkey": ["b", "b", "a", "c", "a", "a", "b"],
.....:                      "data1": pd.Series(range(7), dtype="Int64")})
```

```
In [45]: df4 = pd.DataFrame({"rkey": ["a", "b", "d"],
.....:                      "data2": pd.Series(range(3), dtype="Int64")})
```

```
In [46]: pd.merge(df3, df4, left_on="lkey", right_on="rkey")
```

```
Out[46]:
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

Вероятно, вы обратили внимание, что значения 'c' и 'd' и ассоциированные с ними данные отсутствуют в результирующем объекте. По умолчанию функция `merge` производит внутреннее соединение ('inner'); в результирующий объект попадают только ключи, присутствующие в обоих объектах-аргументах. Альтернативы: 'left', 'right' и 'outer'. В случае внешнего соединения ('outer') берется объединение ключей, т. е. получается то же самое, что при совместном применении левого и правого соединений:

```
In [47]: pd.merge(df1, df2, how="outer")
```

```
Out[47]:
```

	key	data1	data2

```
0  b    0    1
1  b    1    1
2  b    6    1
3  a    2    0
4  a    4    0
5  a    5    0
6  c    3  <NA>
7  d  <NA>    2
```

```
In [48]: pd.merge(df3, df4, left_on="lkey", right_on="rkey", how="outer")
Out[48]:
   lkey  data1  rkey  data2
0    b     0    b     1
1    b     1    b     1
2    b     6    b     1
3    a     2    a     0
4    a     4    a     0
5    a     5    a     0
6    c     3  NaN  <NA>
7  NaN  <NA>    d     2
```

При внешнем соединении строки из левого или правого объекта DataFrame, которым не нашлось равного ключа в другом объекте, представлены значениями NA в столбцах, происходящих из этого другого объекта. В табл. 8.1 перечислены возможные значения аргумента `how`.

Таблица 8.1. Различные типы соединения, задаваемые аргументом `how`

Значение	Поведение
'inner'	Брать только комбинации ключей, встречающиеся в обеих таблицах
'left'	Брать все ключи, встречающиеся в левой таблице
'right'	Брать все ключи, встречающиеся в правой таблице
'outer'	Брать все комбинации ключей

При соединении типа многие ко многим строится декартово произведение соответственных ключей. Вот пример:

```
In [49]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
.....:                      "data1": pd.Series(range(6), dtype="Int64")})
In [50]: df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
.....:                      "data2": pd.Series(range(5), dtype="Int64")})

In [51]: df1
Out[51]:
   key  data1
0    b     0
1    b     1
2    a     2
3    c     3
4    a     4
5    b     5
```

```

In [52]: df2
Out[52]:
   key  data2
0    a      0
1    b      1
2    a      2
3    b      3
4    d      4

In [53]: pd.merge(df1, df2, on="key", how="left")
Out[53]:
   key  data1  data2
0    b      0      1
1    b      0      3
2    b      1      1
3    b      1      3
4    a      2      0
5    a      2      2
6    c      3    <NA>
7    a      4      0
8    a      4      2
9    b      5      1
10   b      5      3

```

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, переданный в аргументе `how`, оказывает влияние только на множество различных ключей в результате:

```

In [54]: pd.merge(df1, df2, how="inner")
Out[54]:
   key  data1  data2
0    b      0      1
1    b      0      3
2    b      1      1
3    b      1      3
4    b      5      1
5    b      5      3
6    a      2      0
7    a      2      2
8    a      4      0
9    a      4      2

```

Для соединения по нескольким ключам передаем список имен столбцов:

```

In [55]: left = pd.DataFrame({"key1": ["foo", "foo", "bar"],
.....:                      "key2": ["one", "two", "one"],
.....:                      "lval": pd.Series([1, 2, 3], dtype='Int64')})

In [56]: right = pd.DataFrame({"key1": ["foo", "foo", "bar", "bar"],
.....:                        "key2": ["one", "one", "one", "two"],
.....:                        "rval": pd.Series([4, 5, 6, 7], dtype='Int64')})

In [57]: pd.merge(left, right, on=["key1", "key2"], how="outer")
Out[57]:
   key1 key2  lval  rval
0  foo  one     1     4

```

```
1 foo one      1      5
2 foo two      2  <NA>
3 bar one      3      6
4 bar two  <NA>    7
```

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.



При соединении столбцов по столбцам индексы над переданными объектами DataFrame отбрасываются. Если нужно сохранить значения индексов, то следует использовать `reset_index` для добавления индекса к столбцам.

Последний момент, касающийся операций слияния, – обработка столбцов с одинаковыми именами. Например:

```
In [58]: pd.merge(left, right, on="key1")
Out[58]:
   key1 key2_x  lval key2_y  rval
0  foo   one    1    one    4
1  foo   one    1    one    5
2  foo  two    2    one    4
3  foo  two    2    one    5
4  bar   one    3    one    6
5  bar   one    3    two    7
```

Хотя эту проблему можно решить вручную (см. раздел о переименовании меток на осях ниже), у функции `pandas.merge` имеется параметр `suffixes`, позволяющий задать строки, которые должны дописываться в конец одинаковых имен в левом и правом объектах DataFrame:

```
In [59]: pd.merge(left, right, on="key1", suffixes=("_left", "_right"))
Out[59]:
   key1 key2_left  lval key2_right  rval
0  foo     one    1         one    4
1  foo     one    1         one    5
2  foo    two    2         one    4
3  foo    two    2         one    5
4  bar     one    3         one    6
5  bar     one    3         two    7
```

В табл. 8.2 приведена справка по аргументам функции `pandas.merge`. Соединение с использованием индекса строк DataFrame – тема следующего раздела.

Таблица 8.2. 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: <code>'inner'</code> , <code>'outer'</code> , <code>'left'</code> , <code>'right'</code>

Аргумент	Описание
<code>on</code>	Имена столбцов, по которым производится соединение. Должны присутствовать в обоих объектах <code>DataFrame</code> . Если не заданы и не указаны никакие другие ключи соединения, то используются имена столбцов, общих для обоих объектов
<code>left_on</code>	Столбцы левого <code>DataFrame</code> , используемые как ключи соединения. Может быть указано имя одного столбца или список имен
<code>right_on</code>	То же для правого <code>DataFrame</code>
<code>left_index</code>	Использовать индекс строк левого <code>DataFrame</code> в качестве его ключа соединения (или нескольких ключей в случае мультииндекса)
<code>right_index</code>	То же, что <code>left_index</code> , но для правого <code>DataFrame</code>
<code>sort</code>	Сортировать слитые данные лексикографически по ключам соединения; по умолчанию <code>False</code>
<code>suffixes</code>	Кортеж строк, которые дописываются в конец совпадающих имен столбцов; по умолчанию <code>('_x', '_y')</code> . Например, если в обоих объектах <code>DataFrame</code> встречается столбец <code>'data'</code> , то в результирующем объекте появятся столбцы <code>'data_x'</code> и <code>'data_y'</code>
<code>copy</code>	Если равен <code>False</code> , то в некоторых особых случаях данные не копируются в результирующую структуру. По умолчанию данные копируются
<code>validate</code>	Проверяет, что тип слияния совпадает с указанным: один к одному, один ко многим или многие ко многим. Подробности см. в строке документации
<code>indicator</code>	Добавляет специальный столбец <code>_merge</code> , который сообщает об источнике каждой строки; он может принимать значения <code>'left_only'</code> , <code>'right_only'</code> или <code>'both'</code> в зависимости от того, как строка попала в результат соединения

Соединение по индексу

Иногда ключ (или ключи) соединения находится в индексе объекта `DataFrame`. В таком случае можно задать параметр `left_index=True` или `right_index=True` (или то и другое), чтобы указать, что в качестве ключа соединения следует использовать индекс:

```
In [60]: left1 = pd.DataFrame({"key": ["a", "b", "a", "a", "b", "c"],
.....:                        "value": pd.Series(range(6), dtype="Int64")})
```

```
In [61]: right1 = pd.DataFrame({"group_val": [3.5, 7]}, index=["a", "b"])
```

```
In [62]: left1
```

```
Out[62]:
```

```
   key  value
0    a      0
1    b      1
2    a      2
3    a      3
4    b      4
5    c      5
```



```
In [69]: lefth
Out[69]:
```

	key1	key2	data
0	Ohio	2000	0
1	Ohio	2001	1
2	Ohio	2002	2
3	Nevada	2001	3
4	Nevada	2002	4

```
In [70]: righth
Out[70]:
```

	event1	event2
Nevada 2001	0	1
2000	2	3
Ohio 2000	4	5
2000	6	7
2001	8	9
2002	10	11

В этом случае необходимо перечислить столбцы, по которым производится соединение, в виде списка (обратите внимание на обработку повторяющихся значений в индексе, когда `how='outer'`):

```
In [71]: pd.merge(lefth, righth, left_on=["key1", "key2"], right_index=True)
Out[71]:
```

	key1	key2	data	event1	event2
0	Ohio	2000	0	4	5
0	Ohio	2000	0	6	7
1	Ohio	2001	1	8	9
2	Ohio	2002	2	10	11
3	Nevada	2001	3	0	1

```
In [72]: pd.merge(lefth, righth, left_on=["key1", "key2"],
....:             right_index=True, how="outer")
Out[72]:
```

	key1	key2	data	event1	event2
0	Ohio	2000	0	4	5
0	Ohio	2000	0	6	7
1	Ohio	2001	1	8	9
2	Ohio	2002	2	10	11
3	Nevada	2001	3	0	1
4	Nevada	2002	4	<NA>	<NA>
4	Nevada	2000	<NA>	2	3

Использовать индексы в обеих частях слияния тоже возможно:

```
In [73]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
....:                         index=["a", "c", "e"],
....:                         columns=["Ohio", "Nevada"]).astype("Int64")

In [74]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
....:                           index=["b", "c", "d", "e"],
....:                           columns=["Missouri", "Alabama"]).astype("Int64")

In [75]: left2
Out[75]:
```


	Ohio	Nevada
a	1	2
c	3	4
e	5	6

```
In [76]: right2
```

```
Out[76]:
```

	Missouri	Alabama
b	7	8
c	9	10
d	11	12
e	13	14

```
In [77]: pd.merge(left2, right2, how="outer", left_index=True, right_index=True)
```

```
Out[77]:
```

	Ohio	Nevada	Missouri	Alabama
a	1	2	<NA>	<NA>
b	<NA>	<NA>	7	8
c	3	4	9	10
d	<NA>	<NA>	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 непересекающимися столбцами. В предыдущем примере можно было бы написать:

```
In [78]: left2.join(right2, how="outer")
```

```
Out[78]:
```

	Ohio	Nevada	Missouri	Alabama
a	1	2	<NA>	<NA>
b	<NA>	<NA>	7	8
c	3	4	9	10
d	<NA>	<NA>	11	12
e	5	6	13	14

По сравнению с `pandas.merge`, метод `join` объекта `DataFrame` по умолчанию выполняет левое внешнее соединение по ключам соединения. Он также поддерживает соединение с индексом переданного `DataFrame` по одному из столбцов вызывающего:

```
In [79]: left1.join(right1, on="key")
```

```
Out[79]:
```

	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

Этот метод можно представлять себе как присоединение данных к объекту, чей метод `join` был вызван.

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 [80]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
.....:                        index=["a", "c", "e", "f"],
.....:                        columns=["New York", "Oregon"])
```

```
In [81]: another
Out[81]:
   New York  Oregon
a         7.0     8.0
c         9.0    10.0
e        11.0    12.0
f        16.0    17.0
```

```
In [82]: left2.join([right2, another])
Out[82]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1       2      <NA>    <NA>       7.0     8.0
c      3       4        9      10       9.0    10.0
e      5       6       13      14      11.0    12.0
```

```
In [83]: left2.join([right2, another], how="outer")
Out[83]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1       2      <NA>    <NA>       7.0     8.0
c      3       4        9      10       9.0    10.0
e      5       6       13      14      11.0    12.0
b  <NA>  <NA>        7        8        NaN     NaN
d  <NA>  <NA>       11       12        NaN     NaN
f  <NA>  <NA>      <NA>    <NA>      16.0    17.0
```

Конкатенация вдоль оси

Еще одну операцию комбинирования данных разные авторы называют по-разному: *конкатенация* или *составление*. В библиотеке NumPy имеется функция `concatenate` для выполнения этой операции над массивами:

```
In [84]: arr = np.arange(12).reshape((3, 4))

In [85]: arr
Out[85]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [86]: np.concatenate([arr, arr], axis=1)
Out[86]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

В контексте объектов `pandas`, `Series` и `DataFrame` наличие помеченных осей позволяет обобщить конкатенацию массивов. В частности, нужно решить следующие вопросы.

- Если объекты по-разному проиндексированы по другим осям, следует ли объединять различные элементы на этих осях или использовать только общие значения?

- Нужно ли иметь возможность идентифицировать группы в результирующем объекте?
- Содержит ли «ось конкатенации» данные, которые необходимо сохранить? Во многих случаях подразумеваемые по умолчанию целочисленные метки в объекте DataFrame в процессе конкатенации лучше отбросить.

Функция `concat` в pandas дает согласованные ответы на эти вопросы. Я покажу, как она работает, на примерах. Допустим, имеются три объекта Series с непересекающимися индексами:

```
In [87]: s1 = pd.Series([0, 1], index=["a", "b"], dtype="Int64")
In [88]: s2 = pd.Series([2, 3, 4], index=["c", "d", "e"], dtype="Int64")
In [89]: s3 = pd.Series([5, 6], index=["f", "g"], dtype="Int64")
```

Если передать их функции `pandas.concat` списком, то она «склеит» данные и индексы:

```
In [90]: s1
Out[90]:
a    0
b    1
dtype: Int64

In [91]: s2
Out[91]:
c    2
d    3
e    4
dtype: Int64

In [92]: s3
Out[92]:
f    5
g    6
dtype: Int64

In [93]: pd.concat([s1, s2, s3])
Out[93]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: Int64
```

По умолчанию `pandas.concat` работает вдоль оси `axis="index"`, порождая новый объект Series. Но если передать параметр `axis="columns"`, то результатом будет DataFrame:

```
In [94]: pd.concat([s1, s2, s3], axis="columns")
Out[94]:
   0  1  2
```

```

a    0  <NA>  <NA>
b    1  <NA>  <NA>
c  <NA>    2  <NA>
d  <NA>    3  <NA>
e  <NA>    4  <NA>
f  <NA>  <NA>    5
g  <NA>  <NA>    6

```

В данном случае на другой оси нет перекрытия, и она, как видно, является отсортированным объединением (соединением в режиме "outer") индексов. Но можно образовать и пересечение индексов, если передать параметр `join="inner"`:

```

In [95]: s4 = pd.concat([s1, s3])

In [96]: s4
Out[96]:
a    0
b    1
f    5
g    6
dtype: Int64

In [97]: pd.concat([s1, s4], axis="columns")
Out[97]:
      0  1
a     0  0
b     1  1
f  <NA>  5
g  <NA>  6

In [98]: pd.concat([s1, s4], axis="columns", join="inner")
Out[98]:
      0  1
a     0  0
b     1  1

```

В последнем примере метки 'f' и 'g' пропали, поскольку был задан аргумент `join="inner"`.

Проблема может возникнуть из-за того, что в результирующем объекте не видно, конкатенацией каких объектов он получен. Допустим, что вы на самом деле хотите построить иерархический индекс на оси конкатенации. Для этого используется аргумент `keys`:

```

In [99]: result = pd.concat([s1, s1, s3], keys=["one", "two", "three"])

In [100]: result
Out[100]:
one  a    0
     b    1
two  a    0
     b    1
three f    5
      g    6
dtype: Int64

```

```
In [101]: result.unstack()
Out[101]:
```

	a	b	f	g
one	0	1	<NA>	<NA>
two	0	1	<NA>	<NA>
three	<NA>	<NA>	5	6

При комбинировании Series вдоль оси `axis="columns"` элементы списка `keys` становятся заголовками столбцов объекта DataFrame:

```
In [102]: pd.concat([s1, s2, s3], axis="columns", keys=["one", "two", "three"])
Out[102]:
```

	one	two	three
a	0	<NA>	<NA>
b	1	<NA>	<NA>
c	<NA>	2	<NA>
d	<NA>	3	<NA>
e	<NA>	4	<NA>
f	<NA>	<NA>	5
g	<NA>	<NA>	6

Эта логика обобщается и на объекты DataFrame:

```
In [103]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=["a", "b", "c"],
.....:                        columns=["one", "two"])
```

```
In [104]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=["a", "c"],
.....:                        columns=["three", "four"])
```

```
In [105]: df1
Out[105]:
```

	one	two
a	0	1
b	2	3
c	4	5

```
In [106]: df2
Out[106]:
```

	three	four
a	5	6
c	7	8

```
In [107]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"])
Out[107]:
```

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Здесь аргумент `keys` используется для создания иерархического индекса, первый уровень которого будет использован для идентификации каждого из конкатенированных объектов DataFrame.

Если передать не список, а словарь объектов, то роль аргумента `keys` будут играть ключи словаря:

```
In [108]: pd.concat({"level1": df1, "level2": df2}, axis="columns")
Out[108]:
```

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Дополнительные аргументы управляют созданием иерархического индекса (см. табл. 8.3). Например, можно поименовать созданные уровни на оси с помощью аргумента `names`:

```
In [109]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"],
.....:               names=["upper", "lower"])
Out[109]:
```

	upper level1		level2		
	lower	one	two	three	four
a		0	1	5.0	6.0
b		2	3	NaN	NaN
c		4	5	7.0	8.0

Последнее замечание касается объектов `DataFrame`, в которых индекс строк не содержит никаких релевантных данных:

```
In [110]: df1 = pd.DataFrame(np.random.standard_normal((3, 4)),
.....:                       columns=["a", "b", "c", "d"])

In [111]: df2 = pd.DataFrame(np.random.standard_normal((2, 3)),
.....:                       columns=["b", "d", "a"])

In [112]: df1
Out[112]:
```

	a	b	c	d
0	1.248804	0.774191	-0.319657	-0.624964
1	1.078814	0.544647	0.855588	1.343268
2	-0.267175	1.793095	-0.652929	-1.886837

```
In [113]: df2
Out[113]:
```

	b	d	a
0	1.059626	0.644448	-0.007799
1	-0.449204	2.448963	0.667226

В таком случае можно передать параметр `ignore_index=True`, тогда индексы из всех объектов `DataFrame` отбрасываются, а конкатенируются только данные в столбцах. При этом назначается новый индекс по умолчанию.

```
In [114]: pd.concat([df1, df2], ignore_index=True)
Out[114]:
```

	a	b	c	d
0	1.248804	0.774191	-0.319657	-0.624964
1	1.078814	0.544647	0.855588	1.343268
2	-0.267175	1.793095	-0.652929	-1.886837
3	-0.007799	1.059626	NaN	0.644448
4	0.667226	-0.449204	NaN	2.448963

В табл. 8.3 описаны аргументы функции `pandas.concat`.

Таблица 8.3. Аргументы функции `concat`

Аргумент	Описание
<code>objs</code>	Список или словарь конкатенируемых объектов <code>pandas</code> . Единственный обязательный аргумент
<code>axis</code>	Ось, вдоль которой производится конкатенация; по умолчанию по строкам (<code>axis="index"</code>)
<code>join</code>	Допустимые значения: <code>'inner'</code> , <code>'outer'</code> (по умолчанию); следует ли пересекать (<code>inner</code>) или объединять (<code>outer</code>) индексы вдоль других осей
<code>keys</code>	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 <code>levels</code> передаются массивы для нескольких уровней)
<code>levels</code>	Конкретные индексы, которые используются на одном или нескольких уровнях иерархического индекса, если задан параметр <code>keys</code>
<code>names</code>	Имена создаваемых уровней иерархического индекса, если заданы параметры <code>keys</code> и (или) <code>levels</code>
<code>verify_integrity</code>	Проверить новую ось в конкатенированном объекте на наличие дубликатов и, если они имеются, возбудить исключение. По умолчанию <code>False</code> – дубликаты разрешены
<code>ignore_index</code>	Не сохранять индексы вдоль оси конкатенации, а вместо этого создать новый индекс <code>range(total_length)</code>

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 выполняет эквивалент выражения `if-else` для массивов:

```
In [115]: a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
.....:                  index=["f", "e", "d", "c", "b", "a"])
```

```
In [116]: b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
.....:                  index=["a", "b", "c", "d", "e", "f"])
```

```
In [117]: a
Out[117]:
f    NaN
e    2.5
d    0.0
c    3.5
b    4.5
a    NaN
dtype: float64
```

```
In [118]: b
Out[118]:
```

```

a    0.0
b    NaN
c    2.0
d    NaN
e    NaN
f    5.0
dtype: float64

```

```

In [119]: np.where(pd.isna(a), b, a)
Out[119]: array([0. , 2.5, 0. , 3.5, 4.5, 5. ])

```

Здесь выбирается значение из `b`, если значение в `a` отсутствует, в противном случае выбирается значение из `a`. Функция `numpy.where` не проверяет, совмещены метки индекса или нет (и даже не требует, чтобы все объекты имели одинаковую длину). Поэтому если вы хотите выровнять значения в соответствии с индексом, то пользуйтесь методом `combine_first` объекта `Series`:

```

In [120]: a.combine_first(b)
Out[120]:
a    0.0
b    4.5
c    3.5
d    0.0
e    2.5
f    5.0
dtype: float64

```

В случае `DataFrame` метод `combine_first` делает то же самое для каждого столбца, так что можно считать, что он «подставляет» вместо данных, отсутствующих в вызывающем объекте, данные из объекта, переданного в аргументе:

```

In [121]: df1 = pd.DataFrame({"a": [1., np.nan, 5., np.nan],
.....:                      "b": [np.nan, 2., np.nan, 6.],
.....:                      "c": range(2, 18, 4)})

In [122]: df2 = pd.DataFrame({"a": [5., 4., np.nan, 3., 7.],
.....:                      "b": [np.nan, 3., 4., 6., 8.]})

In [123]: df1
Out[123]:
   a    b    c
0  1.0 NaN   2
1  NaN  2.0   6
2  5.0 NaN  10
3  NaN  6.0  14

In [124]: df2
Out[124]:
   a    b
0  5.0 NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0

In [125]: df1.combine_first(df2)

```



```
Out[125]:
   a    b    c
0  1.0 NaN  2.0
1  4.0  2.0  6.0
2  5.0  4.0 10.0
3  3.0  6.0 14.0
4  7.0  8.0 NaN
```

В результате `combine_first` применительно к объектам `DataFrame` будут присутствовать имена всех столбцов.

8.3. ИЗМЕНЕНИЕ ФОРМЫ И ПОВОРОТ

Существует ряд фундаментальных операций реорганизации табличных данных. Иногда их называют *изменением формы* (`reshape`), а иногда – *поворотом* (`pivot`).

Изменение формы с помощью иерархического индексирования

Иерархическое индексирование дает естественный способ реорганизовать данные в `DataFrame`. Есть два основных действия:

`stack`

Это «поворот», который переносит данные из столбцов в строки.

`unstack`

Обратный поворот, который переносит данные из строк в столбцы.

Проиллюстрирую эти операции на примерах. Рассмотрим небольшой `DataFrame`, в котором индексы строк и столбцов – массивы строк.

```
In [126]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
.....:                        index=pd.Index(["Ohio", "Colorado"], name="state"),
.....:                        columns=pd.Index(["one", "two", "three"],
.....:                                         name="number"))
```

```
In [127]: data
Out[127]:
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 [128]: result = data.stack()
```

```
In [129]: result
Out[129]:
state number
Ohio    one    0
         two    1
         three  2
```

```
Colorado one 3
          two 4
          three 5
dtype: int64
```

Имея иерархически проиндексированный объект Series, мы можем восстановить DataFrame методом `unstack`:

```
In [130]: result.unstack()
Out[130]:
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 [131]: result.unstack(level=0)
Out[131]:
state Ohio Colorado
number
one    0          3
two    1          4
three  2          5

In [132]: result.unstack(level="state")
Out[132]:
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 [133]: s1 = pd.Series([0, 1, 2, 3], index=["a", "b", "c", "d"], dtype="Int64")

In [134]: s2 = pd.Series([4, 5, 6], index=["c", "d", "e"], dtype="Int64")

In [135]: data2 = pd.concat([s1, s2], keys=["one", "two"])

In [136]: data2
Out[136]:
one a 0
    b 1
    c 2
    d 3
two c 4
    d 5
    e 6
dtype: Int64
```

При выполнении поворота отсутствующие данные по умолчанию отфильтровываются, поэтому операция обратима:

```

In [137]: data2.unstack()
Out[137]:
      a      b  c  d      e
one    0      1  2  3  <NA>
two  <NA>  <NA>  4  5      6

In [138]: data2.unstack().stack()
Out[138]:
one  a      0
     b      1
     c      2
     d      3
two  c      4
     d      5
     e      6
dtype: Int64

In [139]: data2.unstack().stack(dropna=False)
Out[139]:
one  a      0
     b      1
     c      2
     d      3
     e  <NA>
two  a  <NA>
     b  <NA>
     c      4
     d      5
     e      6
dtype: Int64

```

В случае обратного поворота DataFrame поворачиваемый уровень становится самым нижним уровнем результирующего объекта:

```

In [140]: df = pd.DataFrame({"left": result, "right": result + 5},
.....: columns=pd.Index(["left", "right"], name="side"))

In [141]: df
Out[141]:
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 [142]: df.unstack(level="state")
Out[142]:
side left      right
state Ohio Colorado Ohio Colorado
number
one      0         3     5         8
two      1         4     6         9
three    2         5     7        10

```

При вызове `stack` также можно указать имя поворачиваемой оси:

```
In [143]: df.unstack(level="state").stack(level="side")
Out[143]:
state      Colorado  Ohio
number side
one   left         3     0
      right        8     5
two   left         4     1
      right        9     6
three left         5     2
      right       10     7
```

Поворот из «длинного» в «широкий» формат

Стандартный способ хранения нескольких временных рядов в базах данных и в CSV-файлах – так называемый *длинный* формат (*в столбик*). В этом формате каждое значение представлено в отдельной строке (в противоположность нескольким значениям в одной строке).

Загрузим демонстрационные данные и займемся переформатированием временных рядов и другими операциями очистки данных:

```
In [144]: data = pd.read_csv("examples/macrodta.csv")

In [145]: data = data.loc[:, ["year", "quarter", "realgdp", "infl", "unemp"]]

In [146]: data.head()
Out[146]:
   year  quarter  realgdp  infl  unemp
0  1959         1  2710.349   0.00    5.8
1  1959         2  2778.801   2.34    5.1
2  1959         3  2775.488   2.74    5.3
3  1959         4  2785.204   0.27    5.6
4  1960         1  2847.699   2.31    5.2
```

Сначала я воспользовался функцией `pandas.PeriodIndex` (представляющей временные интервалы, а не моменты времени), которую мы будем подробно обсуждать в главе 11. Она объединяет столбцы `year` и `quarter`, так что индекс содержит значения типа `datetime` в конце каждого квартала:

```
In [147]: periods = pd.PeriodIndex(year=data.pop("year"),
.....:                             quarter=data.pop("quarter"),
.....:                             name="date")

In [148]: periods
Out[148]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
            '1960Q3', '1960Q4', '1961Q1', '1961Q2',
            ...
            '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
            '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', name='date', length=203)

In [149]: data.index = periods.to_timestamp("D")

In [150]: data.head()
```

```
Out[150]:
```

	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

Здесь я воспользовался методом `pop`, который удаляет столбец из объекта `DataFrame` и возвращает его.

Затем выбрал подмножество столбцов и присвоил индексу `columns` имя `"item"`:

```
In [151]: data = data.reindex(columns=["realgdp", "infl", "unemp"])
```

```
In [152]: data.columns.name = "item"
```

```
In [153]: data.head()
```

```
Out[153]:
```

item	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

И напоследок я изменил форму методом `stack`, преобразовал новые уровни индекса в столбцы методом `reset_index` и присвоил столбцу, содержащему значения данных, имя `"value"`:

```
In [154]: long_data = (data.stack()
.....:                  .reset_index()
.....:                  .rename(columns={0: "value"}))
```

Теперь `ldata` выглядит следующим образом:

```
In [155]: long_data[:10]
```

```
Out[155]:
```

	date	item	value
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340
5	1959-04-01	unemp	5.100
6	1959-07-01	realgdp	2775.488
7	1959-07-01	infl	2.740
8	1959-07-01	unemp	5.300
9	1959-10-01	realgdp	2785.204

В этом, так называемом *длинном*, формате для нескольких временных рядов каждая строка таблицы соответствует одному наблюдению.

Так данные часто хранятся в реляционных базах данных, поскольку при наличии фиксированной схемы (совокупность имен и типов данных столбцов) количество различных значений в столбце `item` может увеличиваться или уменьшаться при добавлении или удалении данных. В примере выше пара

столбцов `date` и `item` обычно выступает в роли первичного ключа (в терминологии реляционных баз данных), благодаря которому обеспечивается целостность данных и упрощаются многие операции соединения. Иногда с данными в таком формате трудно работать; предпочтительнее иметь объект `DataFrame`, содержащий по одному столбцу на каждое уникальное значение `item` и проиндексированный временными метками в столбце `date`. Метод `pivot` объекта `DataFrame` именно такое преобразование и выполняет:

```
In [156]: pivoted = long_data.pivot(index="date", columns="item",
.....:                               values="value")
```

```
In [157]: pivoted.head()
Out[157]:
item      infl r  realgdp  unemp
date
1959-01-01  0.00  2710.349    5.8
1959-04-01  2.34  2778.801    5.1
1959-07-01  2.74  2775.488    5.3
1959-10-01  0.27  2785.204    5.6
1960-01-01  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 [158]: long_data["value2"] = np.random.standard_normal(len(long_data))
```

```
In [159]: long_data[:10]
Out[159]:
   date      item      value      value2
0 1959-01-01  realgdp  2710.349  0.802926
1 1959-01-01    infl      0.000  0.575721
2 1959-01-01    unemp      5.800  1.381918
3 1959-04-01  realgdp  2778.801  0.000992
4 1959-04-01    infl      2.340 -0.143492
5 1959-04-01    unemp      5.100 -0.206282
6 1959-07-01  realgdp  2775.488 -0.222392
7 1959-07-01    infl      2.740 -1.682403
8 1959-07-01    unemp      5.300  1.811659
9 1959-10-01  realgdp  2785.204 -0.351305
```

Опустив последний аргумент, мы получим `DataFrame` с иерархическими столбцами:

```
In [160]: pivoted = long_data.pivot(index="date", columns="item")
```

```
In [161]: pivoted.head()
Out[161]:
   value      value2
item      infl realgdp unemp      infl realgdp  unemp
date
1959-01-01  0.00  2710.349  5.8  0.575721  0.802926  1.381918
1959-04-01  2.34  2778.801  5.1 -0.143492  0.000992 -0.206282
1959-07-01  2.74  2775.488  5.3 -1.682403 -0.222392  1.811659
1959-10-01  0.27  2785.204  5.6  0.128317 -0.351305 -1.313554
1960-01-01  2.31  2847.699  5.2 -0.615939  0.498327  0.174072
```

```
In [162]: pivoted["value"].head()
Out[162]:
item      infl  realgdp  unemp
date
1959-01-01  0.00  2710.349  5.8
1959-04-01  2.34  2778.801  5.1
1959-07-01  2.74  2775.488  5.3
1959-10-01  0.27  2785.204  5.6
1960-01-01  2.31  2847.699  5.2
```

Отметим, что метод `pivot` – эквивалент создания иерархического индекса методом `set_index` с последующим вызовом `unstack`:

```
In [163]: unstacked = long_data.set_index(["date", "item"]).unstack(level="item")

In [164]: unstacked.head()
Out[164]:
value      value2
item      infl  realgdp  unemp      infl  realgdp  unemp
date
1959-01-01  0.00  2710.349  5.8  0.575721  0.802926  1.381918
1959-04-01  2.34  2778.801  5.1 -0.143492  0.000992 -0.206282
1959-07-01  2.74  2775.488  5.3 -1.682403 -0.222392  1.811659
1959-10-01  0.27  2785.204  5.6  0.128317 -0.351305 -1.313554
1960-01-01  2.31  2847.699  5.2 -0.615939  0.498327  0.174072
```

Поворот из «широкого» в «длинный» формат

Обратной к `pivot` операцией является `pandas.melt`. Вместо того чтобы преобразовать один столбец в несколько в новом объекте `DataFrame`, она объединяет несколько столбцов в один, порождая `DataFrame` длиннее входного. Рассмотрим пример:

```
In [166]: df = pd.DataFrame({"key": ["foo", "bar", "baz"],
.....:                      "A": [1, 2, 3],
.....:                      "B": [4, 5, 6],
.....:                      "C": [7, 8, 9]})

In [167]: df
Out[167]:
key  A  B  C
0  foo  1  4  7
1  bar  2  5  8
2  baz  3  6  9
```

Столбец `'key'` может быть индикатором группы, а остальные столбцы – значениями данных. При использовании функции `pandas.melt` необходимо указать, какие столбцы являются индикаторами группы (если таковые имеются). Будем считать, что в данном случае `"key"` – единственный индикатор группы:

```
In [168]: melted = pd.melt(df, id_vars="key")

In [169]: melted
Out[169]:
key variable  value
0  foo      A      1
1  bar      A      2
```

```

2 baz      A      3
3 foo      B      4
4 bar      B      5
5 baz      B      6
6 foo      C      7
7 bar      C      8
8 baz      C      9

```

Применив `pivot`, мы можем вернуться к исходной форме:

```
In [170]: reshaped = melted.pivot(index="key", columns="variable",
.....:                             values="value")
```

```
In [171]: reshaped
Out[171]:
variable A B C
key
bar      2 5 8
baz      3 6 9
foo      1 4 7

```

Поскольку в результате работы `pivot` из столбца создается индекс, используемый как метки строк, возможно, понадобится вызвать `reset_index`, чтобы переместить данные обратно в столбец:

```
In [172]: reshaped.reset_index()
Out[172]:
variable key A B C
0        bar 2 5 8
1        baz 3 6 9
2        foo 1 4 7

```

Можно также указать, какое подмножество столбцов следует использовать в роли значений:

```
In [173]: pd.melt(df, id_vars="key", value_vars=["A", "B"])
Out[173]:
   key variable value
0  foo         A     1
1  bar         A     2
2  baz         A     3
3  foo         B     4
4  bar         B     5
5  baz         B     6

```

Функцию `pandas.melt` можно использовать и без идентификаторов групп:

```
In [174]: pd.melt(df, value_vars=["A", "B", "C"])
Out[174]:
   variable value
0         A     1
1         A     2
2         A     3
3         B     4
4         B     5
5         B     6
6         C     7
7         C     8
8         C     9

```



```
In [175]: pd.melt(df, value_vars=["key", "A", "B"])
```

```
Out[175]:
```

	variable	value
0	key	foo
1	key	bar
2	key	baz
3	A	1
4	A	2
5	A	3
6	B	4
7	B	5
8	B	6

8.4. ЗАКЛЮЧЕНИЕ

Теперь, вооружившись базовыми знаниями о применении pandas для импорта, очистки и реорганизации данных, мы готовы перейти к визуализации с помощью matplotlib. Но мы еще вернемся к pandas, когда начнем обсуждать дополнительные средства аналитики.

Построение графиков и визуализация

Информативная визуализация (называемая также *построением графиков*) – одна из важнейших задач анализа данных. Она может быть частью процесса исследования, например применяться для выявления выбросов, определения необходимых преобразований данных или поиска идей для построения моделей. В других случаях построение интерактивной визуализации для веб-сайта может быть конечной целью. Для Python имеется много дополнительных библиотек статической и динамической визуализации, но я буду использовать в основном matplotlib (<http://matplotlib.sourceforge.net>) и надстроенные над ней библиотеки.

Matplotlib – это пакет для построения графиков (главным образом двумерных) полиграфического качества. Проект был основан Джоном Хантером в 2002 году с целью реализовать на Python интерфейс, аналогичный MATLAB. Впоследствии сообщества matplotlib и IPython совместно работали над тем, чтобы упростить интерактивное построение графиков из оболочки IPython (а теперь и Jupyter-блокнотов). Matplotlib поддерживает разнообразные графические интерфейсы пользователя во всех операционных системах, а также умеет экспортировать графические данные во всех векторных и растровых форматах: PDF, SVG, JPG, PNG, BMP, GIF и т. д. С его помощью я построил почти все рисунки для этой книги, за исключением нескольких диаграмм.

Со временем над matplotlib было надстроено много дополнительных библиотек визуализации. Одну из них, seaborn (<http://seaborn.pydata.org/>), мы будем изучать в этой главе.

Для проработки приведенных в этой главе примеров кода проще всего воспользоваться интерактивным построением графиков в Jupyter-блокноте. Чтобы настроить этот режим, выполните в Jupyter-блокноте такую команду:

```
%matplotlib inline
```



После выхода первого издания этой книги в 2012 году появилось много новых библиотек визуализации данных, и некоторые из них (например, Vokeh и Altair) пользуются преимуществами современных веб-технологий для создания интерактивных визуализаций, которые хорошо интегрируются с Jupyter-блокнотом. Но я решил не разбрасываться и остался верным библиотеке matplotlib, прекрасно подходящей для обучения основам, особенно учитывая хорошую интеграцию pandas и matplotlib. А вы можете на основе описанных в этой главе принципов научиться использовать и другие библиотеки визуализации.

9.1. КРАТКОЕ ВВЕДЕНИЕ В API БИБЛИОТЕКИ MATPLOTLIB

При работе с `matplotlib` мы будем использовать следующее соглашение об импорте:

```
In [13]: import matplotlib.pyplot as plt
```

После выполнения команды `%matplotlib notebook` в Jupyter (или просто `%matplotlib` в IPython) уже можно создать простой график. Если все настроено правильно, то должна появиться прямая линия, показанная на рис. 9.1:

```
In [14]: data = np.arange(10)
```

```
In [15]: data
```

```
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [16]: plt.plot(data)
```

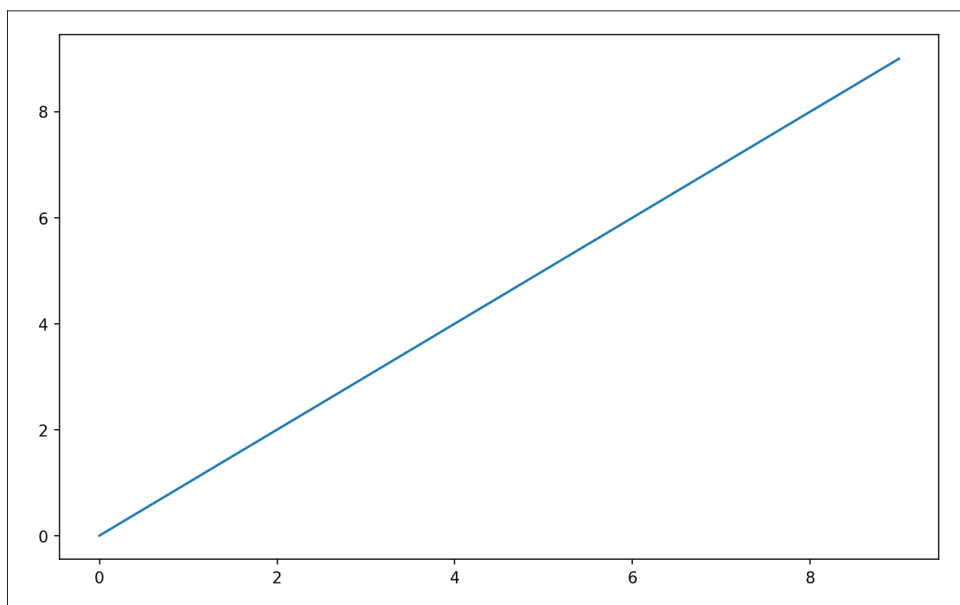


Рис. 9.1. Простой линейный график

Библиотеки типа `seaborn` и встроенные в `pandas` функции построения графиков берут на себя многие рутинные детали, но если предусмотренных в них параметров вам недостаточно, то придется разбираться с API библиотеки `matplotlib`.



В этой книге не хватит места для рассмотрения функциональности `matplotlib` во всей полноте. Но достаточно показать, что делать, и дальше вы все освоите самостоятельно. Для изучения продвинутых возможностей нет ничего лучше галереи и документации `matplotlib`.

Рисунки и подграфики

Графики в matplotlib «живут» внутри объекта рисунка `Figure`. Создать новый рисунок можно методом `plt.figure()`:

```
In [17]: fig = plt.figure()
```

В IPython, если сначала выполнить команду `%matplotlib` для настройки интеграции с matplotlib, появится пустое окно графика, но в Jupyter не появится ничего, пока мы не выполним еще несколько команд.

У команды `plt.figure()` имеется ряд параметров, в частности `figsize` гарантирует, что при сохранении на диске рисунок будет иметь определенные размер и отношение сторон.

Нельзя создать график, имея пустой рисунок. Сначала нужно создать один или несколько подграфиков с помощью метода `add_subplot`:

```
In [18]: ax1 = fig.add_subplot(2, 2, 1)
```

Это означает, что рисунок будет расчерчен сеткой 2×2, и мы выбираем первый из четырех подграфиков (нумерация начинается с 1). Если создать следующие два подграфика, то получится рисунок, изображенный на рис. 9.2.

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

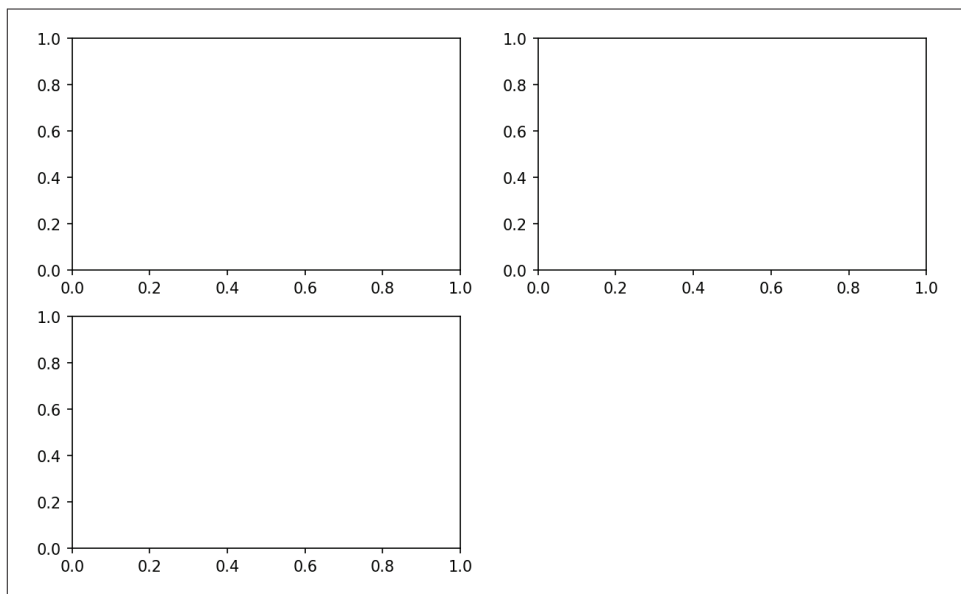


Рис. 9.2. Пустой рисунок matplotlib с тремя подграфиками



Один из нюансов работы с Jupyter-блокнотами заключается в том, что графики сбрасываются в исходное состояние после вычисления каждой ячейки, поэтому если графики достаточно сложны, то следует помещать все команды построения в одну ячейку блокнота.

В примере ниже все команды находятся в одной ячейке:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

Эти объекты осей графика обладают различными методами, которые создают графики различных типов. Рекомендуется использовать именно методы осей, а не верхнеуровневые функции построения типа `plt.plot`. Например, можно было бы построить линейный график методом `plot` (см. рис. 9.3):

```
In [21]: ax3.plot(np.random.standard_normal(50).cumsum(), color="black",
....:             linestyle="dashed")
```

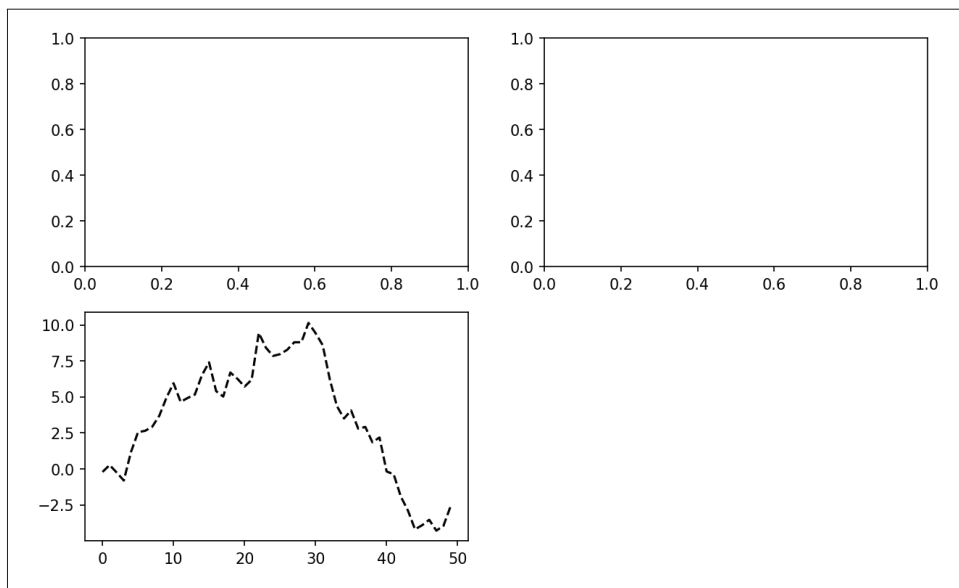


Рис. 9.3. Визуализация данных после построения одного графика

Возможно, вы обратили внимание на сообщение `<matplotlib.lines.Line2D at ...>`, появившееся после выполнения этого метода. `matplotlib` возвращает объекты, ссылающиеся на только что добавленную компоненту графика. Чаще всего результат можно спокойно игнорировать или поставить в конце строки точку с запятой, чтобы подавить вывод.

Дополнительные параметры говорят `matplotlib`, что график должен быть нарисован черной пунктирной линией. Вызов `fig.add_subplot` возвращает объект `AxesSubplot`, который позволяет рисовать в другом пустом подграфике, вызывая его методы экземпляра (см. рис. 9.4):

```
In [22]: ax1.hist(np.random.standard_normal(100), bins=20, color="black", alpha=0.3);
```

```
In [23]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.standard_normal(30));
```

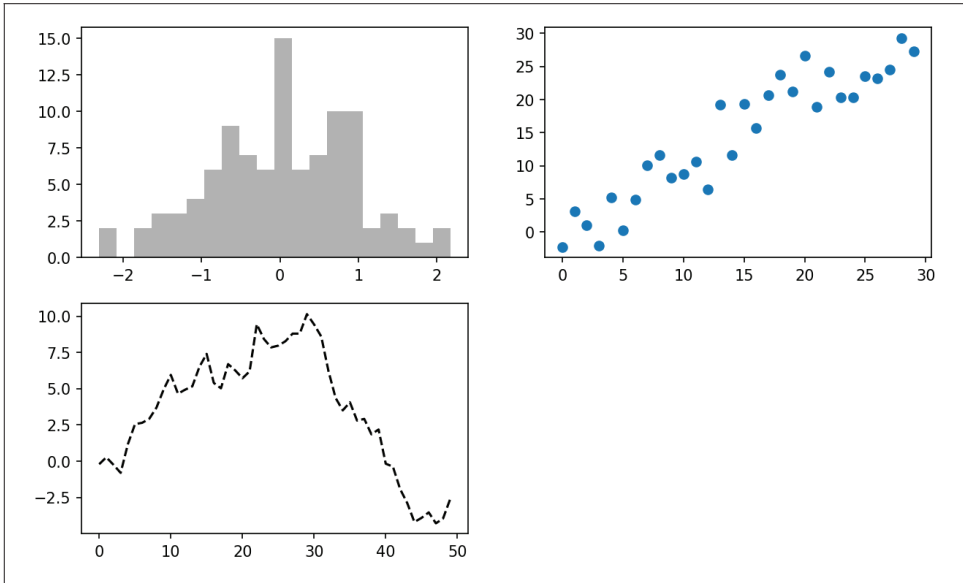


Рис. 9.4. Визуализация данных после построения дополнительных графиков

Параметр стиля `alpha=0.3` задает степень прозрачности наложенного графика.

Полный перечень типов графиков имеется в документации по matplotlib (<https://matplotlib.org/>).

Чтобы сделать создание подграфиков более удобным, matplotlib предоставляет метод `plt.subplots`, который создает новый рисунок и возвращает массив NumPy, содержащий созданные в нем объекты подграфиков:

```
In [25]: fig, axes = plt.subplots(2, 3)

In [26]: axes
Out[26]:
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```

Затем к массиву `axes` можно обращаться как к двумерному массиву, например `axes[0, 1]` ссылается на подграфик во второй позиции верхней строки. Можно также указать, что подграфики должны иметь общую ось *x* или *y*, задав параметры `sharex` и `sharey` соответственно. Это удобно, когда нужно сравнить данные в одном масштабе; иначе matplotlib автоматически и независимо выбирает масштаб графика. Подробнее об этом методе см. табл. 9.1.

Таблица 9.1. Параметры метода `matplotlib.pyplot.subplots`

Аргумент	Описание
<code>nrows</code>	Число строк в сетке подграфиков
<code>ncols</code>	Число столбцов в сетке подграфиков
<code>sharex</code>	Все подграфики должны иметь одинаковые риски на оси X (настройка <code>xlim</code> отражается на всех подграфиках)

Аргумент	Описание
<code>sharey</code>	Все подграфики должны иметь одинаковые риски на оси Y (настройка <code>ylim</code> отражается на всех подграфиках)
<code>subplot_kw</code>	Словарь ключевых слов, передаваемый функции <code>add_subplot</code> при создании каждого подграфика
<code>**fig_kw</code>	Дополнительные ключевые слова используются при создании рисунка, например <code>plt.subplots(2, 2, figsize=(8, 6))</code>

Задание свободного места вокруг подграфиков

По умолчанию `matplotlib` оставляет пустое место вокруг каждого подграфика и между подграфиками. Размер этого места определяется относительно высоты и ширины графика, так что если изменить размер графика программно или вручную (изменив размер окна), то график автоматически перестроится. Величину промежутка легко изменить с помощью метода `subplots_adjust` объекта `Figure`:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

Параметры `wspace` и `hspace` определяют, какой процент от ширины (соответственно высоты) рисунка должен составлять промежуток между подграфиками. В примере ниже я задал нулевой промежуток (рис. 9.5):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.standard_normal(500), bins=50,
                        color="black", alpha=0.5)
fig.subplots_adjust(wspace=0, hspace=0)
```

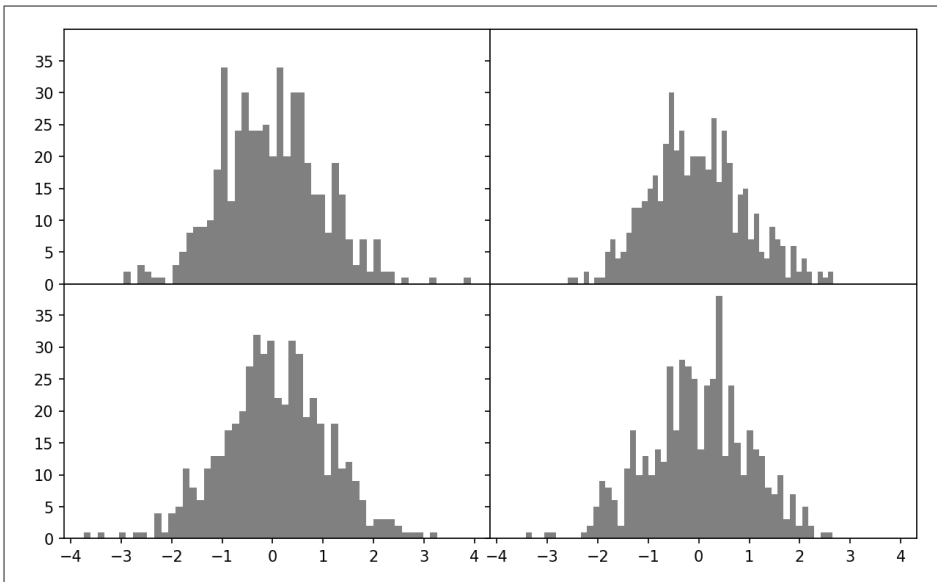


Рис. 9.5. Визуализация данных, в которой подграфики не разделены промежутками

Вы, наверное, обратили внимание, что риски на осях наложились друг на друга. matplotlib это не проверяет, поэтому если такое происходит, то вам придется самостоятельно подкорректировать риски, явно указав их положения и надписи (как это сделать, мы узнаем в следующих разделах).

Цвета, маркеры и стили линий

Функция рисования линии – `plot` – принимает массивы координат x и y , а также необязательные параметры стилизации. Например, чтобы нарисовать график зависимости y от x зеленой штриховой линией, нужно написать:

```
ax.plot(x, y, linestyle="--", color="green")
```

Для наиболее употребительных цветов предоставляются названия, но вообще-то любой цвет можно представить своим шестнадцатеричным кодом (например, `'#CECECE'`). Некоторые поддерживаемые стили линий перечислены в строке документации для функции `plot` (в IPython или Jupyter введите `plot?`). А полный перечень имеется в онлайн-официальной документации.

Линейные графики могут быть также снабжены *маркерами*, обозначающими точки, по которым построен график. Поскольку matplotlib создает непрерывный линейный график, производя интерполяцию между точками, иногда не ясно, где же находятся исходные точки. Маркер можно задать в виде дополнительного параметра стиля (рис. 9.6):

```
In [31]: ax = fig.add_subplot()
```

```
In [32]: ax.plot(np.random.standard_normal(30).cumsum(), color="black",
.....:          linestyle="dashed", marker="o");
```

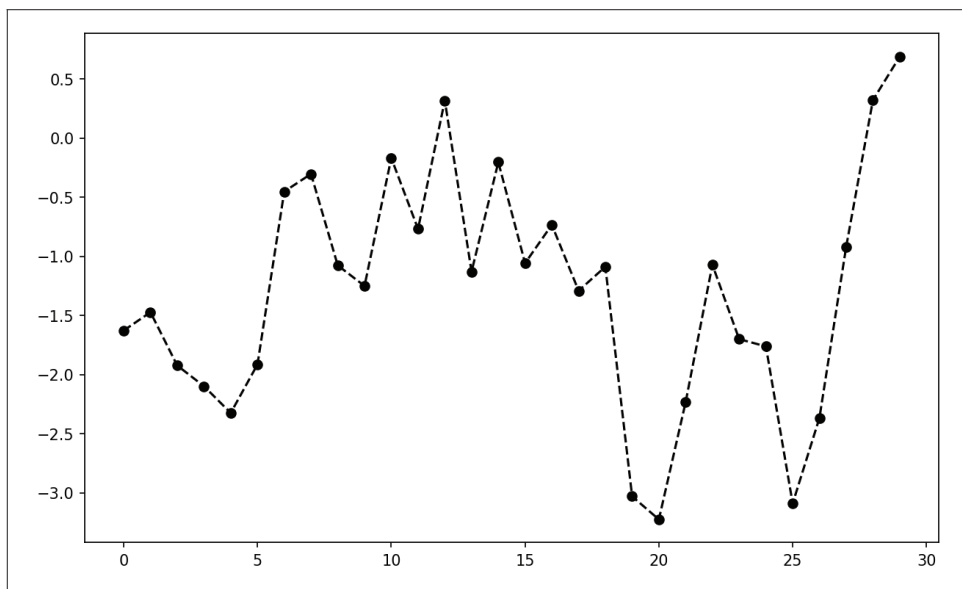


Рис. 9.6. Линейный график с маркерами

По умолчанию на линейных графиках соседние точки соединяются отрезками прямой, т. е. производится линейная интерполяция. Параметр `drawstyle` позволяет изменить этот режим:

```
In [34]: fig = plt.figure()

In [35]: ax = fig.add_subplot()

In [36]: data = np.random.standard_normal(30).cumsum()

In [37]: ax.plot(data, color="black", linestyle="dashed", label="Default");

In [38]: ax.plot(data, color="black", linestyle="dashed",
.....:         drawstyle="steps-post", label="steps-post");

In [39]: ax.legend()
```

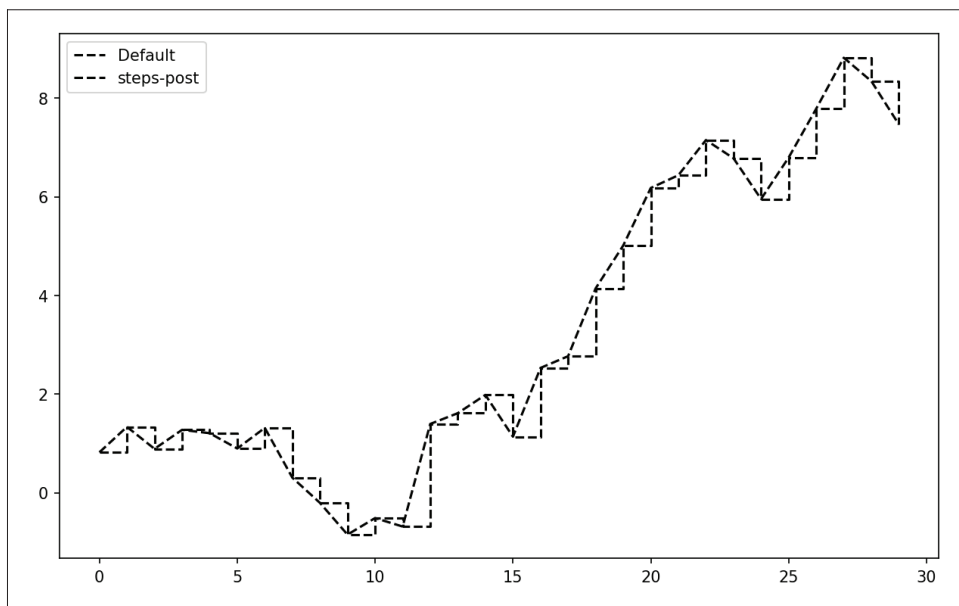


Рис. 9.7. Линейный график с различными значениями параметра `drawstyle`

В данном случае мы передали функции `plot` аргумент `label`, поэтому можем с помощью метода `plt.legend` нанести на график надпись, описывающую каждую линию. Подробнее о надписях мы поговорим ниже.



Для создания надписи необходимо вызвать метод `ax.legend` вне зависимости от того, передавали вы аргумент `label` при построении графика или нет.

Риски, метки и надписи

Для доступа к большинству средств оформления графиков у объектов осей имеются специальные методы, в т. ч. `xlim`, `xticks` и `xticklabels`. Они управляют

размером области, занятой графиком, положением и метками рисок соответственно. Использовать их можно двумя способами:

- при вызове без аргументов возвращается текущее значение параметра. Например, метод `plt.xlim()` возвращает текущий диапазон значений по оси X;
- при вызове с аргументами устанавливается новое значение параметра. Например, в результате вызова `plt.xlim([0, 10])` диапазон значений по оси X устанавливается от 0 до 10.

Все подобные методы действуют на активный или созданный последним объект `AxesSubplot`. Каждому из них соответствуют два метода самого объекта подграфика; в случае `xlim` это методы `ax.get_xlim` и `ax.set_xlim`.

Задание названия графика, названий осей, рисок и их меток

Чтобы проиллюстрировать оформление осей, я создам простой рисунок и в нем график случайного блуждания (рис. 9.8):

```
In [40]: fig, ax = plt.subplots()
```

```
In [41]: ax.plot(np.random.standard_normal(1000).cumsum());
```

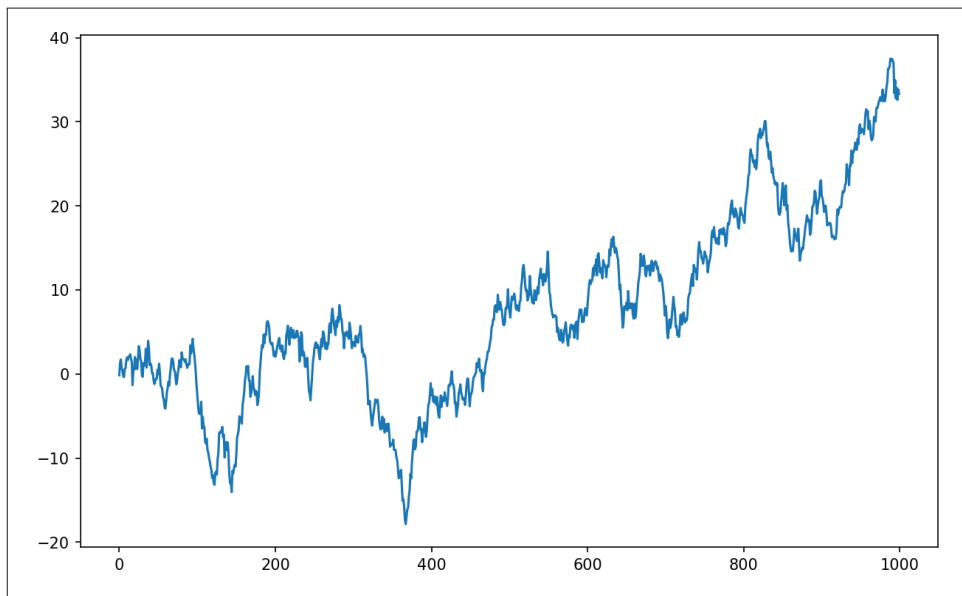


Рис. 9.8. Простой график для иллюстрации рисок (с метками)

Для изменения рисок на оси X проще всего воспользоваться методами `set_xticks` и `set_xticklabels`. Первый говорит matplotlib, где в пределах диапазона значений данных ставить риски; по умолчанию их числовые значения изображаются также и в виде меток. Но можно задать и другие метки с помощью метода `set_xticklabels`:

```
In [42]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])

In [43]: labels = ax.set_xticklabels(["one", "two", "three", "four", "five"],
    ....:                             rotation=30, fontsize=8)
```

Аргумент `rotation` устанавливает угол наклона меток рисок к оси x равным 30 градусам. Наконец, метод `set_xlabel` именуется ось x , а метод `set_title` задает название подграфика (см. окончательный результат на рис. 9.9):

```
In [44]: ax.set_xlabel("Stages")

Out[44]: Text(0.5, 6.666666666666667, 'Stages')
In [45]: ax.set_title("My first matplotlib plot")
```

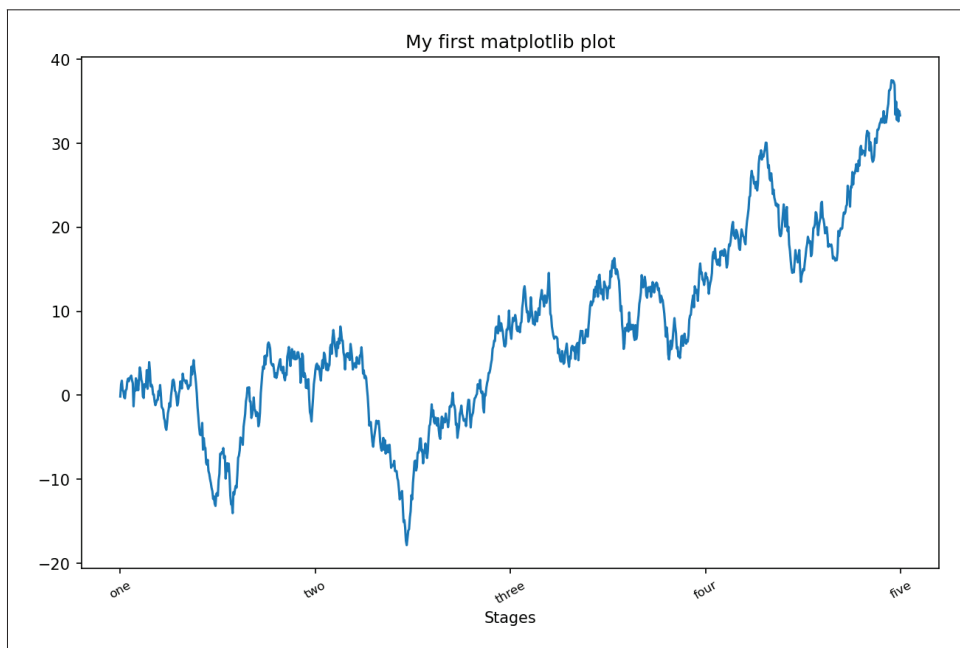


Рис. 9.9. Простой график для иллюстрации рисок

Модификация оси y производится точно так же с заменой x на y . В классе оси имеется метод `set`, позволяющий задавать сразу несколько свойств графика. Так, предыдущий пример можно было записать в следующем виде:

```
ax.set(title="My first matplotlib plot", xlabel="Stages")
```

Добавление пояснительных надписей

Пояснительная надпись – еще один важный элемент оформления графика. Добавить ее можно двумя способами. Проще всего передать аргумент `label` при добавлении каждого нового графика:

```
In [46]: fig, ax = plt.subplots()

In [47]: ax.plot(np.random.randn(1000).cumsum(), color="black", label="one");
```

```
In [48]: ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dashed",
....:          label="two");
In [49]: ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dotted",
....:          label="three");
```

После этого можно вызвать метод `ax.legend()`, и он автоматически создаст пояснительную надпись. Получившийся график показан на рис. 9.10:

```
In [50]: ax.legend()
```

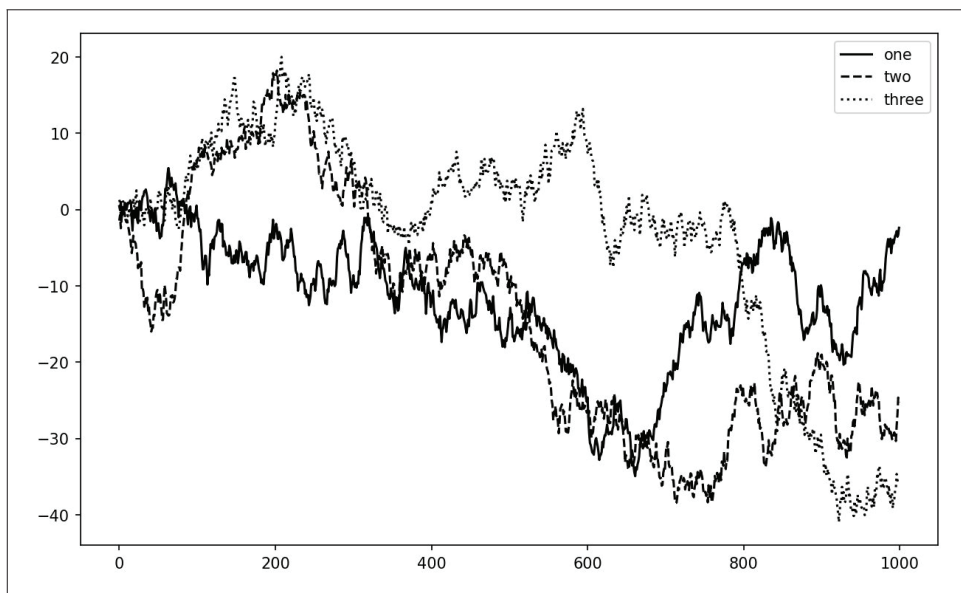


Рис. 9.10. Простой график с тремя линиями и пояснительной надписью

Аргумент `loc` метода `legend` может принимать и другие значения. Дополнительные сведения см. в строке документации (введите `ax.legend?`).

Аргумент `loc` говорит, где поместить надпись. По умолчанию подразумевается значение `'best'`, в этом случае место выбирается так, чтобы по возможности не загромождать сам график. Чтобы исключить из надписи один или несколько элементов, не задавайте параметр `label` вовсе или задайте `label='_nolegend_'`.

Аннотации и рисование в подграфике

Помимо стандартных типов графиков, разрешается наносить на график свои аннотации, которые могут включать текст, стрелки и другие фигуры. Для добавления аннотаций и текста предназначены функции `text`, `arrow` и `annotate`. Функция `text` наносит на график текст, начиная с точки с заданными координатами (x, y) , с факультативной стилизацией:

```
ax.text(x, y, "Hello world!",
       family="monospace", fontsize=10)
```

В аннотациях могут встречаться текст и стрелки. В качестве примера построим график цен закрытия по индексу S&P 500 начиная с 2007 года (данные получены

с сайта Yahoo! Finance) и аннотируем его некоторыми важными датами, относящимися к финансовому кризису 2008–2009 годов. Воспроизвести этот код можно, введя его в одну ячейку Jupyter-блокнота. Результат изображен на рис. 9.11.

```
from datetime import datetime

fig, ax = plt.subplots()

data = pd.read_csv("examples/spx.csv", index_col=0, parse_dates=True)
spx = data["SPX"]

spx.plot(ax=ax, color="black")

crisis_data = [
    (datetime(2007, 10, 11), "Peak of bull market"),
    (datetime(2008, 3, 12), "Bear Stearns Fails"),
    (datetime(2008, 9, 15), "Lehman Bankruptcy")
]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor="black", headwidth=4, width=2,
                                headlength=4),
                horizontalalignment="left", verticalalignment="top")

# Приблизить годы 2007-2010
ax.set_xlim(["1/1/2007", "1/1/2011"])
ax.set_ylim([600, 1800])

ax.set_title("Important dates in the 2008-2009 financial crisis")
```

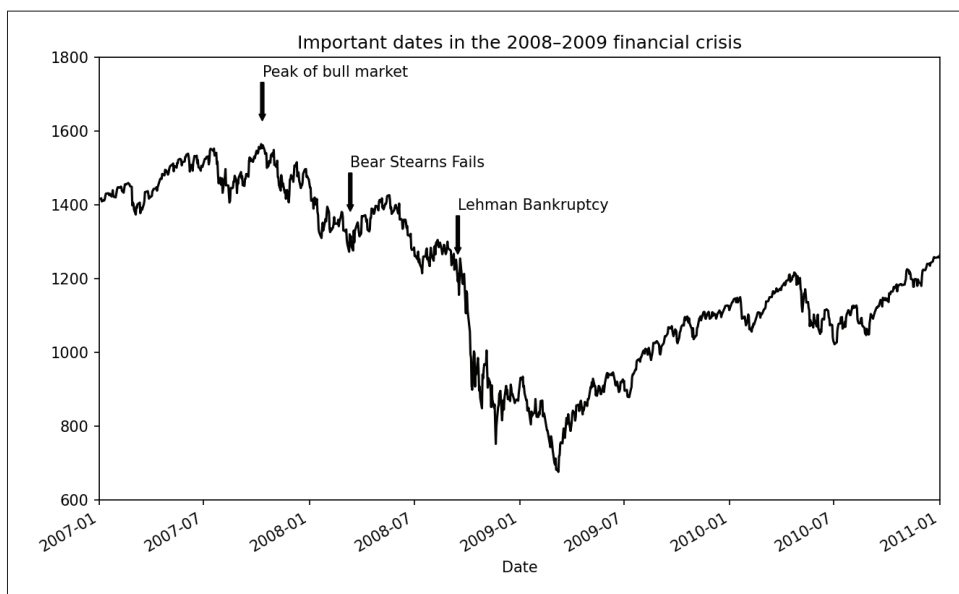


Рис. 9.11. Важные даты, относящиеся к финансовому кризису 2008–2009 годов

Отметим несколько важных моментов. Метод `ax.annotate` умеет рисовать метку в позиции, заданной координатами x и y . Мы воспользовались методами `set_xlim` и `set_ylim`, чтобы вручную задать нижнюю и верхнюю границы графика, а не полагаться на умолчания matplotlib. Наконец, метод `ax.set_title` задает название графика в целом.

В галерее matplotlib в сети есть много других поучительных примеров аннотаций.

Для рисования фигур требуется больше усилий. В matplotlib имеются объекты, соответствующие многим стандартным фигурам, они называются *патчами* (patches). Часть из них, например `Rectangle`, `Circle` и `Polygon`, находится в модуле `matplotlib.pyplot`, а весь набор – в модуле `matplotlib.patches`.

Чтобы поместить на график фигуру, мы создаем объект патча `shp` и добавляем его в подграфик, вызывая метод `ax.add_patch(shp)` (см. рис. 9.12):

```
fig, ax = plt.subplots()

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color="black", alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color="blue", alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                    color="green", alpha=0.5)
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon))
```

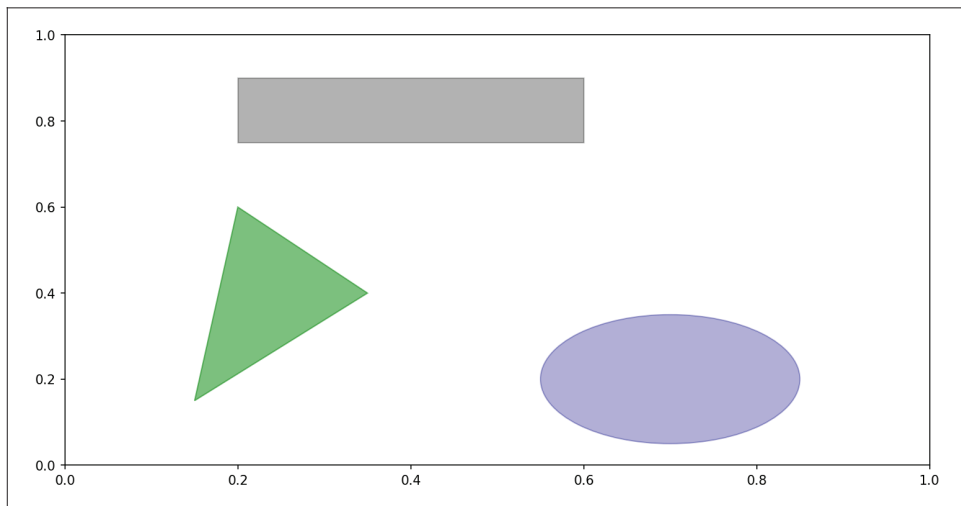


Рис. 9.12. Визуализация данных, составленная из трех разных патчей

Заглянув в код многих знакомых типов графиков, вы увидите, что они составлены из патчей.

Сохранение графиков в файле

Активный рисунок можно сохранить в файле методом экземпляра рисунка `savefig`. Например, чтобы сохранить рисунок в формате SVG, достаточно указать только имя файла:

```
fig.savefig("figpath.svg")
```

Формат выводится из расширения имени файла. Если бы мы задали файл с расширением `.pdf`, то рисунок был бы сохранен в формате PDF. При публикации графики я часто использую параметр `dpi` – разрешение в точках на дюйм. Чтобы получить тот же самый график в формате PNG с разрешением 400 DPI, нужно было бы написать:

```
fig.savefig("figpath.png", dpi=400)
```

В табл. 9.2 перечислены некоторые другие аргументы метода `savefig`. Полный список см. в строке документации в IPython или Jupyter.

Таблица 9.2. Аргументы метода `Figure.savefig`

Аргумент	Описание
<code>fname</code>	Строка, содержащая путь к файлу или файлоподобному объекту Python. Формат рисунка определяется по расширению имени файла, например PDF для <code>.pdf</code> и PNG для <code>.png</code>
<code>dpi</code>	Разрешение рисунка в точках на дюйм; по умолчанию 100 в IPython и 72 в Jupiter, но может настраиваться
<code>facecolor</code> , <code>edgecolor</code>	Цвет фона рисунка вне области, занятой подграфиками. По умолчанию <code>'w'</code> (белый)
<code>format</code>	Явно заданный формат файла (<code>'png'</code> , <code>'pdf'</code> , <code>'svg'</code> , <code>'ps'</code> , <code>'eps'</code> и т. д.)

Конфигурирование matplotlib

В начальной конфигурации `matplotlib` заданы цветовые схемы и умолчания, ориентированные главным образом на подготовку рисунков к публикации. По счастью, почти все аспекты поведения по умолчанию можно сконфигурировать с помощью обширного набора глобальных параметров, определяющих размер рисунка, промежутки между подграфиками, цвета, размеры шрифтов, стили сетки и т. д. Есть два основных способа работы с системой конфигурирования `matplotlib`. Первый – программный, с помощью метода `rc`. Например, чтобы глобально задать размер рисунка равным 10×10 , нужно написать:

```
plt.rc("figure", figsize=(10, 10))
```

Все текущие конфигурационные параметры хранятся в словаре `plt.rcParams`, для восстановления значения по умолчанию следует вызвать функцию `plt.rcParamsdefaults()`.

Первый аргумент `rc` – настраиваемый компонент, например `'figure'`, `'axes'`, `'xtick'`, `'ytick'`, `'grid'`, `'legend'` и т. д. Вслед за ним идут именованные аргументы, задающие параметры этого компонента. В программе описывать параметры проще всего в виде словаря:

```
plt.rc("font", family="monospace", weight="bold", size=8)
```

Если требуется более тщательная настройка, то можно воспользоваться входящим в состав `matplotlib` конфигурационным файлом `matplotlibrc` в каталоге `matplotlib/mpl-data`, где перечислены все параметры. Если вы настроите этот файл

и поместите его в свой домашний каталог под именем `.matplotlibrc`, то он будет загружаться при каждом использовании `matplotlib`.

В следующем разделе мы увидим, что в пакете `seaborn` имеется несколько встроенных тем, или *стилей*, настроенных над конфигурационной системой `matplotlib`.

9.2. ПОСТРОЕНИЕ ГРАФИКОВ С ПОМОЩЬЮ PANDAS И SEABORN

Библиотека `matplotlib` – средство довольно низкого уровня. График собирается из базовых компонентов: способ отображения данных (тип графика: линейный график, столбчатая диаграмма, коробчатая диаграмма, диаграмма рассеяния, контурный график и т. д.), пояснительная надпись, название, метки рисок и прочие аннотации).

В библиотеке `pandas` может быть несколько столбцов данных, а с ними метки строк и метки столбцов. В саму `pandas` встроены методы построения, упрощающие создание визуализаций объектов `DataFrame` и `Series`. Существует также высокоуровневая библиотека `seaborn` для создания статистических графиков, основанная на `matplotlib`. `Seaborn` упрощает создание многих типов визуализаций.

Линейные графики

У объектов `Series` и `DataFrame` имеется метод `plot`, который умеет строить графики разных типов. По умолчанию он строит линейные графики (см. рис. 9.13):

```
In [61]: s = pd.Series(np.random.standard_normal(10).cumsum(), index=np.arange(0, 100, 10))
```

```
In [62]: s.plot()
```

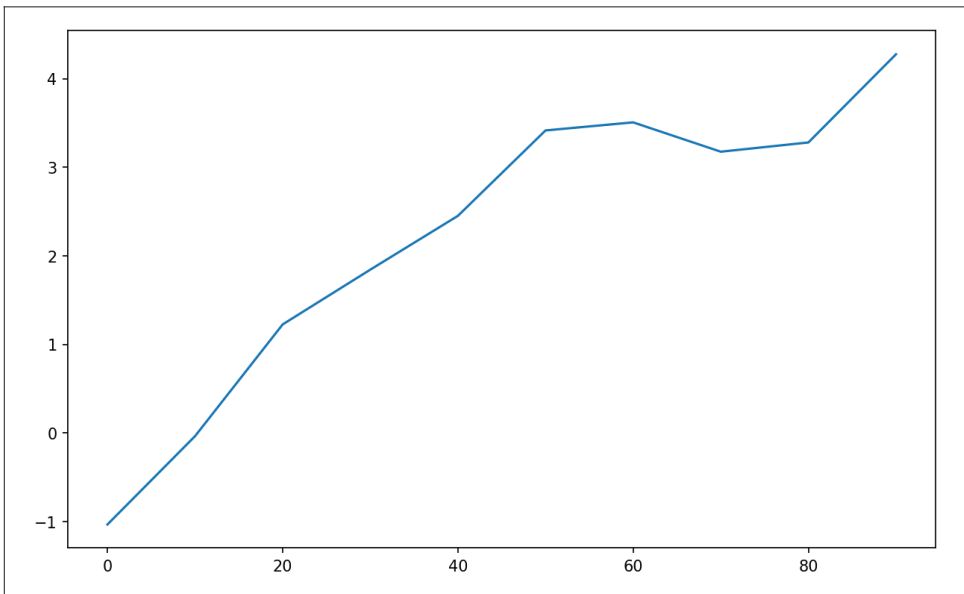


Рис. 9.13. Простой пример графика для объекта `Series`

Индекс объекта Series передается matplotlib для нанесения рисок на ось *x*, но это можно отключить, задав параметр `use_index=False`. Риски и диапазон значений на оси *x* можно настраивать с помощью параметров `xticks` и `xlim`, а на оси *y* – с помощью параметров `yticks` и `ylim`. Полный перечень параметров метода `plot` приведен в табл. 9.3. О некоторых я расскажу в этом разделе, а остальные оставлю вам для самостоятельного изучения.

Таблица 9.3. Параметры метода Series.plot

Аргумент	Описание
<code>label</code>	Метка для пояснительной надписи на графике
<code>ax</code>	Объект подграфика matplotlib, внутри которого строим график. Если параметр не задан, то используется активный подграфик
<code>style</code>	Строка стиля, например <code>'ko--'</code> , которая передается matplotlib
<code>alpha</code>	Уровень непрозрачности графика (число от 0 до 1)
<code>kind</code>	Может принимать значения <code>"area"</code> , <code>"bar"</code> , <code>"barh"</code> , <code>"density"</code> , <code>"hist"</code> , <code>"kde"</code> , <code>"line"</code> , <code>"pie"</code> ; по умолчанию <code>"line"</code>
<code>figsize</code>	Размер создаваемого объекта рисунка
<code>logx</code>	<code>True</code> означает, что нужно использовать логарифмический масштаб по оси <i>x</i> , а <code>"sym"</code> – симметрический логарифм, допускающий отрицательные значения
<code>logy</code>	<code>True</code> означает, что нужно использовать логарифмический масштаб по оси <i>y</i> , а <code>"sym"</code> – симметрический логарифм, допускающий отрицательные значения
<code>title</code>	Заголовок графика
<code>use_index</code>	Брать метки рисок из индекса объекта
<code>rot</code>	Угол поворота меток рисок (от 0 до 360)
<code>xticks</code>	Значения рисок на оси <i>x</i>
<code>yticks</code>	Значения рисок на оси <i>y</i>
<code>xlim</code>	Границы по оси <i>x</i> (например, <code>[0,10]</code>)
<code>ylim</code>	Границы по оси <i>y</i>
<code>grid</code>	Отображать координатную сетку (по умолчанию включено)

Большинство методов построения графиков в pandas принимают необязательный параметр `ax` – объект подграфика matplotlib. Это позволяет гибко расположить подграфики в сетке.

Метод `plot` объекта DataFrame строит отдельные графики каждого столбца внутри одного подграфика и автоматически создает пояснительную надпись (см. рис. 9.14).

```
In [63]: df = pd.DataFrame(np.random.standard_normal((10, 4)).cumsum(0),
.....:                    columns=["A", "B", "C", "D"],
.....:                    index=np.arange(0, 100, 10))

In [64]: plt.style.use('grayscale')

In [65]: df.plot()
```

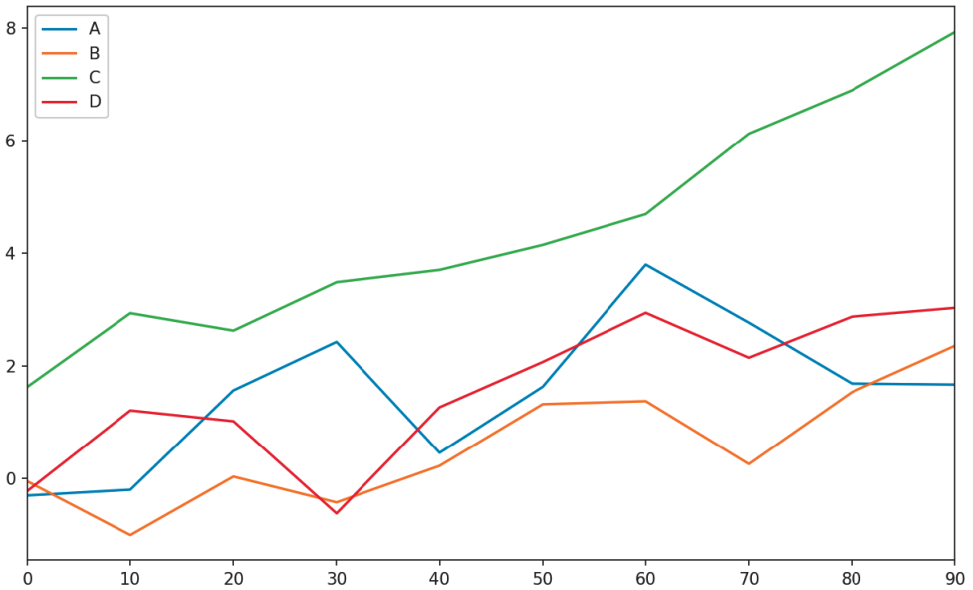


Рис. 9.14. Простой пример графика для объекта DataFrame



Здесь я воспользовался методом `plt.style.use('grayscale')`, чтобы переключиться на цветовую схему, более подходящую для черно-белой печати, поскольку не все читатели смогут увидеть полноцветные графики.

Атрибут `plot` содержит «семейство» методов для различных типов графиков. Например, `df.plot()` эквивалентно `df.plot.line()`. Некоторые из этих методов мы рассмотрим ниже.



Дополнительные именованные аргументы метода `plot` без изменения передаются соответствующей функции matplotlib, поэтому, внимательно изучив API matplotlib, вы сможете настраивать графики более точно.

У объекта DataFrame есть ряд параметров, которые гибко описывают обработку столбцов, например нужно ли строить их графики внутри одного и того же или разных подграфиков. Все они перечислены в табл. 9.4.

Таблица 9.4. Параметры метода DataFrame.plot

Аргумент	Описание
<code>subplots</code>	Рисовать график каждого столбца DataFrame в отдельном подграфике
<code>layouts</code>	2-кортеж (строки, столбцы), определяющий расположение подграфиков
<code>sharex</code>	Если <code>subplots=True</code> , то совместно использовать ось X, объединяя риски и границы
<code>sharey</code>	Если <code>subplots=True</code> , то совместно использовать ось Y
<code>legend</code>	Помещать в подграфик пояснительную надпись (по умолчанию <code>True</code>)
<code>sort_columns</code>	Строить графики столбцов в алфавитном порядке; по умолчанию используется существующий порядок столбцов



О построении графиков временных рядов см. главу 11.

Столбчатые диаграммы

Методы `plot.bar()` и `plot.barh()` строят соответственно вертикальную и горизонтальную столбчатые диаграммы. В этом случае индекс Series или DataFrame будет использоваться для нанесения рисок на ось *x* (`bar`) или *y* (`barh`) (см. рис. 9.15):

```
In [66]: fig, axes = plt.subplots(2, 1)
```

```
In [67]: data = pd.Series(np.random.uniform(size=16), index=list("abcdefghijklmnop"))
```

```
In [68]: data.plot.bar(ax=axes[0], color="black", alpha=0.7)
```

```
Out[68]: <AxesSubplot:>
```

```
In [69]: data.plot.barh(ax=axes[1], color="black", alpha=0.7)
```

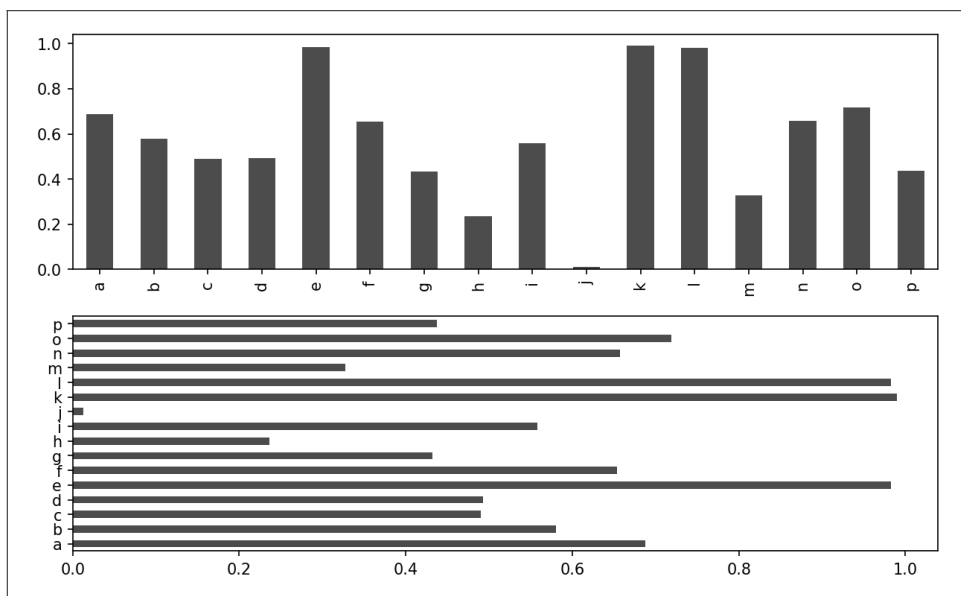


Рис. 9.15. Примеры горизонтальной и вертикальной столбчатых диаграмм

В случае DataFrame значения из каждой строки объединяются в группы столбиков, расположенные поодаль друг от друга. См. рис. 9.16.

```
In [71]: df = pd.DataFrame(np.random.uniform(size=(6, 4)),
.....: index=["one", "two", "three", "four", "five", "six"],
.....: columns=pd.Index(["A", "B", "C", "D"], name="Genus"))
```

```
In [72]: df
Out[72]:
```

Genus	A	B	C	D
one	0.370670	0.602792	0.229159	0.486744
two	0.420082	0.571653	0.049024	0.880592
three	0.814568	0.277160	0.880316	0.431326
four	0.374020	0.899420	0.460304	0.100843
five	0.433270	0.125107	0.494675	0.961825
six	0.601648	0.478576	0.205690	0.560547

```
In [73]: df.plot.bar()
```

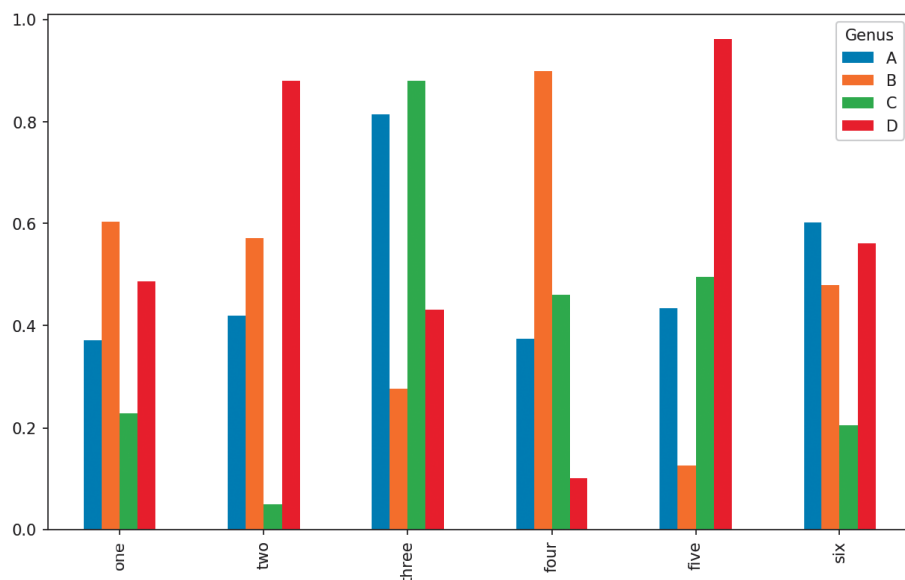


Рис. 9.16. Столбчатая диаграмма для DataFrame

Обратите внимание, что название столбцов DataFrame – «Genus» – используется в заголовке пояснительной надписи.

Для построения составной столбчатой диаграммы по объекту DataFrame нужно задать параметр `stacked=True`, тогда столбики, соответствующие значению в каждой строке, будут приставлены друг к другу (рис. 9.17):

```
In [75]: df.plot.barh(stacked=True, alpha=0.5)
```

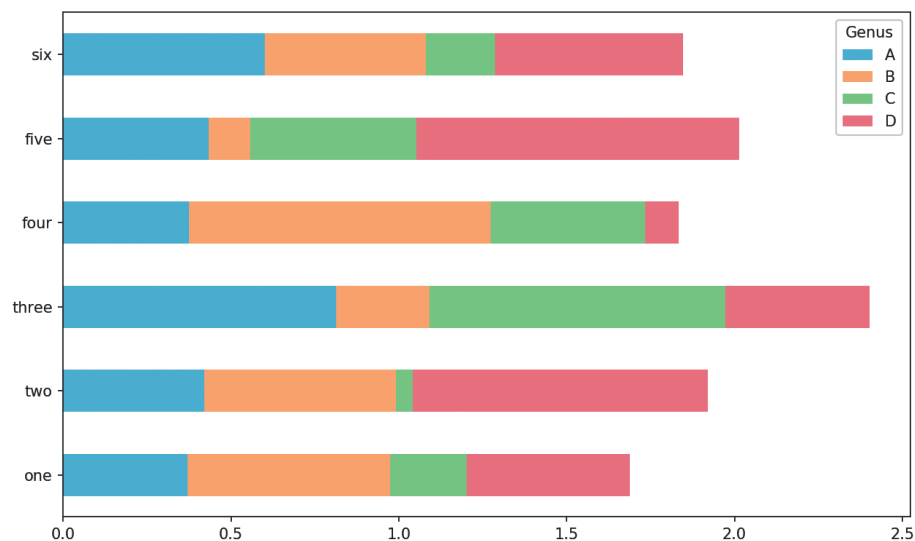


Рис. 9.17. Составная столбчатая диаграмма для DataFrame



Столбчатые диаграммы полезны для визуализации частоты значений в объекте Series с применением метода `value_counts`:
`s.value_counts().plot.bar()`.

Рассмотрим набор данных о чаевых в ресторане. Допустим, мы хотим построить составную столбчатую диаграмму, показывающую процентную долю данных, относящихся к каждому значению количества гостей в группе за каждый день. Я загружаю данные методом `read_csv` и выполняю кросс-табуляцию по дню и количеству гостей. Для вычисления простой таблицы частот по двум столбцам DataFrame удобна функция `pandas.crosstab`:

```
In [77]: tips = pd.read_csv("examples/tips.csv")

In [78]: tips.head()
Out[78]:
```

	total_bill	tip	smoker	day	time	size
0	16.99	1.01	No	Sun	Dinner	2
1	10.34	1.66	No	Sun	Dinner	3
2	21.01	3.50	No	Sun	Dinner	3
3	23.68	3.31	No	Sun	Dinner	2
4	24.59	3.61	No	Sun	Dinner	4

```
In [79]: party_counts = pd.crosstab(tips["day"], tips["size"])

In [80]: party_counts = party_counts.reindex(index=["Thur", "Fri", "Sat", "Sun"])

In [81]: party_counts
Out[81]:
```

size	1	2	3	4	5	6
day						
Thur	1	48	4	5	1	3
Fri	1	16	1	1	0	0
Sat	2	53	18	13	1	0
Sun	0	39	15	18	3	1

Поскольку групп, состоящих из одного или шести гостей, мало, я их удаляю:

```
In [82]: party_counts = party_counts.loc[:, 2:5]
```

Затем нормирую значения, так чтобы сумма в каждой строке была равна 1, и строю график (рис. 9.18):

```
# Нормировка на сумму 1
In [83]: party_pcts = party_counts.div(party_counts.sum(axis="columns"),
....:                                  axis="index")

In [84]: party_pcts
Out[84]:
```

size	2	3	4	5
day				
Thur	0.827586	0.068966	0.086207	0.017241
Fri	0.888889	0.055556	0.055556	0.000000
Sat	0.623529	0.211765	0.152941	0.011765
Sun	0.520000	0.200000	0.240000	0.040000

```
In [85]: party_pcts.plot.bar(stacked=True)
```

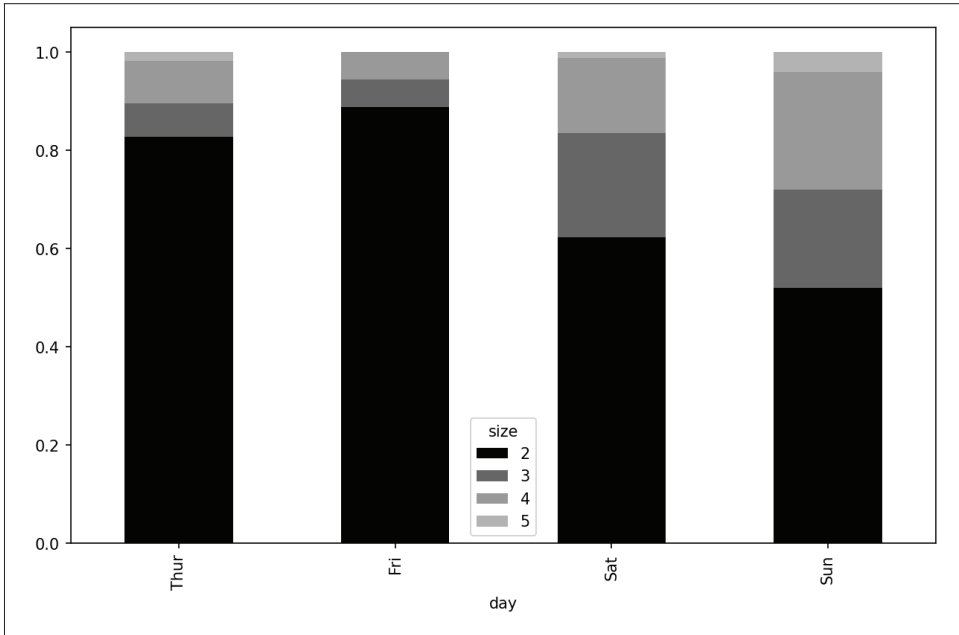


Рис. 9.18. Распределение по количеству гостей в группе за каждый день недели

Как видим, в выходные количество гостей в одной группе увеличивается.

Если перед построением графика данные необходимо как-то агрегировать, то пакет `seaborn` может существенно упростить жизнь (установите пакет командой `install seaborn`). Посмотрим, как с помощью `seaborn` посчитать процент чаевых в зависимости от дня (результат показан на рис. 9.19):

```
In [87]: import seaborn as sns
```

```
In [88]: tips["tip_pct"] = tips["tip"] / (tips["total_bill"] - tips["tip"])
```

```
In [89]: tips.head()
```

```
Out[89]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.063204
1	10.34	1.66	No	Sun	Dinner	3	0.191244
2	21.01	3.50	No	Sun	Dinner	3	0.199886
3	23.68	3.31	No	Sun	Dinner	2	0.162494
4	24.59	3.61	No	Sun	Dinner	4	0.172069

```
In [90]: sns.barplot(x="tip_pct", y="day", data=tips, orient="h")
```

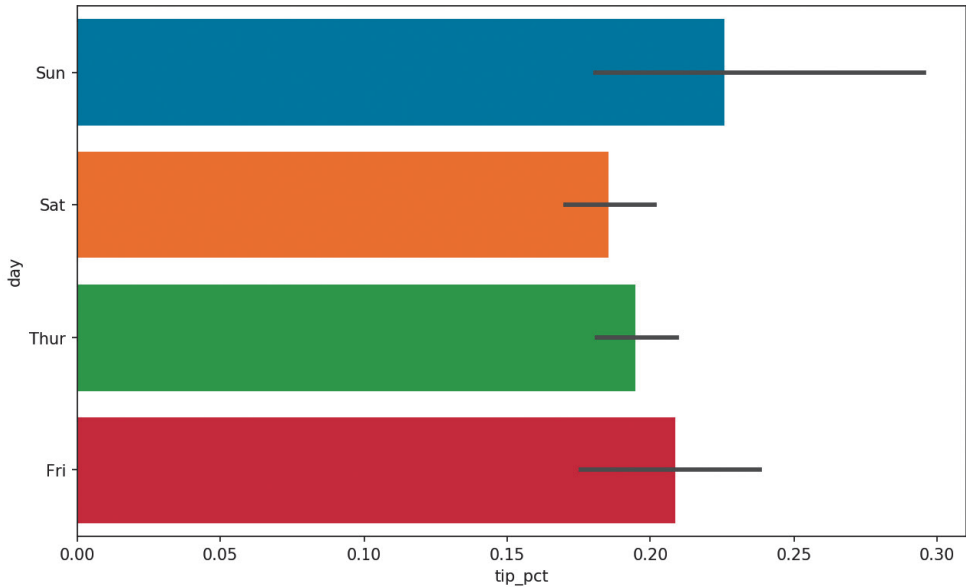


Рис. 9.19. Процент чаевых в зависимости от дня с доверительными интервалами

Функции построения графиков из библиотеки `seaborn` принимают аргумент `data`, в роли которого может выступать объект `pandas DataFrame`. Остальные аргументы относятся к именам столбцов. Поскольку для каждого значения в `day` имеется несколько наблюдений, столбики отражают среднее значение `tip_pct`. Черные линии поверх столбиков представляют 95-процентные доверительные интервалы (эту величину можно настроить, задав дополнительные аргументы).

Функция `seaborn.barplot` принимает аргумент `hue`, позволяющий произвести разбиение по дополнительному дискретному значению (рис. 9.20):

```
In [92]: sns.barplot(x="tip_pct", y="day", hue="time", data=tips, orient="h")
```

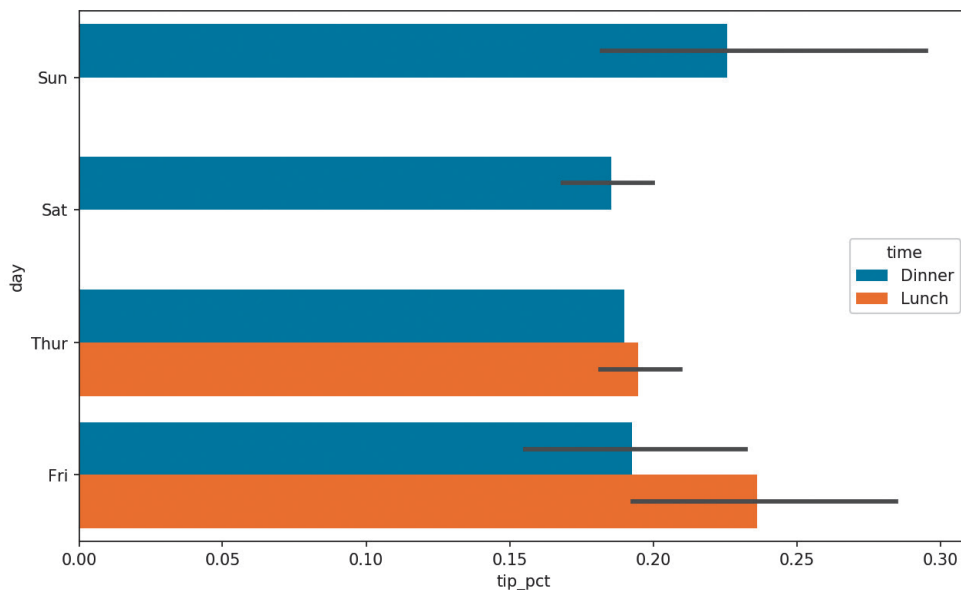



Рис. 9.20. Процент чаевых в зависимости от дня и времени суток

Обратите внимание, что `seaborn` автоматически изменила внешний вид диаграмм: цветовую палитру по умолчанию, цвет фона и цвета линий сетки. Менять внешний вид графиков позволяет функция `seaborn.set_style`:

```
In [94]: sns.set_style("whitegrid")
```

При построении графиков для черно-белой печати полезно задавать полутоновую палитру:

```
sns.set_palette("Greys_r")
```

Гистограммы и графики плотности

Гистограмма – это разновидность столбчатой диаграммы, показывающая дискретизированное представление частоты. Результаты измерений распределяются по дискретным интервалам равной ширины, а на гистограмме отображается количество точек в каждом интервале. На примере приведенных выше данных о чаевых от гостей ресторана мы можем с помощью метода `hist` объекта `Series` построить гистограмму распределения процента чаевых от общей суммы счета (рис. 9.21):

```
In [96]: tips["tip_pct"].plot.hist(bins=50)
```

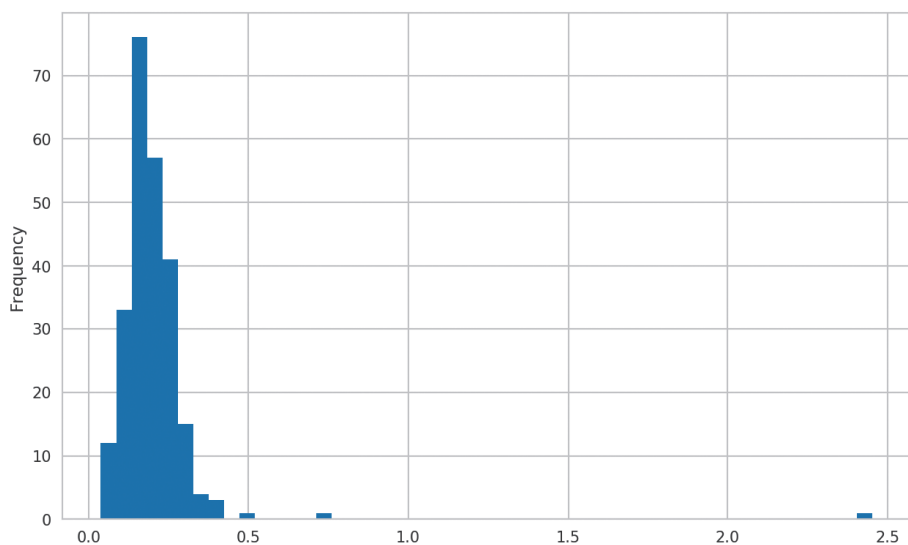


Рис. 9.21. Гистограмма процента чаевых

С гистограммой тесно связан *график плотности*, который строится на основе оценки непрерывного распределения вероятности по результатам измерений. Обычно стремятся аппроксимировать это распределение комбинацией ядер, т. е. более простых распределений, например нормального (гауссова). Поэтому графики плотности еще называют графиками ядерной оценки плотности (KDE – kernel density estimate). Функция `plot` с параметром `kind='kde'` строит график плотности, применяя стандартный метод комбинирования нормальных распределений (рис. 9.22):

```
In [98]: tips["tip_pct"].plot.density()
```

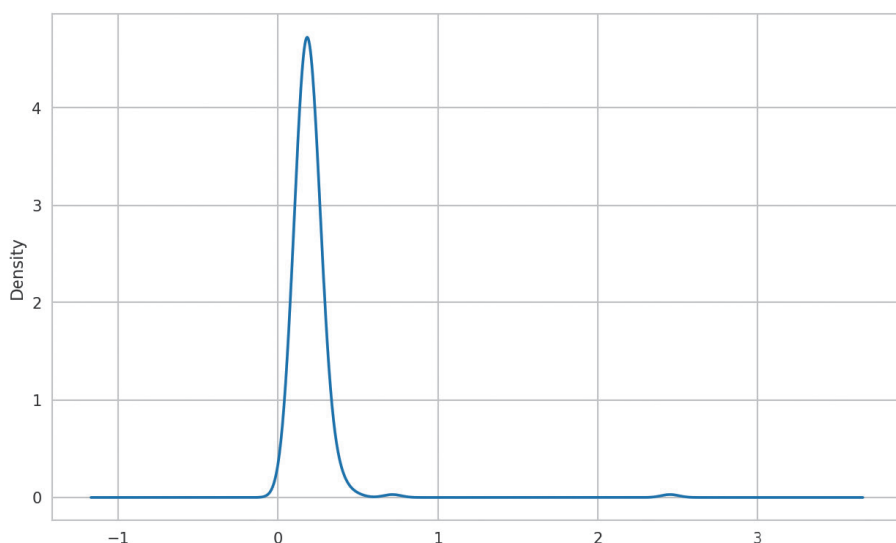


Рис. 9.22. График плотности процента чаевых

Для построения таких графиков необходима библиотека SciPy. Если вы не установили ее раньше, сделайте это сейчас командой

```
conda install scipy
```

Seaborn еще упрощает построение гистограмм и графиков плотности благодаря методу `histplot`, который может строить одновременно гистограмму и непрерывную оценку плотности. В качестве примера рассмотрим бимодальное распределение, содержащее выборки из двух разных стандартных нормальных распределений (рис. 9.23):

```
In [100]: comp1 = np.random.standard_normal(200)
In [101]: comp2 = 10 + 2 * np.random.standard_normal(200)
In [102]: values = pd.Series(np.concatenate([comp1, comp2]))
In [103]: sns.histplot(values, bins=100, color="black")
```

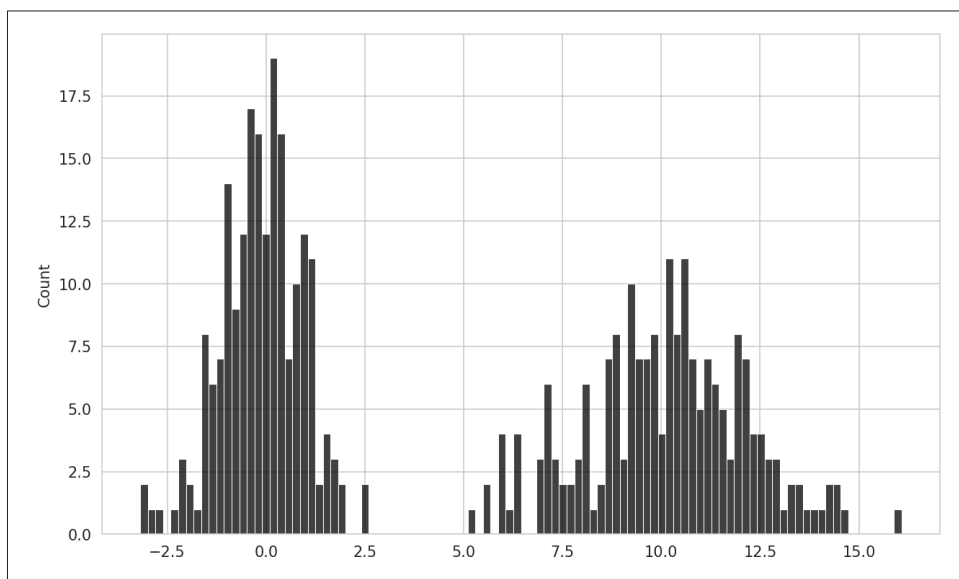


Рис. 9.23. Нормированная гистограмма смеси нормальных распределений

Диаграммы рассеяния

Диаграмма рассеяния, или точечная диаграмма, – полезный способ исследования соотношения между двумя одномерными рядами данных. Для демонстрации я загрузил набор данных `macrodata` из проекта `statsmodels`, выбрал несколько переменных и вычислил логарифмические разности:

```
In [104]: macro = pd.read_csv("examples/macrodata.csv")
In [105]: data = macro[["cpi", "m1", "tbilrate", "unemp"]]
```

```
In [106]: trans_data = np.log(data).diff().dropna()
```

```
In [107]: trans_data.tail()
```

```
Out[107]:
```

	cpi	m1	tbilrate	unemp
198	-0.007904	0.045361	-0.396881	0.105361
199	-0.021979	0.066753	-2.277267	0.139762
200	0.002340	0.010286	0.606136	0.160343
201	0.008419	0.037461	-0.200671	0.127339
202	0.008894	0.012202	-0.405465	0.042560

Затем мы можем использовать метод `regplot` из библиотеки `seaborn`, чтобы построить диаграмму рассеяния и аппроксимирующую ее прямую линейной регрессии (рис. 9.24):

```
In [109]: ax = sns.regplot(x="m1", y="unemp", data=trans_data)
```

```
In [110]: ax.title("Changes in log(m1) versus log(unemp)")
```

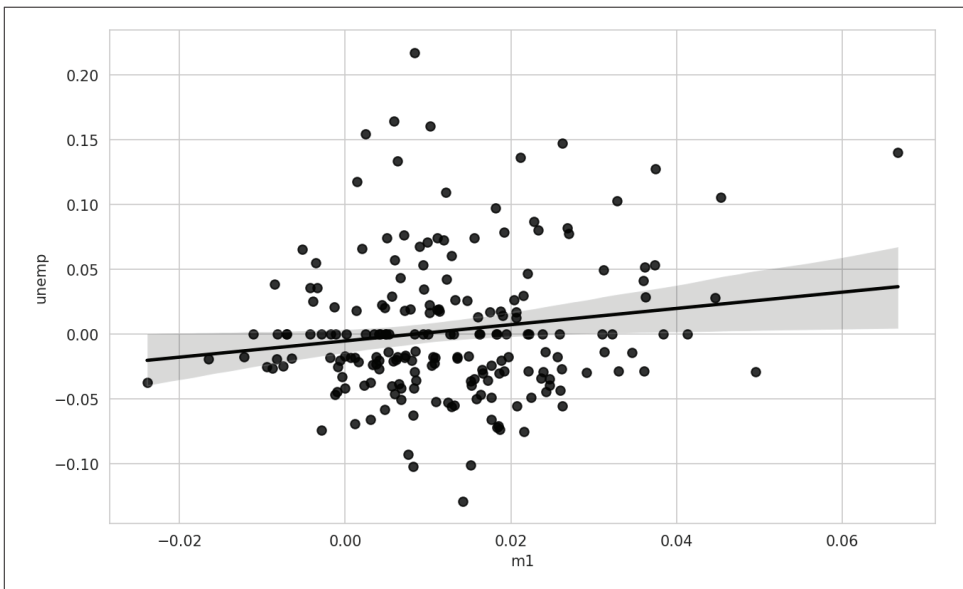


Рис. 9.24. Диаграмма рассеяния с прямой регрессии, построенная средствами `seaborn`

В разведочном анализе данных полезно видеть все диаграммы рассеяния для группы переменных; это называется *диаграммой пар*, или *матрицей диаграмм рассеяния*. Построение такого графика с нуля – довольно утомительное занятие, поэтому в `seaborn` имеется функция `pairplot`, которая поддерживает размещение гистограмм или графиков оценки плотности каждой переменной вдоль диагонали (см. результирующий график на рис. 9.25):

```
In [111]: sns.pairplot(trans_data, diag_kind="kde", plot_kws={"alpha": 0.2})
```

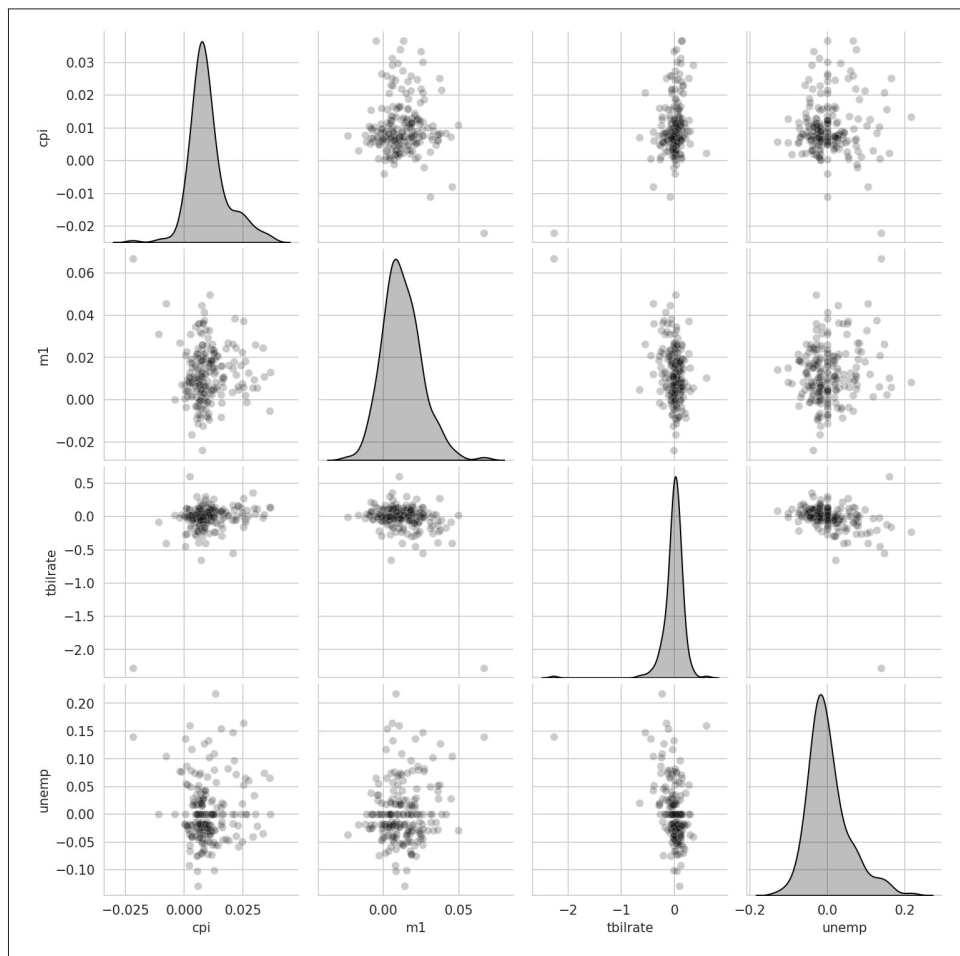


Рис. 9.25. Матрица диаграмм рассеяния для набора данных macrodata из проекта statsmodels

Обратите внимание на аргумент `plot_kws`. Он позволяет передавать конфигурационные параметры отдельным вызовам построения во внедиагональных элементах. Дополнительные сведения о конфигурационных параметрах см. в строке документации [seaborn.pairplot](#).

Фасетные сетки и категориальные данные

Как быть с наборами данных, в которых имеются дополнительные группировочные измерения? Один из способов визуализировать данные с большим числом категориальных переменных – воспользоваться *фасетной сеткой*, т. е. двумерной сеткой графиков, в которой данные распределены между графиками по каждой оси, исходя из различных значений некоторой дискретной переменной. В seaborn имеется полезная функция `catplot`, которая упрощает построение разнообразных фасетных графиков, определяемых категориальными переменными (результатирующий график см. на рис. 9.26):

```
In [112]: sns.catplot(x="day", y="tip_pct", hue="time", col="smoker",
.....:               kind="bar", data=tips[tips.tip_pct < 1])
```

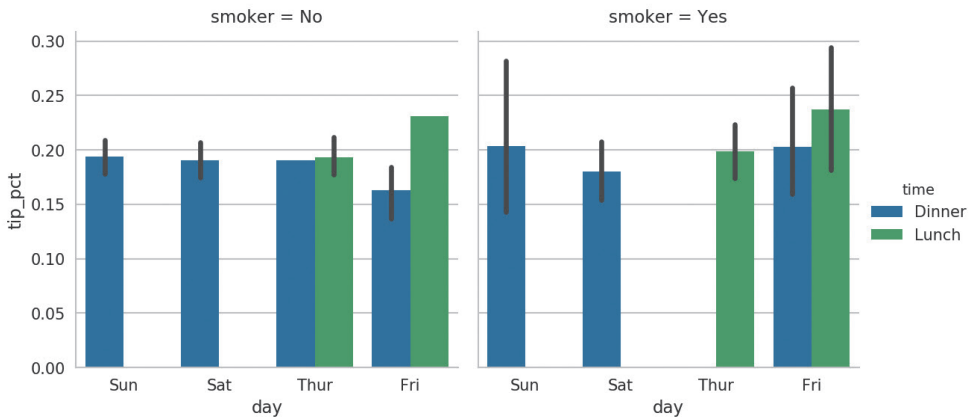


Рис. 9.26. Процент чаевых в зависимости от дня, времени суток и курения

Вместо того чтобы использовать для группировки по 'time' столбики разных цветов внутри фасетки, мы можем расширить фасетную сетку, добавив по одной строке на каждое значение time (рис. 9.27):

```
In [113]: sns.catplot(x="day", y="tip_pct", row="time",
.....:               col="smoker",
.....:               kind="bar", data=tips[tips.tip_pct < 1])
```

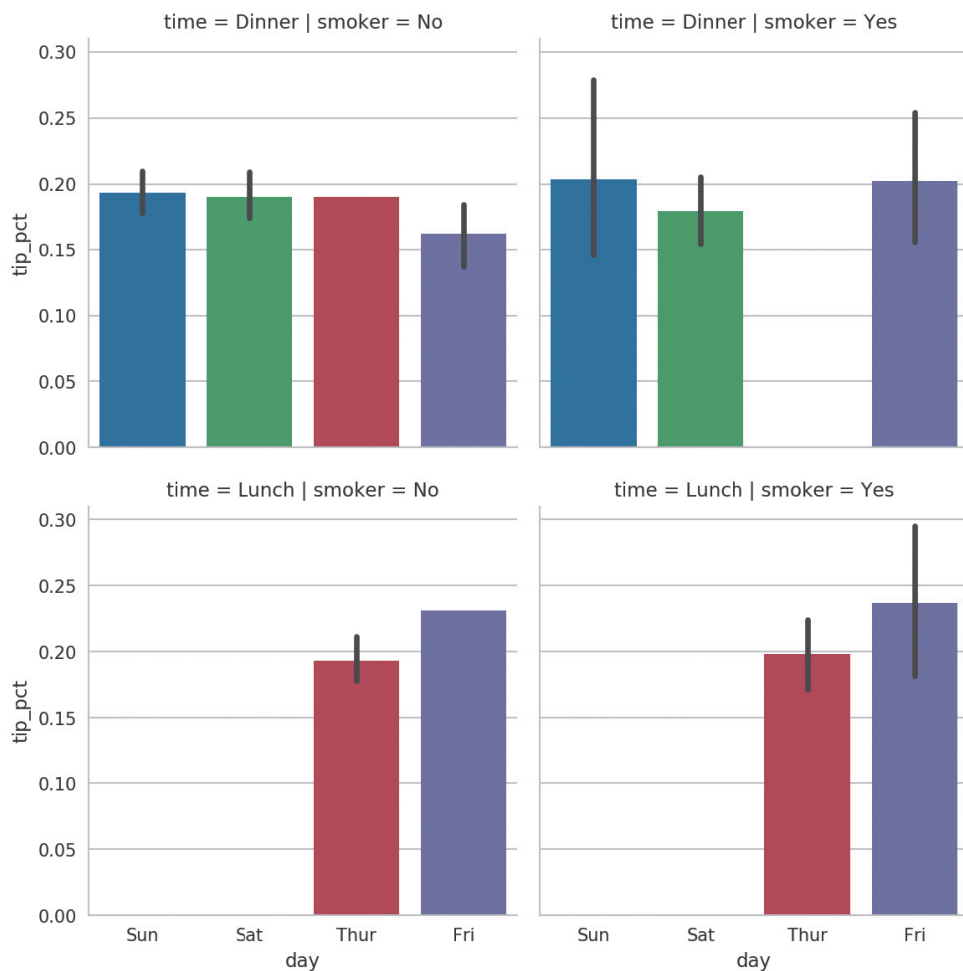


Рис. 9.27. Процент чаевых в зависимости от дня; фасетки по времени суток и курению

Функция `catplot` поддерживает и другие типы графиков, которые могут оказаться полезными в зависимости от того, что мы пытаемся показать. Например, *диаграммы размаха* (на которых изображаются медиана, квартили и выбросы) могут оказаться эффективным средством визуализации (рис. 9.28):

```
In [114]: sns.catplot(x="tip_pct", y="day", kind="box",
.....:               data=tips[tips.tip_pct < 0.5])
```

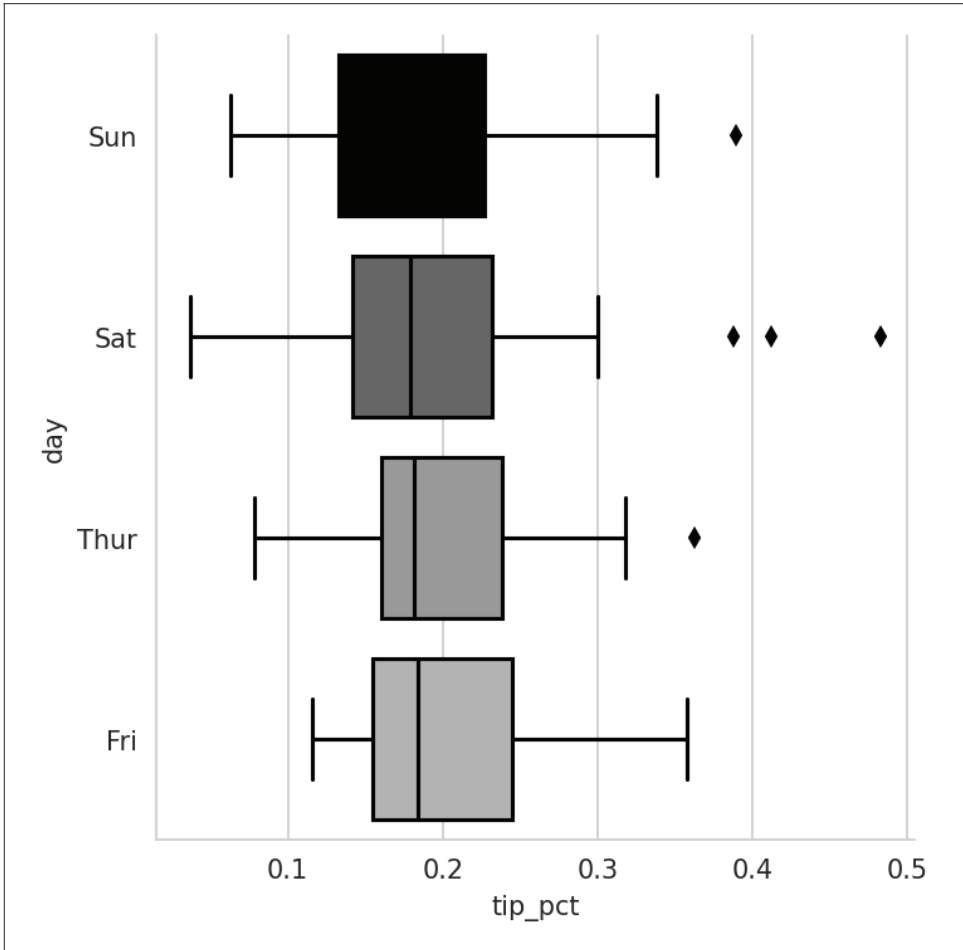


Рис. 9.28. Диаграмма размаха, описывающая процент чаевых в зависимости от дня

С помощью более общего класса `seaborn.FacetGrid` можно создавать собственные фасетные сетки. Дополнительные сведения см. в документации по `seaborn` по адресу <https://seaborn.pydata.org/>.

9.3. ДРУГИЕ СРЕДСТВА ВИЗУАЛИЗАЦИИ ДЛЯ PYTHON

Как обычно бывает в проектах с открытым исходным кодом, в средствах создания графики для Python нехватки не ощущается (их слишком много, чтобы все перечислить). Начиная с 2010 года усилия разработчиков были сосредоточены на создании интерактивной графики для публикации в вебе. Благодаря таким инструментам, как Altair (<https://altair-viz.github.io/>), Bokeh (<http://bokeh.pydata.org/>) и Plotly (<https://github.com/plotly/plotly.py>), стало возможно описывать на Python динамичную интерактивную графику, ориентированную на отображение в браузере.

Если вы создаете статическую графику для печати или для веба, то я рекомендую начать с `matplotlib` и основанной на ней библиотеках, таких как `pandas` и `seaborn`. Если же требования к визуализации иные, то полезно изучить какой-нибудь из других имеющихся в сети инструментов. Настоятельно советую изучать экосистему, потому что она продолжает развиваться и устремлена в будущее.

Есть прекрасная книга по визуализации данных: Claus O. Wilke «Fundamentals of Data Visualization» (O'Reilly). Она распространяется в печатной форме и на сайте Клауса по адресу <https://clauswilke.com/dataviz>.

9.4. ЗАКЛЮЧЕНИЕ

В этой главе нашей целью было познакомиться с основными средствами визуализации на основе `pandas`, `matplotlib` и `seaborn`. Если наглядное представление результатов анализа данных важно для вашей работы, то рекомендую почитать еще что-нибудь об эффективной визуализации данных. Это активная область исследований, поэтому вы найдете немало отличных учебных ресурсов как в сети, так и в печатной форме.

В следующей главе мы займемся агрегированием данных и операциями группировки в `pandas`.

Агрегирование данных и групповые операции

Разбиение набора данных на группы и применение некоторой функции к каждой группе, будь то в целях агрегирования или преобразования, зачастую является одной из важнейших операций анализа данных. После загрузки, слияния и подготовки набора данных обычно вычисляют статистику по группам или, возможно, *сводные таблицы* для построения отчета или визуализации. В библиотеке pandas имеется гибкий и быстрый механизм `groupby`, который позволяет формировать продольные и поперечные срезы, а также агрегировать наборы данных естественным образом.

Одна из причин популярности реляционных баз данных и языка SQL (структурированного языка запросов) – простота соединения, фильтрации, преобразования и агрегирования данных. Однако в том, что касается групповых операций, языки запросов типа SQL несколько ограничены. Как мы увидим, выразительность и мощь языка Python и библиотеки pandas позволяют выполнять гораздо более сложные групповые операции, выразив их в виде написанных нами функций, которые манипулируют данными, ассоциированными с каждой группой. В этой главе мы изучим следующие возможности:

- разделение объекта pandas на части по одному или нескольким ключам (в виде функций, массивов или имен столбцов объекта DataFrame);
- вычисление групповых статистик, например общее количество, среднее, стандартное отклонение или определенная пользователем функция;
- применение переменного множества функций к каждому столбцу DataFrame;
- применение внутригрупповых преобразований или иных манипуляций, например нормировка, линейная регрессия, ранжирование или выборка подмножества;
- вычисление сводных таблиц и перекрестное табулирование;
- квантильный анализ и другие виды статистического анализа групп.



Частный случай механизма `groupby`, агрегирование временных рядов по времени, в этой книге называется *передискретизацией* и рассматривается отдельно в главе 11.

Как и в других главах, начнем с импорта NumPy и pandas:

```
In [12]: import numpy as np
```

```
In [13]: import pandas as pd
```

10.1. КАК ПРЕДСТАВЛЯТЬ СЕБЕ ГРУППОВЫЕ ОПЕРАЦИИ

Хэдли Уикхэм (Hadley Wickham), автор многих популярных пакетов на языке программирования R, предложил для групповых операций термин *разделение–применение–объединение*, который, как мне кажется, удачно описывает процесс. На первом этапе данные, хранящиеся в объекте pandas, будь то Series, DataFrame или что-то еще, *разделяются* на группы по одному или нескольким указанным вами *ключам*. Разделение производится вдоль одной оси объекта. Например, DataFrame можно группировать по строкам (*axis="index"*) или по столбцам (*axis="columns"*). Затем к каждой группе *применяется* некоторая функция, которая порождает новое значение. Наконец, результаты применения всех функций *объединяются* в результирующий объект. Форма результирующего объекта обычно зависит от того, что именно проделывается с данными. На рис. 10.1 показан схематический пример простого группового агрегирования.

Ключи группировки могут задаваться по-разному и необязательно должны быть одного типа:

- список или массив значений той же длины, что ось, по которой производится группировка;
- значение, определяющее имя столбца объекта DataFrame;
- словарь или объект Series, определяющий соответствие между значениями на оси группировки и именами групп;
- функция, которой передается индекс оси или отдельные метки из этого индекса.

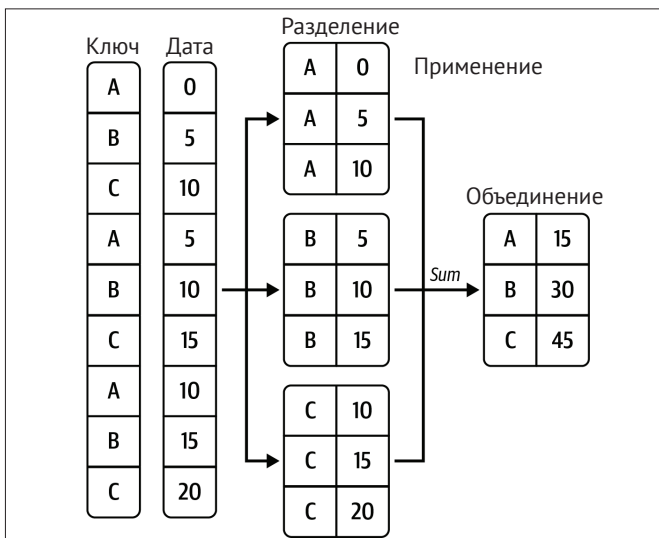


Рис. 10.1. Иллюстрация группового агрегирования

Отметим, что последние три метода – просто различные способы порождения массива значений, используемого далее для разделения объекта на группы. Не пугайтесь, если это кажется слишком абстрактным. В этой главе будут приведены многочисленные примеры каждого метода. Для начала рассмотрим очень простой табличный набор данных, представленный в виде объекта DataFrame:

```
In [14]: df = pd.DataFrame({"key1" : ["a", "a", None, "b", "b", "a", None],
.....:                    "key2" : pd.Series([1, 2, 1, 2, 1, None, 1], dtype="Int64"),
.....:                    "data1" : np.random.standard_normal(7),
.....:                    "data2" : np.random.standard_normal(7)})

In [15]: df
Out[15]:
```

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
2	None	1	-0.519439	1.246435
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221
5	a	<NA>	1.393406	0.274992
6	None	1	0.092908	0.228913

Пусть требуется вычислить среднее по столбцу `data1`, используя метки групп в столбце `key1`. Сделать это можно несколькими способами. Первый – взять столбец `data1` и вызвать метод `groupby`, передав ему столбец (объект Series) `key1`:

```
In [16]: grouped = df["data1"].groupby(df["key1"])

In [17]: grouped
Out[17]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa9270e0a00>
```

Переменная `grouped` – это объект `GroupBy`. Пока что он не вычислил ничего, кроме промежуточных данных о групповом ключе `df['key1']`. Идея в том, что этот объект хранит всю информацию, необходимую для последующего применения некоторой операции к каждой группе. Например, чтобы вычислить среднее по группам, мы можем вызвать метод `mean` объекта `GroupBy`:

```
In [18]: grouped.mean()
Out[18]:
```

key1	
a	0.555881
b	0.705025

Name: data1, dtype: float64

Позже, в разделе 10.2, я подробнее объясню, что происходит при вызове `.mean()`. Важно, что данные (объект Series) агрегированы путем разделения данных по групповому ключу, и в результате создан новый объект Series, индексированный уникальными значениями в столбце `key1`. Получившийся индекс назван `'key1'`, потому что так назывался столбец `df['key1']` объекта DataFrame.

Если бы мы передали несколько массивов в виде списка, то получили бы другой результат:

```
In [19]: means = df["data1"].groupby([df["key1"], df["key2"]]).mean()
```

```
In [20]: means
```

```
Out[20]:
```

```
key1  key2
a      1    -0.204708
      2     0.478943
B      1     1.965781
      2    -0.555730
Name: data1, dtype: float64
```

В этом случае данные сгруппированы по двум ключам, а у результирующего объекта Series имеется иерархический индекс, который состоит из уникальных пар значений ключей, встретившихся в исходных данных:

```
In [21]: means.unstack()
```

```
Out[21]:
```

```
key2      1      2
key1
a    -0.204708  0.478943
b     1.965781 -0.555730
```

В этом примере групповыми ключами были объекты Series, но можно было бы использовать и произвольные массивы правильной длины:

```
In [22]: states = np.array(["OH", "CA", "CA", "OH", "OH", "CA", "OH"])
```

```
In [23]: years = [2005, 2005, 2006, 2005, 2006, 2005, 2006]
```

```
In [24]: df["data1"].groupby([states, years]).mean()
```

```
Out[24]:
```

```
CA  2005     0.936175
     2006    -0.519439
OH  2005    -0.380219
     2006     1.029344
Name: data1, dtype: float64
```

Часто информация о группировке находится в том же объекте DataFrame, что и группируемые данные. В таком случае в качестве групповых ключей можно передать имена столбцов (не важно, что они содержат: строки, числа или другие объекты Python):

```
In [25]: df.groupby("key1").mean()
```

```
Out[25]:
```

```
key2    data1    data2
key1
a      1.5  0.555881  0.441920
b      1.5  0.705025 -0.144516
```

```
In [26]: df.groupby("key2").mean()
```

```
Out[26]:
```

```
data1    data2
key2
1      0.333636  0.115218
2     -0.038393  0.888106
```

```
In [27]: df.groupby(["key1", "key2"]).mean()
```

```
Out[27]:
```

		data1	data2
key1	key2		
a	1	-0.204708	0.281746
	2	0.478943	0.769023
b	1	1.965781	-1.296221
	2	-0.555730	1.007189

Вероятно, вы обратили внимание, что во втором случае – `df.groupby("key1").mean()` – результат не содержал столбца `key1`. Поскольку `df["key1"]` содержит нечисловые данные, говорят, что это *посторонний столбец*, и в результат не включают. По умолчанию агрегируются все числовые столбцы, хотя можно выбрать и некоторое их подмножество, как мы вскоре увидим.

Вне зависимости от цели использования `groupby` у объекта `GroupBy` есть полезный метод `size`, который возвращает объект `Series`, содержащий размеры групп:

```
In [28]: df.groupby(["key1", "key2"]).size()
Out[28]:
```

key1	key2	
a	1	1
	2	1
b	1	1
	2	1

dtype: int64

Обратите внимание, что данные, соответствующие отсутствующим в групповом ключе значениям, по умолчанию исключаются из результата. Это поведение можно подавить, передав `groupby` параметр `dropna=False`:

```
In [29]: df.groupby("key1", dropna=False).size()
Out[29]:
```

key1	
a	3
b	2
NaN	2

dtype: int64

```
In [30]: df.groupby(["key1", "key2"], dropna=False).size()
Out[30]:
```

key1	key2	
a	1	1
	2	1
	<NA>	1
b	1	1
	2	1
NaN	1	2

dtype: int64

Еще одна групповая функция в духе `size` – это `count`; она вычисляет количество отличных от null значений в каждой группе:

```
In [31]: df.groupby("key1").count()
Out[31]:
```

	key2	data1	data2
key1			
a	2	3	3
b	2	2	2

Обход групп

Объект, возвращенный `groupby`, поддерживает итерирование, в результате которого генерируется последовательность 2-кортежей, содержащих имя группы и блок данных. Взгляните на следующий код:

```
In [32]: for name, group in df.groupby("key1"):
.....:     print(name)
.....:     print(group)
.....:
```

a

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
5	a	<NA>	1.393406	0.274992

b

	key1	key2	data1	data2
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221

В случае нескольких ключей первым элементом кортежа будет кортеж, содержащий значения ключей:

```
In [33]: for (k1, k2), group in df.groupby(["key1", "key2"]):
.....:     print((k1, k2))
.....:     print(group)
.....:
```

('a', 1)

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746

('a', 2)

	key1	key2	data1	data2
1	a	2	0.478943	0.769023

('b', 1)

	key1	key2	data1	data2
4	b	1	1.965781	-1.296221

('b', 2)

	key1	key2	data1	data2
3	b	2	-0.55573	1.007189

Разумеется, только вам решать, что делать с блоками данных. Возможно, пригодится следующий однострочный код, который строит словарь блоков:

```
In [34]: pieces = {name: group for name, group in df.groupby("key1")}

In [35]: pieces["b"]
Out[35]:
```

	key1	key2	data1	data2
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221

По умолчанию метод `groupby` группирует по оси `axis="index"`, но можно задать любую другую ось. Например, в нашем примере столбцы объекта `df` можно было бы сгруппировать по тому, начинаются они с "key" или "data":

```
In [36]: grouped = df.groupby({"key1": "key", "key2": "key",
.....:                        "data1": "data", "data2": "data"}, axis="columns")
```

Группы можно распечатать следующим образом:

```
In [37]: for group_key, group_values in grouped:
.....:     print(group_key)
.....:     print(group_values)
.....:
data
      data1      data2
0 -0.204708  0.281746
1  0.478943  0.769023
2 -0.519439  1.246435
3 -0.555730  1.007189
4  1.965781 -1.296221
5  1.393406  0.274992
6  0.092908  0.228913
key
      key1  key2
0      a      1
1      a      2
2  None      1
3      b      2
4      b      1
5      a  <NA>
6  None      1
```

Выборка столбца или подмножества столбцов

Доступ по индексу к объекту GroupBy, полученному группировкой объекта DataFrame путем задания имени столбца или массива имен столбцов, имеет тот же эффект, что выборка этих столбцов для агрегирования. Это означает, что

```
df.groupby("key1")["data1"]
df.groupby("key1")[["data2"]]
```

– просто удобный способ записи вместо:

```
df["data1"].groupby(df["key1"])
df[["data2"]].groupby(df["key1"])
```

Большие наборы данных обычно желательно агрегировать лишь по немногим столбцам. Так, чтобы в приведенном выше примере вычислить среднее только по столбцу `data2` и получить результат в виде DataFrame, можно было бы написать:

```
In [38]: df.groupby(["key1", "key2"])[["data2"]].mean()
Out[38]:
data2
key1 key2
a      1    0.281746
      2    0.769023
b      1   -1.296221
      2    1.007189
```

В результате этой операции доступа по индексу возвращается сгруппированный DataFrame, если передан список, или массив, или сгруппированный Series, если передано только одно имя столбца:


```
In [39]: s_grouped = df.groupby(["key1", "key2"])["data2"]

In [40]: s_grouped
Out[40]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa9270e3520>

In [41]: s_grouped.mean()
Out[41]:
key1 key2
a      1    0.281746
      2    0.769023
b      1   -1.296221
      2    1.007189
Name: data2, dtype: float64
```

Группировка с помощью словарей и объектов Series

Информацию о группировке можно передавать не только в виде массива. Рассмотрим еще один объект DataFrame:

```
In [42]: people = pd.DataFrame(np.random.standard_normal((5, 5)),
.....:                        columns=["a", "b", "c", "d", "e"],
.....:                        index=["Joe", "Steve", "Wanda", "Jill", "Trey"])

In [43]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [44]: people
Out[44]:
```

	a	b	c	d	e
Joe	1.352917	0.886429	-2.001637	-0.371843	1.669025
Steve	-0.438570	-0.539741	0.476985	3.248944	-1.021228
Wanda	-0.577087	NaN	NaN	0.523772	0.000940
Jill	1.343810	-0.713544	-0.831154	-2.370232	-1.860761
Trey	-0.860757	0.560145	-1.265934	0.119827	-1.063512

Теперь предположим, что имеется соответствие между столбцами и группами и нужно просуммировать столбцы для каждой группы:

```
In [45]: mapping = {"a": "red", "b": "red", "c": "blue",
.....:             "d": "blue", "e": "red", "f": "orange"}
```

Из этого словаря нетрудно построить массив и передать его `groupby`, но можно вместо этого передать и сам словарь (я включил ключ "f", чтобы показать, что неиспользуемые ключи группировки не составляют проблемы):

```
In [46]: by_column = people.groupby(mapping, axis="columns")

In [47]: by_column.sum()
Out[47]:
```

	blue	red
Joe	-2.373480	3.908371
Steve	3.725929	-1.999539
Wanda	0.523772	-0.576147
Jill	-3.201385	-1.230495
Trey	-1.146107	-1.364125

То же самое относится и к объекту Series, который можно рассматривать как отображение фиксированного размера.

```
In [48]: map_series = pd.Series(mapping)

In [49]: map_series
Out[49]:
a      red
b      red
c     blue
d     blue
e      red
f   orange
dtype: object

In [50]: people.groupby(map_series, axis="columns").count()
Out[50]:
         blue  red
Joe         2    3
Steve        2    3
Wanda        1    2
Jill         2    3
Trey         2    3
```

Группировка с помощью функций

Использование функций Python – более абстрактный способ определения соответствия групп по сравнению со словарями или объектами Series. Функция, переданная в качестве группового ключа, будет вызвана по одному разу для каждого значения в индексе (или по одному разу для каждого значения в столбце, если используется `axis="columns"`), а возвращенные ей значения станут именами групп. Конкретно, рассмотрим пример объекта DataFrame из предыдущего раздела, где значениями индекса являются имена людей. Пусть требуется сгруппировать по длине имени; можно было бы вычислить массив длин строк, но лучше вместо этого просто передать функцию `len`:

```
In [51]: people.groupby(len).sum()
Out[51]:
         a         b         c         d         e
3  1.352917  0.886429 -2.001637 -0.371843  1.669025
4  0.483052 -0.153399 -2.097088 -2.250405 -2.924273
5 -1.015657 -0.539741  0.476985  3.772716 -1.020287
```

Использование попеременно функций, массивов, словарей и объектов Series вполне допустимо, потому что внутри все преобразуется в массивы:

```
In [52]: key_list = ["one", "one", "one", "two", "two"]

In [53]: people.groupby([len, key_list]).min()
Out[53]:
         a         b         c         d         e
3 one  1.352917  0.886429 -2.001637 -0.371843  1.669025
4 two -0.860757 -0.713544 -1.265934 -2.370232 -1.860761
5 one -0.577087 -0.539741  0.476985  0.523772 -1.021228
```

Группировка по уровням индекса

Наконец, иерархически индексированные наборы данных можно агрегировать по одному из уровней индекса оси. Рассмотрим пример:

```
In [54]: columns = pd.MultiIndex.from_arrays([["US", "US", "US", "JP", "JP"],
....:                                     [1, 3, 5, 1, 3]],
....:                                     names=["cty", "tenor"])

In [55]: hier_df = pd.DataFrame(np.random.standard_normal((4, 5)), columns=columns)
```

```
In [56]: hier_df
Out[56]:
```

	US			JP	
cty					
tenor	1	3	5	1	3
0	0.332883	-2.359419	-0.199543	-1.541996	-0.970736
1	-1.307030	0.286350	0.377984	-0.753887	0.331286
2	1.349742	0.069877	0.246674	-0.011862	1.004812
3	1.327195	-0.919262	-1.549106	0.022185	0.758363

Для группировки по уровню нужно передать номер или имя уровня в именованном параметре `level`:

```
In [57]: hier_df.groupby(level="cty", axis="columns").count()
Out[57]:
```

cty	JP	US
0	2	3
1	2	3
2	2	3
3	2	3

10.2. АГРЕГИРОВАНИЕ ДАННЫХ

Под *агрегированием* я обычно понимаю любое преобразование данных, которое порождает скалярные значения из массива. В примерах выше мы встречали несколько таких преобразований: `mean`, `count`, `min` и `sum`. Вероятно, вам интересно, что происходит при вызове `mean()` для объекта `GroupBy`. Реализации многих стандартных операций агрегирования, в частности перечисленных в табл. 10.1, оптимизированы. Однако необязательно ограничиваться только этими методами.

Таблица 10.1. Оптимизированные методы агрегирования

Имя функции	Описание
<code>any</code> , <code>all</code>	Возвращают <code>True</code> , если хотя бы одно или все значения, отличные от NA, «похожи на истину»
<code>count</code>	Количество отличных от NA значений в группе
<code>cummin</code> , <code>cummax</code>	Накопительный минимум и максимум отличных от NA значений
<code>cumsum</code>	Накопительная сумма отличных от NA значений
<code>cumprod</code>	Накопительное произведение отличных от NA значений
<code>first</code> , <code>last</code>	Первое и последнее отличные от NA значения
<code>mean</code>	Среднее отличных от NA значений
<code>median</code>	Медиана отличных от NA значений

Окончание табл. 10.1

Имя функции	Описание
<code>min, max</code>	Минимальное и максимальное отличные от NA значения
<code>nth</code>	Значение, которое занимало бы позицию <i>n</i> , если бы данные были отсортированы
<code>ohlc</code>	Статистика типа «начальное–наибольшее–наименьшее–конечное» для данных типа временных рядов
<code>prod</code>	Произведение отличных от NA значений
<code>quantile</code>	Выборочный квантиль
<code>rank</code>	Порядковые ранги отличных от NA значений, как, например, при вызове <code>Series.rank</code>
<code>size</code>	Вычисляет размеры группы и возвращает результат в виде объекта <code>Series</code>
<code>sum</code>	Сумма отличных от NA значений
<code>std, var</code>	Выборочное стандартное отклонение и выборочная дисперсия

Вы можете придумать собственные способы агрегирования и, кроме того, вызвать любой метод, определенный для группируемого объекта. Например, метод `nsmallest` выбирает из данных запрошенное количество наименьших значений. Хотя в классе `GroupBy` метод `nsmallest` явно не реализован, мы все-таки можем использовать его неоптимизированную реализацию. На самом деле объект `GroupBy` разбивает `Series` на части, вызывает `piece.nsmallest(n)` для каждой части, а затем собирает результаты в итоговый объект.

```
In [58]: df
Out[58]:
```

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
2	None	1	-0.519439	1.246435
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221
5	a	<NA>	1.393406	0.274992
6	None	1	0.092908	0.228913

```
In [59]: grouped = df.groupby("key1")

In [60]: grouped["data1"].nsmallest(2)
Out[60]:
```

key1		
a	0	-0.204708
	1	0.478943
b	3	-0.555730
	4	1.965781

```
Name: data1, dtype: float64
```

Для использования собственных функций агрегирования передайте функцию, агрегирующую массив, методу `aggregate` или его псевдониму `agg`:

3	23.68	3.31	No	Sun	Dinner	2
4	24.59	3.61	No	Sun	Dinner	4

Теперь добавим в него столбец `tip_pct`, содержащий размер чаевых в процентах от суммы счета:

```
In [66]: tips["tip_pct"] = tips["tip"] / tips["total_bill"]

In [67]: tips.head()
Out[67]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Как мы уже видели, для агрегирования объекта `Series` или всех столбцов объекта `DataFrame` достаточно воспользоваться методом `aggregate`, передав ему требуемую функцию, или вызвать метод `mean`, `std` и им подобный. Однако иногда нужно использовать разные функции в зависимости от столбца или сразу несколько функций. К счастью, сделать это совсем нетрудно, что я и продемонстрирую в следующих примерах. Для начала сгруппирую столбец `tips` по значениям `sex` и `smoker`:

```
In [60]: grouped = tips.groupby(['day', 'smoker'])
```

Отметим, что в случае описательных статистик типа тех, что приведены в табл. 10.1, можно передать имя функции в виде строки:

```
In [69]: grouped_pct = grouped["tip_pct"]

In [70]: grouped_pct.agg("mean")
Out[70]:
```

day	smoker	
Fri	No	0.151650
	Yes	0.174783
Sat	No	0.158048
	Yes	0.147906
Sun	No	0.160113
	Yes	0.187250
Thur	No	0.160298
	Yes	0.163863

Name: tip_pct, dtype: float64

Если вместо этого передать список функций или имен функций, то будет возвращен объект `DataFrame`, в котором имена столбцов совпадают с именами функций:

```
In [71]: grouped_pct.agg(["mean", "std", "peak_to_peak"])
Out[71]:
```

		mean	std	peak_to_peak
day	smoker			
Fri	No	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	No	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095

Sun	No	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	No	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

Здесь мы передали список функций агрегирования методу `agg`, который независимо вычисляет агрегаты для групп данных.

Совершенно необязательно соглашаться с именами столбцов, предложенными объектом `GroupBy`; в частности все лямбда-функции называются `'<lambda>'`, поэтому различить их затруднительно (можете убедиться сами, распечатав атрибут функции `__name__`). Поэтому если передать список кортежей вида `(name, function)`, то в качестве имени столбца `DataFrame` будет взят первый элемент кортежа (можно считать, что список 2-кортежей – упорядоченное отображение):

```
In [72]: grouped_pct.agg([("average", "mean"), ("stdev", np.std)])
Out[72]:
```

		average	stdev
day	smoker		
Fri	No	0.151650	0.028123
	Yes	0.174783	0.051293
Sat	No	0.158048	0.039767
	Yes	0.147906	0.061375
Sun	No	0.160113	0.042347
	Yes	0.187250	0.154134
Thur	No	0.160298	0.038774
	Yes	0.163863	0.039389

В случае `DataFrame` диапазон возможностей шире, поскольку можно задавать список функций, применяемых ко всем столбцам, или разные функции для разных столбцов. Допустим, нам нужно вычислить три одинаковые статистики для столбцов `tip_pct` и `total_bill`:

```
In [73]: functions = ["count", "mean", "max"]

In [74]: result = grouped[["tip_pct", "total_bill"]].agg(functions)

In [75]: result
Out[75]:
```

		tip_pct			total_bill		
		count	mean	max	count	mean	max
day	smoker						
Fri	No	4	0.151650	0.187735	4	18.420000	22.75
	Yes	15	0.174783	0.263480	15	16.813333	40.17
Sat	No	45	0.158048	0.291990	45	19.661778	48.33
	Yes	42	0.147906	0.325733	42	21.276667	50.81
Sun	No	57	0.160113	0.252672	57	20.506667	48.17
	Yes	19	0.187250	0.710345	19	24.120000	45.35
Thur	No	45	0.160298	0.266312	45	17.113111	41.19
	Yes	17	0.163863	0.241255	17	19.190588	43.11

Как видите, в результирующем `DataFrame` имеются иерархические столбцы – точно так же, как было бы, если бы мы агрегировали каждый столбец по отдельности, а потом склеили результаты с помощью метода `concat`, передав ему имена столбцов в качестве аргумента `keys`:

```
In [76]: result["tip_pct"]
Out[76]:
```

		count	mean	max
day	smoker			
Fri	No	4	0.151650	0.187735
	Yes	15	0.174783	0.263480
Sat	No	45	0.158048	0.291990
	Yes	42	0.147906	0.325733
Sun	No	57	0.160113	0.252672
	Yes	19	0.187250	0.710345
Thur	No	45	0.160298	0.266312
	Yes	17	0.163863	0.241255

Как и раньше, можно передавать список кортежей, содержащий желаемые имена:

```
In [77]: ftuples = [("Average", "mean"), ("Variance", np.var)]

In [78]: grouped[["tip_pct", "total_bill"]].agg(ftuples)
Out[78]:
```

		tip_pct		total_bill	
		Average	Variance	Average	Variance
day	smoker				
Fri	No	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	No	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535
Sun	No	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	No	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

Предположим далее, что требуется применить потенциально различные функции к одному или нескольким столбцам. Делается это путем передачи методу `agg` словаря, который содержит отображение имен столбцов на любой из рассмотренных выше объектов, задающих функции:

```
In [79]: grouped.agg({"tip" : np.max, "size" : "sum"})
Out[79]:
```

		tip	size
day	smoker		
Fri	No	3.50	9
	Yes	4.73	31
Sat	No	9.00	115
	Yes	10.00	104
Sun	No	6.00	167
	Yes	6.50	49
Thur	No	6.70	112
	Yes	5.00	40

```
In [80]: grouped.agg({"tip_pct" : ["min", "max", "mean", "std"],
.....:                 "size" : "sum"})
Out[80]:
```

		tip_pct		size		
		min	max	mean	std	sum
day	smoker					
Fri	No	0.120385	0.187735	0.151650	0.028123	9

	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	No	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	No	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	No	0.072961	0.266312	0.160298	0.038774	112
	Yes	0.090014	0.241255	0.163863	0.039389	40

Объект DataFrame будет содержать иерархические столбцы, только если хотя бы к одному столбцу было применено несколько функций.

Возврат агрегированных данных без индексов строк

Во всех рассмотренных выше примерах агрегированные данные сопровождались индексом, иногда иерархическим, составленным из уникальных встретившихся комбинаций групповых ключей. Такое поведение не всегда желательно, поэтому его можно подавить, передав методу `groupby` аргумент `as_index=False`:

```
In [81]: tips.groupby(["day", "smoker"], as_index=False).mean()
Out[81]:
```

	day	smoker	total_bill	tip	size	tip_pct
0	Fri	No	18.420000	2.812500	2.250000	0.151650
1	Fri	Yes	16.813333	2.714000	2.066667	0.174783
2	Sat	No	19.661778	3.102889	2.555556	0.158048
3	Sat	Yes	21.276667	2.875476	2.476190	0.147906
4	Sun	No	20.506667	3.167895	2.929825	0.160113
5	Sun	Yes	24.120000	3.516842	2.578947	0.187250
6	Thur	No	17.113111	2.673778	2.488889	0.160298
7	Thur	Yes	19.190588	3.030000	2.352941	0.163863

Разумеется, для получения данных в таком формате всегда можно вызвать метод `reset_index` результата. Аргумент `as_index=False` просто позволяет избежать некоторых лишних вычислений.

10.3. МЕТОД APPLY: ОБЩИЙ ПРИНЦИП

РАЗДЕЛЕНИЯ — ПРИМЕНЕНИЯ — ОБЪЕДИНЕНИЯ

Самым общим из методов класса `GroupBy` является `apply`, ему мы и посвятим остаток этого раздела. `apply` разделяет обрабатываемый объект на части, вызывает для каждой части переданную функцию, а затем пытается конкатенировать все части вместе.

Возвращаясь к набору данных о чаевых, предположим, что требуется выбрать первые пять значений `tip_pct` в каждой группе. Прежде всего нетрудно написать функцию, которая отбирает строки с наибольшими значениями в указанном столбце:

```
In [82]: def top(df, n=5, column="tip_pct"):
.....:     return df.sort_values(column, ascending=False)[:n]

In [83]: top(tips, n=6)
Out[83]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
232	11.61	3.39	No	Sat	Dinner	2	0.291990
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Если теперь сгруппировать по столбцу `smoker` и вызвать метод `apply`, передав ему эту функцию, то получим следующее:

```
In [84]: tips.groupby("smoker").apply(top)
Out[84]:
```

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	232	11.61	3.39	No	Sat	Dinner	2	0.291990
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
	185	20.69	5.00	No	Sun	Dinner	5	0.241663
	88	24.71	5.85	No	Thur	Lunch	2	0.236746
Yes	172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Что здесь произошло? Сначала объект `DataFrame` `tips` разбит на группы по значению `smoker`. Затем для каждой группы вызывается функция `top`, после чего результаты склеиваются методом `pandas.concat`, а частям сопоставляются метки, совпадающие с именами групп. Поэтому результат имеет иерархический индекс, внутренний уровень которого содержит индексные значения из исходного объекта `DataFrame`.

Если передать методу `apply` функцию, которая принимает еще какие-то позиционные или именованные аргументы, то их можно передать вслед за самой функцией:

```
In [85]: tips.groupby(["smoker", "day"]).apply(top, n=1, column="total_bill")
Out[85]:
```

			total_bill	tip	smoker	day	time	size	tip_pct
smoker day									
No	Fri	94	22.75	3.25	No	Fri	Dinner	2	0.142857
	Sat	212	48.33	9.00	No	Sat	Dinner	4	0.186220
	Sun	156	48.17	5.00	No	Sun	Dinner	6	0.103799
	Thur	142	41.19	5.00	No	Thur	Lunch	5	0.121389
Yes	Fri	95	40.17	4.73	Yes	Fri	Dinner	4	0.117750
	Sat	170	50.81	10.00	Yes	Sat	Dinner	3	0.196812
	Sun	182	45.35	3.50	Yes	Sun	Dinner	3	0.077178
	Thur	197	43.11	5.00	Yes	Thur	Lunch	4	0.115982

Это лишь простейшие приемы, а вообще возможности `apply` ограничены только вашей изобретательностью. Что именно делает переданная функция, решать вам, требуется лишь, чтобы она возвращала объект `pandas` или скалярное значение. Далее в этой главе будут в основном примеры, показывающие, как решать различные задачи с помощью `groupby`.

Вы, наверное, помните, что выше я вызывал метод `describe` от имени объекта `GroupBy`:

```
In [86]: result = tips.groupby("smoker")["tip_pct"].describe()

In [87]: result
Out[87]:
```

	count	mean	std	min	25%	50%	75% \
smoker							
No	151.0	0.159328	0.039910	0.056797	0.136906	0.155625	0.185014
Yes	93.0	0.163196	0.085119	0.035638	0.106771	0.153846	0.195059
	max						
smoker							
No	0.291990						
Yes	0.710345						

```

In [88]: result.unstack("smoker")
Out[88]:
```

	smoker	
count	No	151.000000
	Yes	93.000000
mean	No	0.159328
	Yes	0.163196
std	No	0.039910
	Yes	0.085119
min	No	0.056797
	Yes	0.035638
25%	No	0.136906
	Yes	0.106771
50%	No	0.155625
	Yes	0.153846
75%	No	0.185014
	Yes	0.195059
max	No	0.291990
	Yes	0.710345

```
dtype: float64
```

Когда от имени GroupBy вызывается метод типа `describe`, на самом деле выполняются такие предложения:

```
def f(group):
    return group.describe()

grouped.apply(f)
```

Подавление групповых ключей

В примерах выше мы видели, что у результирующего объекта имеется иерархический индекс, образованный групповыми ключами и индексами каждой части исходного объекта. Создание этого индекса можно подавить, передав методу `groupby` параметр `group_keys=False`:

```
In [89]: tips.groupby("smoker", group_keys=False).apply(top)
Out[89]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
232	11.61	3.39	No	Sat	Dinner	2	0.291990
149	7.51	2.00	No	Thur	Lunch	2	0.266312
51	10.29	2.60	No	Sun	Dinner	2	0.252672
185	20.69	5.00	No	Sun	Dinner	5	0.241663
88	24.71	5.85	No	Thur	Lunch	2	0.236746

172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Квантильный и интервальный анализы

Напомним, что в главе 8 шла речь о некоторых средствах библиотеки pandas, в том числе функциях `pandas.cut` и `pandas.qcut`, которые позволяют распределить данные по интервалам, размер которых задан вами или определяется выборочными квантилями. В сочетании с функцией `groupby` эти методы позволяют очень просто подвергнуть набор данных интервальному или квантильному анализам. Рассмотрим простой набор случайных данных и распределение по интервалам равной длины с помощью `pandas.cut`:

```
In [90]: frame = pd.DataFrame({"data1": np.random.standard_normal(1000),
.....:                        "data2": np.random.standard_normal(1000)})

In [91]: frame.head()
Out[91]:
   data1    data2
0 -0.660524 -0.612905
1  0.862580  0.316447
2 -0.010032  0.838295
3  0.050009 -1.034423
4  0.670216  0.434304

In [92]: quartiles = pd.cut(frame["data1"], 4)

In [93]: quartiles.head(10)
Out[93]:
0    (-1.23, 0.489]
1    (0.489, 2.208]
2    (-1.23, 0.489]
3    (-1.23, 0.489]
4    (0.489, 2.208]
5    (0.489, 2.208]
6    (-1.23, 0.489]
7    (-1.23, 0.489]
8    (-2.956, -1.23]
9    (-1.23, 0.489]
Name: data1, dtype: category
Categories (4, interval[float64, right]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.208] < (2.208, 3.928]]
```

Объект `Categorical`, возвращаемый функцией `cut`, можно передать непосредственно `groupby`. Следовательно, набор групповых статистик для квантилей можно вычислить следующим образом:

```
In [94]: def get_stats(group):
.....:     return pd.DataFrame(
.....:         {"min": group.min(), "max": group.max(),
.....:          "count": group.count(), "mean": group.mean()}
.....:     )
```

```
In [95]: grouped = frame.groupby(quartiles)
```

```
In [96]: grouped.apply(get_stats)
```

```
Out[96]:
```

		min	max	count	mean
data1					
(-2.956, -1.23]	data1	-2.949343	-1.230179	94	-1.658818
	data2	-3.399312	1.670835	94	-0.033333
(-1.23, 0.489]	data1	-1.228918	0.488675	598	-0.329524
	data2	-2.989741	3.260383	598	-0.002622
(0.489, 2.208]	data1	0.489965	2.200997	298	1.065727
	data2	-3.745356	2.954439	298	0.078249
(2.208, 3.928]	data1	2.212303	3.927528	10	2.644253
	data2	-1.929776	1.765640	10	0.024750

Имейте в виду, что тот же результат можно вычислить проще:

```
In [97]: grouped.agg(["min", "max", "count", "mean"])
```

```
Out[97]:
```

	data1				data2			
	min	max	count	mean	min	max	count	mean
data1								
(-2.956, -1.23]	-2.949343	-1.230179	94	-1.658818	-3.399312	1.670835	94	-0.033333
(-1.23, 0.489]	-1.228918	0.488675	598	-0.329524	-2.989741	3.260383	598	-0.002622
(0.489, 2.208]	0.489965	2.200997	298	1.065727	-3.745356	2.954439	298	0.078249
(2.208, 3.928]	2.212303	3.927528	10	2.644253	-1.929776	1.765640	10	0.024750

	mean
data1	
(-2.956, -1.23]	-0.033333
(-1.23, 0.489]	-0.002622
(0.489, 2.208]	0.078249
(2.208, 3.928]	0.024750

Это были интервалы одинаковой длины, а чтобы вычислить интервалы равного размера на основе выборочных квантилей, нужно использовать функцию `pandas.qcut`. Мы можем передать количество выборочных квантилей для вычисления интервалов (4) и `labels=False`, чтобы получать только индексы квантилей, а не интервалы:

```
In [98]: quartiles_samp = pd.qcut(frame["data1"], 4, labels=False)
```

```
In [99]: quartiles_samp.head()
```

```
Out[99]:
```

```
0    1
1    3
2    2
3    2
4    3
```

```
Name: data1, dtype: int64
```

```
In [100]: grouped = frame.groupby(quartiles_samp)
```

```
In [101]: grouped.apply(get_stats)
```

```
Out[101]:
```

		min	max	count	mean
data1					
0	data1	-2.949343	-0.685484	250	-1.212173

```

      data2 -3.399312  2.628441    250 -0.027045
1    data1 -0.683066 -0.030280    250 -0.368334
      data2 -2.630247  3.260383    250 -0.027845
2    data1 -0.027734  0.618965    250  0.295812
      data2 -3.056990  2.458842    250  0.014450
3    data1  0.623587  3.927528    250  1.248875
      data2 -3.745356  2.954439    250  0.115899

```

Пример: подстановка зависящих от группы значений вместо отсутствующих

Иногда отсутствующие данные требуется отфильтровать методом `dropna`, а иногда восполнить их, подставив либо фиксированное значение, либо значение, зависящее от данных. Для этой цели предназначен метод `fillna`; вот, например, как можно заменить отсутствующие значения средним:

```
In [102]: s = pd.Series(np.random.standard_normal(6))
```

```
In [103]: s[::2] = np.nan
```

```

In [104]: s
Out[104]:
0      NaN
1    0.227290
2      NaN
3   -2.153545
4      NaN
5   -0.375842
dtype: float64

```

```

In [105]: s.fillna(s.mean())
Out[105]:
0   -0.767366
1    0.227290
2   -0.767366
3   -2.153545
4   -0.767366
5   -0.375842
dtype: float64

```

А что делать, если подставляемое значение зависит от группы? Один из способов решить задачу – сгруппировать данные и вызвать метод `apply`, передав ему функцию, которая вызывает `fillna` для каждого блока данных. Ниже приведены данные о некоторых штатах США с разделением на восточные и западные:

```

In [106]: states = ["Ohio", "New York", "Vermont", "Florida",
.....:              "Oregon", "Nevada", "California", "Idaho"]

In [107]: group_key = ["East", "East", "East", "East",
.....:                  "West", "West", "West", "West"]

In [108]: data = pd.Series(np.random.standard_normal(8), index=states)

In [109]: data

```

```
Out[109]:
Ohio      0.329939
New York   0.981994
Vermont    1.105913
Florida    -1.613716
Oregon     1.561587
Nevada     0.406510
California 0.359244
Idaho      -0.614436
dtype: float64
```

Сделаем так, чтобы некоторые значения отсутствовали:

```
In [110]: data[["Vermont", "Nevada", "Idaho"]] = np.nan
```

```
In [111]: data
Out[111]:
Ohio      0.329939
New York   0.981994
Vermont      NaN
Florida    -1.613716
Oregon     1.561587
Nevada      NaN
California 0.359244
Idaho      NaN
dtype: float64
```

```
In [112]: data.groupby(group_key).size()
Out[112]:
East      4
West      4
dtype: int64
```

```
In [113]: data.groupby(group_key).count()
Out[113]:
East      3
West      2
dtype: int64
```

```
In [114]: data.groupby(group_key).mean()
Out[114]:
East    -0.100594
West     0.960416
dtype: float64
```

Чтобы подставить вместо отсутствующих значений групповые средние, нужно поступить так:

```
In [115]: def fill_mean(group):
.....:     return group.fillna(group.mean())

In [116]: data.groupby(group_key).apply(fill_mean)
Out[116]:
Ohio      0.329939
New York   0.981994
Vermont    -0.100594
Florida    -1.613716
Oregon     1.561587
```

```
Nevada      0.960416
California  0.359244
Idaho       0.960416
dtype: float64
```

Или, возможно, требуется подставлять вместо отсутствующих значений фиксированные, но зависящие от группы. Поскольку у групп есть автоматически устанавливаемый атрибут `name`, мы можем воспользоваться им:

```
In [117]: fill_values = {"East": 0.5, "West": -1}

In [118]: def fill_func(group):
.....:     return group.fillna(fill_values[group.name])

In [119]: data.groupby(group_key).apply(fill_func)
Out[119]:
Ohio      0.329939
New York  0.981994
Vermont   0.500000
Florida  -1.613716
Oregon    1.561587
Nevada    -1.000000
California 0.359244
Idaho     -1.000000
dtype: float64
```

Пример: случайная выборка и перестановка

Предположим, что требуется произвести случайную выборку (с возвращением или без) из большого набора данных для моделирования методом Монте-Карло или какой-то другой задачи. Существуют разные способы выборки, одни более эффективны, другие – менее; здесь мы воспользуемся методом `sample` для объекта `Series`.

Для демонстрации сконструируем колоду игральных карт:

```
# Hearts (черви), Spades (пики), Clubs (трефы), Diamonds (бубны)
suits = ["H", "S", "C", "D"]
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ["A"] + list(range(2, 11)) + ["J", "K", "Q"]
cards = []
for suit in suits:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

Теперь у нас есть объект `Series` длины 52, индекс которого содержит названия карт, а значения – ценность карт в блэкджеке и других играх (для простоты я присвоил тузу значение 1).

```
In [121]: deck.head(13)
Out[121]:
AH      1
2H      2
3H      3
4H      4
5H      5
```



```

6H      6
7H      7
8H      8
9H      9
10H     10
JH      10
KH      10
QH      10
dtype: int64

```

Исходя из сказанного выше, сдать пять карт из колоды можно следующим образом:

```

In [122]: def draw(deck, n=5):
.....:     return deck.sample(n)
In [123]: draw(deck)
Out[123]:
4D      4
QH      10
8S      8
7D      7
9C      9
dtype: int64

```

Пусть требуется выбрать по две случайные карты из каждой масти. Поскольку масть обозначается последним символом названия карты, то можно произвести по ней группировку и воспользоваться методом `apply`:

```

In [124]: def get_suit(card):
.....:     # последняя буква обозначает масть
.....:     return card[-1]

In [125]: deck.groupby(get_suit).apply(draw, n=2)
Out[125]:
C  6C      6
   KC      10
D  7D      7
   3D      3
H  7H      7
   9H      9
S  2S      2
   QS      10
dtype: int64

```

Можно было бы вместо этого передать параметр `group_keys=False`, чтобы отбросить внешний индекс мастей, оставив только выбранные карты:

```

In [126]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[126]:
AC      1
3C      3
5D      5
4D      4
10H     10
7H      7
QS      10
7S      7
dtype: int64

```

Пример: групповое взвешенное среднее и корреляция

Принцип разделения–применения–объединения, лежащий в основе `groupby`, позволяет легко выразить такие операции между столбцами `DataFrame` или двумя объектами `Series`, как вычисление группового взвешенного среднего. В качестве примера возьмем следующий набор данных, содержащий групповые ключи, значения и веса:

```
In [127]: df = pd.DataFrame({"category": ["a", "a", "a", "a",
.....:                                "b", "b", "b", "b"],
.....:                      "data": np.random.standard_normal(8),
.....:                      "weights": np.random.uniform(size=8)})
```

```
In [128]: df
Out[128]:
```

	category	data	weights
0	a	-1.691656	0.955905
1	a	0.511622	0.012745
2	a	-0.401675	0.137009
3	a	0.968578	0.763037
4	b	-1.818215	0.492472
5	b	0.279963	0.832908
6	b	-0.200819	0.658331
7	b	-0.217221	0.612009

Групповое взвешенное среднее по столбцу `category` равно:

```
In [129]: grouped = df.groupby("category")

In [130]: def get_wavg(group):
.....:     return np.average(group["data"], weights=group["weights"])

In [131]: grouped.apply(get_wavg)
Out[131]:
```

category	
a	-0.495807
b	-0.357273

dtype: float64

В качестве другого примера рассмотрим набор данных с сайта Yahoo! Finance, содержащий цены дня на некоторые акции и индекс S&P 500 (торговый код `SPX`):

```
In [132]: close_px = pd.read_csv("examples/stock_px.csv", parse_dates=True,
.....:                           index_col=0)

In [133]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
#   Column   Non-Null Count  Dtype
---  ---
0    AAPL    2214 non-null   float64
1    MSFT    2214 non-null   float64
2    XOM     2214 non-null   float64
3    SPX     2214 non-null   float64
dtypes: float64(4)
memory usage: 86.5 KB
```

```
In [134]: close_px.tail(4)
Out[134]:
```

	AAPL	MSFT	XOM	SPX
2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

Метод DataFrame `info()` здесь дает удобный способ получить сводную информацию о содержимом DataFrame.

Было бы интересно вычислить объект DataFrame, содержащий годовые корреляции между суточной доходностью (вычисленной по процентному изменению) и `SPX`. Один из способов решения этой задачи состоит в том, чтобы сначала создать функцию, которая вычисляет попарную корреляцию между каждым столбцом и столбцом `"SPX"`:

```
In [135]: def spx_corr(group):
.....:     return group.corrwith(group["SPX"])
```

Затем вычислим процентное изменение в `close_px` с помощью метода `pct_change()`:

```
In [136]: rets = close_px.pct_change().dropna()
```

И наконец, сгруппируем процентные изменения по годам, которые можно извлечь из метки строки однострочной функцией, возвращающей атрибут `year` метки `datetime`:

```
In [137]: def get_year(x):
.....:     return x.year

In [138]: by_year = rets.groupby(get_year)
```

```
In [139]: by_year.apply(spx_corr)
Out[139]:
```

	AAPL	MSFT	XOM	SPX
2003	0.541124	0.745174	0.661265	1.0
2004	0.374283	0.588531	0.557742	1.0
2005	0.467540	0.562374	0.631010	1.0
2006	0.428267	0.406126	0.518514	1.0
2007	0.508118	0.658770	0.786264	1.0
2008	0.681434	0.804626	0.828303	1.0
2009	0.707103	0.654902	0.797921	1.0
2010	0.710105	0.730118	0.839057	1.0
2011	0.691931	0.800996	0.859975	1.0

Разумеется, ничто не мешает вычислить корреляцию между столбцами. Ниже мы вычисляем годовую корреляцию между Apple и Microsoft:

```
In [140]: def corr_aapl_msft(group):
.....:     return group["AAPL"].corr(group["MSFT"])

In [141]: by_year.apply(corr_aapl_msft)
Out[141]:
```

2003	0.480868
2004	0.259024
2005	0.300093

```

2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64

```

Пример: групповая линейная регрессия

Следуя той же методике, что в предыдущем примере, мы можем применить `groupby` для выполнения более сложного статистического анализа на группах; главное, чтобы функция возвращала объект `pandas` или скалярное значение. Например, я могу определить функцию `regress` (воспользовавшись эконометрической библиотекой `statsmodels`), которая вычисляет регрессию методом обыкновенных наименьших квадратов для каждого блока данных:

```

import statsmodels.api as sm
def regress(data, yvar=None, xvars=None):
    Y = data[yvar]
    X = data[xvars]
    X["intercept"] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params

```

Пакет `statsmodels` можно установить с помощью `conda`, если он еще не установлен:

```
conda install statsmodels
```

Теперь для вычисления линейной регрессии `AAPL` от суточного оборота `SPX` по годам нужно написать:

```

In [143]: by_year.apply(regress, yvar="AAPL", xvars=["SPX"])
Out[143]:
      SPX  intercept
2003  1.195406    0.000710
2004  1.363463    0.004201
2005  1.766415    0.003246
2006  1.645496    0.000080
2007  1.198761    0.003438
2008  0.968016   -0.001110
2009  0.879103    0.002954
2010  1.052608    0.001261
2011  0.806605    0.001514

```

10.4. ГРУППОВЫЕ ПРЕОБРАЗОВАНИЯ И «РАЗВЕРНУТАЯ» ГРУППИРОВКА

В разделе 10.3 мы рассмотрели, как метод `apply` применяется в групповых операциях для выполнения преобразований. Имеется еще один встроенный метод, `transform`, который похож на `apply`, но налагает больше ограничений на то, какие функции можно использовать:

- функция может возвращать скалярное значение, которое будет уложено на группу;
- функция может возвращать объект такой же формы, как входная группа;
- функция не должна модифицировать свой вход.

Рассмотрим в качестве иллюстрации простой пример:

```
In [144]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
.....:                      'value': np.arange(12.)})
```

```
In [145]: df
Out[145]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

Вычислим групповые средние для каждого ключа:

```
In [146]: g = df.groupby('key')['value']
```

```
In [147]: g.mean()
Out[147]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

Но предположим, что вместо этого мы хотим получить объект Series такой же формы, как `df['value']`, в котором значения заменены средним, сгруппированным по 'key'. Мы можем передать методу `transform` функцию, которая вычисляет среднее для одной группы:

```
In [148]: def get_mean(group):
.....:     return group.mean()

In [149]: g.transform(get_mean)
Out[149]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
```

```

9      4.5
10     5.5
11     6.5
Name: value, dtype: float64

```

Для встроенных агрегатных функций мы можем передать их официальное название в виде строки, как в случае метода `agg` объекта `GroupBy`:

```

In [150]: g.transform('mean')
Out[150]:
0      4.5
1      5.5
2      6.5
3      4.5
4      5.5
5      6.5
6      4.5
7      5.5
8      6.5
9      4.5
10     5.5
11     6.5
Name: value, dtype: float64

```

Как и `apply`, `transform` готов работать с функциями, возвращающими `Series`, но результат должен быть такого же размера, как вход. Например, можно умножить каждую группу на 2, воспользовавшись вспомогательной функцией:

```

In [151]: def times_two(group):
.....:     return group * 2

In [152]: g.transform(times_two)
Out[152]:
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11    22.0
Name: value, dtype: float64

```

В качестве более сложного примера вычислим ранги элементов в каждой группе в порядке убывания:

```

In [153]: def get_ranks(group):
.....:     return group.rank(ascending=False)

In [154]: g.transform(get_ranks)
Out[154]:
0      4.0
1      4.0

```

```
2    4.0
3    3.0
4    3.0
5    3.0
6    2.0
7    2.0
8    2.0
9    1.0
10   1.0
11   1.0
```

```
Name: value, dtype: float64
```

Рассмотрим функцию группового преобразования, полученную комбинированием простых агрегатов:

```
In [155]: def normalize(x):
.....:     return (x - x.mean()) / x.std()
```

В данном случае `transform` и `apply` дают эквивалентные результаты:

```
In [156]: g.transform(normalize)
```

```
Out[156]:
```

```
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
```

```
Name: value, dtype: float64
```

```
In [157]: g.apply(normalize)
```

```
Out[157]:
```

```
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
```

```
Name: value, dtype: float64
```

Встроенные агрегатные функции типа `'mean'` или `'sum'` часто работают гораздо быстрее, чем метод `apply`. При использовании совместно с `transform` они выбирают «быстрый путь». Это позволяет выполнять так называемую развернутую групповую операцию:

```

In [158]: g.transform('mean')
Out[158]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
9    4.5
10   5.5
11   6.5
Name: value, dtype: float64

In [159]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')

In [160]: normalized
Out[160]:
0   -1.161895
1   -1.161895
2   -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64

```

Здесь мы выполняем арифметические операции между выходами нескольких операций GroupBy, вместо того чтобы написать функцию и передать ее методу `groupby(...).apply`. Именно это и понимается под «развертыванием».

Хотя развернутая групповая операция может включать несколько групповых агрегирований, выгода, достигаемая за счет векторизации операций, часто перевешивает.

10.5. Сводные таблицы и перекрестная табуляция

Сводная таблица – это средство обобщения данных, применяемое в электронных таблицах и других аналитических программах. Оно агрегирует таблицу по одному или нескольким ключам и строит другую таблицу, в которой одни групповые ключи расположены в строках, а другие – в столбцах. Библиотека `pandas` позволяет строить сводные таблицы с помощью описанного выше механизма `groupby` в сочетании с операциями изменения формы с применением иерархического индексирования. В классе `DataFrame` имеется метод `pivot_table`, а на верхнем уровне – функция `pandas.pivot_table`. Помимо удобного интерфейса к `groupby`, функция `pivot_table` еще умеет добавлять частичные итоги, которые называются *маргиналами*.

Вернемся к набору данных о чаевых и вычислим таблицу групповых средних (тип агрегирования по умолчанию, подразумеваемый `pivot_table`) по столбцам `day` и `smoker`, расположив их в строках:

```
In [161]: tips.head()
Out[161]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

```
In [162]: tips.pivot_table(index=["day", "smoker"])
Out[162]:
```

day	smoker	size	tip	tip_pct	total_bill
Fri	No	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

Это можно было бы легко сделать и непосредственно с помощью `groupby`, написав `tips.groupby(["day", "smoker"]).mean()`. Пусть теперь требуется вычислить среднее только для `tip_pct` и `size`, добавив еще группировку по `time`. Я помещу `smoker` в столбцы таблицы, а `time` и `day` – в строки:

```
In [163]: tips.pivot_table(index=["time", "day"], columns="smoker",
.....:                      values=["tip_pct", "size"])
Out[163]:
```

	time	day	size		tip_pct	
			No	Yes	No	Yes
Dinner	Fri	Fri	2.000000	2.222222	0.139622	0.165347
		Sat	2.555556	2.476190	0.158048	0.147906
	Sun	Sun	2.929825	2.578947	0.160113	0.187250
		Thur	2.000000	NaN	0.159744	NaN
Lunch	Fri	Fri	3.000000	1.833333	0.187735	0.188937
	Thur	Thur	2.500000	2.352941	0.160311	0.163863

Эту таблицу можно было бы дополнить, включив частичные итоги, для чего следует задать параметр `margins=True`. Тогда будут добавлены строка и столбец с меткой `All`, значениями в которых будут групповые статистики по всем данным на одном уровне.

```
In [164]: tips.pivot_table(index=["time", "day"], columns="smoker",
.....:                      values=["tip_pct", "size"], margins=True)
Out[164]:
```

	time	day	size		tip_pct	No	Yes	All
			No	Yes	All			
Dinner	Fri	Fri	2.000000	2.222222	0.139622	0.165347		
		Sat	2.555556	2.476190	0.158048	0.147906		
	Sun	Sun	2.929825	2.578947	0.160113	0.187250		
		Thur	2.000000	NaN	0.159744	NaN		
Lunch	Fri	Fri	3.000000	1.833333	0.187735	0.188937		
	Thur	Thur	2.500000	2.352941	0.160311	0.163863		

Dinner	Fri	2.000000	2.222222	2.166667	0.139622	0.165347	0.158916
	Sat	2.555556	2.476190	2.517241	0.158048	0.147906	0.153152
	Sun	2.929825	2.578947	2.842105	0.160113	0.187250	0.166897
	Thur	2.000000	NaN	2.000000	0.159744	NaN	0.159744
Lunch	Fri	3.000000	1.833333	2.000000	0.187735	0.188937	0.188765
	Thur	2.500000	2.352941	2.459016	0.160311	0.163863	0.161301
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

Здесь столбцы `All` содержат средние без учета того, является гость курящим или некурящим, а строка `All` – средние по обоим уровням группировки.

Для применения функции агрегирования, отличной от `mean`, передайте ее в именованном параметре `aggfunc`. Например, передача `"count"` или `len` даст перекрестную таблицу (счетчики или частоты) размеров групп (хотя `"count"` не учитывает значения `null`, встречающиеся в группах, а `len` учитывает):

```
In [165]: tips.pivot_table(index=["time", "smoker"], columns="day",
.....:                    values="tip_pct", aggfunc=len, margins=True)
Out[165]:
day      Fri  Sat  Sun  Thur  All
time  smoker
Dinner No    3.0  45.0  57.0    1.0  106
      Yes    9.0  42.0  19.0    NaN   70
Lunch  No    1.0   NaN   NaN   44.0   45
      Yes    6.0   NaN   NaN   17.0   23
All     19.0  87.0  76.0  62.0  244
```

Для восполнения отсутствующих комбинаций можно задать параметр `fill_value`:

```
In [166]: tips.pivot_table(index=["time", "size", "smoker"], columns="day",
.....:                    values="tip_pct", fill_value=0)
Out[166]:
day      size  smoker      Fri      Sat      Sun      Thur
time
Dinner  1    No      0.000000  0.137931  0.000000  0.000000
      1    Yes      0.000000  0.325733  0.000000  0.000000
      2    No      0.139622  0.162705  0.168859  0.159744
      2    Yes      0.171297  0.148668  0.207893  0.000000
      3    No      0.000000  0.154661  0.152663  0.000000
...
Lunch   3    Yes      0.000000  0.000000  0.000000  0.204952
      4    No      0.000000  0.000000  0.000000  0.138919
      4    Yes      0.000000  0.000000  0.000000  0.155410
      5    No      0.000000  0.000000  0.000000  0.121389
      6    No      0.000000  0.000000  0.000000  0.173706
[21 rows x 4 columns]
```

В табл. 10.2 приведена сводка аргументов метода `pivot_table`.

Таблица 10.2. Аргументы метода `pivot_table`

Параметр	Описание
<code>values</code>	Имя (или имена) одного или нескольких столбцов, по которым производится агрегирование. По умолчанию агрегируются все числовые столбцы
<code>index</code>	Имена столбцов или другие групповые ключи для группировки по строкам результирующей сводной таблицы
<code>columns</code>	Имена столбцов или другие групповые ключи для группировки по столбцам результирующей сводной таблицы
<code>aggfunc</code>	Функция агрегирования или список таких функций; по умолчанию <code>"mean"</code> . Можно задать произвольную функцию, допустимую в контексте <code>groupby</code>
<code>fill_value</code>	Чем заменять отсутствующие значения в результирующей таблице
<code>dropna</code>	Если <code>True</code> , не включать столбцы, в которых все значения отсутствуют
<code>margins</code>	Добавлять частичные итоги и общий итог по строкам и столбцам (по умолчанию <code>False</code>)
<code>margins_name</code>	Как назвать маргинальную строку (столбец), если передан параметр <code>margins=True</code> ; по умолчанию <code>"All"</code>
<code>observed</code>	Только для категориальных групповых ключей: если <code>True</code> , показывать только наблюдаемые в ключах категории, в противном случае показывать все категории

Перекрестная табуляция: `crosstab`

Перекрестная таблица (cross-tabulation, или для краткости `crosstab`) – частный случай сводной таблицы, в которой представлены частоты групп. Приведем пример:

```
In [167]: from io import StringIO

In [168]: data = """Sample Nationality Handedness
.....: 1   USA Right-handed
.....: 2   Japan Left-handed
.....: 3   USA Right-handed
.....: 4   Japan Right-handed
.....: 5   Japan Left-handed
.....: 6   Japan Right-handed
.....: 7   USA Right-handed
.....: 8   USA Left-handed
.....: 9   Japan Right-handed
.....: 10  USA Right-handed"""

In [169]: data = pd.read_table(StringIO(data), sep="\s+")

In [170]: data
Out[170]:
```

	Sample	Nationality	Handedness
0	1	USA	Right-handed
1	2	Japan	Left-handed
2	3	USA	Right-handed

3	4	Japan	Right-handed
4	5	Japan	Left-handed
5	6	Japan	Right-handed
6	7	USA	Right-handed
7	8	USA	Left-handed
8	9	Japan	Right-handed
9	10	USA	Right-handed

В ходе анализа-обследования мы могли бы обобщить эти данные по национальности и праворуконости/леворуконости. Для этой цели можно использовать метод `pivot_table`, но функция `pandas.crosstab` удобнее:

```
In [171]: pd.crosstab(data["Nationality"], data["Handedness"], margins=True)
Out[171]:
Handedness  Left-handed  Right-handed  All
Nationality
Japan                2             3    5
USA                  1             4    5
All                  3             7   10
```

Каждый из первых двух аргументов `crosstab` может быть массивом, объектом `Series` или списком массивов. Например, в случае данных о чаевых:

```
In [172]: pd.crosstab([tips["time"], tips["day"]], tips["smoker"], margins=True)
Out[172]:
smoker      No  Yes  All
time day
Dinner Fri    3    9   12
       Sat   45   42   87
       Sun   57   19   76
       Thur    1    0    1
Lunch  Fri    1    6    7
       Thur   44   17   61
All           151   93  244
```

10.5. ЗАКЛЮЧЕНИЕ

Уверенное владение средствами группировки, имеющимися в `pandas`, поможет вам как при очистке данных, так и в процессе моделирования или статистического анализа. В главе 13 мы рассмотрим дополнительные примеры использования `groupby` для реальных данных.

А в следующей главе обратимся к временным рядам.

Временные ряды

Временные ряды – важная разновидность структурированных данных. Они встречаются во многих областях, в том числе в финансах, экономике, экологии, нейронауках и физике. Любые результаты наблюдений или измерений в разные моменты времени образуют временной ряд. Для многих временных рядов характерна *фиксированная частота*, т. е. интервалы между соседними точками одинаковы – измерения производятся, например, один раз в 15 секунд, 5 минут или в месяц. Но временные ряды могут быть и *нерегулярными*, когда интервалы времени между соседними точками различаются. Как разметить временной ряд и обращаться к нему, зависит от приложения. Существуют следующие варианты:

Временные метки

Конкретные моменты времени.

Фиксированные периоды

Например, январь 2007 или весь 2010 год.

Временные интервалы

Обозначаются метками начала и конца. Периоды можно считать частными случаями интервалов.

Время эксперимента или истекшее время

Каждая временная метка измеряет время, прошедшее с некоторого начального момента. Например, результаты ежесекундного измерения диаметра печени с момента помещения теста в духовку.

В этой главе меня в основном будут интересовать временные ряды трех первых видов, хотя многие методы применимы и к экспериментальным временным рядам, когда индекс может содержать целые или вещественные значения, обозначающие время, прошедшее с начала эксперимента. Простейший вид временных рядов – ряды, индексированные временной меткой.



pandas поддерживает также индексы, построенные по приращению времени, это полезный способ представления времени эксперимента или истекшего времени. Такие индексы в этой книге не рассматриваются, но вы можете прочитать о них в документации (<http://pandas.pydata.org/>).

В библиотеке `pandas` имеется стандартный набор инструментов и алгоритмов для работы с временными рядами. Он позволяет эффективно работать с большими рядами, легко строить продольные и поперечные срезы, агрегировать и производить передискретизацию регулярных и нерегулярных временных рядов. Как нетрудно догадаться, многие из этих инструментов особенно полезны в финансовых и эконометрических приложениях, но никто не мешает применять их, например, к анализу журналов сервера.

Как и в других главах, прежде всего импортируем `NumPy` и `pandas`:

```
In [12]: import numpy as np
```

```
In [13]: import pandas as pd
```

11.1. Типы данных и инструменты, относящиеся к дате и времени

В стандартной библиотеке Python имеются типы данных для представления даты и времени, а также средства, относящиеся к календарю. Начинать изучение надо с модулей `datetime`, `time` и `calendar`. Особенно широко используется тип `datetime.datetime`, или просто `datetime`:

```
In [14]: from datetime import datetime
```

```
In [15]: now = datetime.now()
```

```
In [16]: now
```

```
Out[16]: datetime.datetime(2022, 8, 12, 9, 11, 337033)
```

```
In [17]: now.year, now.month, now.day
```

```
Out[17]: (2022, 8, 12)
```

В объекте типа `datetime` хранятся дата и время с точностью до микросекунды. Класс `datetime.timedelta`, или просто `timedelta`, представляет интервал времени между двумя объектами `datetime`:

```
In [18]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)
```

```
In [19]: delta
```

```
Out[19]: datetime.timedelta(days=926, seconds=56700)
```

```
In [20]: delta.days
```

```
Out[20]: 926
```

```
In [21]: delta.seconds
```

```
Out[21]: 56700
```

Можно прибавить (или вычесть) объект типа `timedelta` или его произведение на целое число к объекту `datetime` и получить в результате новый объект того же типа, представляющий соответственно сдвинутый момент времени:

```
In [22]: from datetime import timedelta
```

```
In [23]: start = datetime(2011, 1, 7)
```

```
In [24]: start + timedelta(12)
Out[24]: datetime.datetime(2011, 1, 19, 0, 0)

In [25]: start - 2 * timedelta(12)
Out[25]: datetime.datetime(2010, 12, 14, 0, 0)
```

Сводка типов данных в модуле `datetime` приведена в табл. 11.1. Хотя в этой главе речь пойдет преимущественно о типах данных в `pandas` и высокоуровневых операциях с временными рядами, вы без сомнения встретите основанные на `datetime` типы и во многих других приложениях, написанных на Python.

Таблица 11.1. Типы в модуле `datetime`

Тип	Описание
<code>date</code>	Хранит дату (год, месяц, день) по григорианскому календарю
<code>time</code>	Хранит время суток (часы, минуты, секунды и микросекунды)
<code>datetime</code>	Хранит дату и время
<code>timedelta</code>	Представляет разность между двумя значениями типа <code>datetime</code> (дни, секунды и микросекунды)
<code>tzinfo</code>	Базовый тип для хранения информации о часовых поясах

Преобразование между строкой и `datetime`

Объекты типа `datetime` и входящего в `pandas` типа `Timestamp`, с которым мы вскоре познакомимся, можно представить в виде отформатированной строки с помощью метода `str` или `strftime`, которому передается спецификация формата:

```
In [26]: stamp = datetime(2011, 1, 3)

In [27]: str(stamp)
Out[27]: '2011-01-03 00:00:00'

In [28]: stamp.strftime("%Y-%m-%d")
Out[28]: '2011-01-03'
```

Полный перечень форматных кодов приведен в табл. 11.2.

Таблица 11.2. Спецификации формата даты в классе `datetime` (совместимо со стандартом ISO C89)

Спецификатор	Описание
<code>%Y</code>	Год с четырьмя цифрами
<code>%y</code>	Год с двумя цифрами
<code>%m</code>	Номер месяца с двумя цифрами [01, 12]
<code>%d</code>	Номер дня с двумя цифрами [01, 31]
<code>%H</code>	Час (в 24-часовом формате) [00, 23]

Кроме того, этот метод умеет обрабатывать значения, которые следует считать отсутствующими (`None`, пустая строка и т. д.):

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dtype='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isna(idx)
Out[38]: array([False, False,  True])
```

NaT (Not a Time – не время) – применяемое в pandas значение для индикации отсутствующей временной метки.



Класс `dateutil.parser` – полезный, но не идеальный инструмент. В частности, он распознает строки, которые не на всякий взгляд являются датами. Например, строка "42" будет разобрана как текущая календарная дата в 2042 году.

У объектов `datetime` имеется также ряд зависимых от локали параметров форматирования для других стран и языков. Например, сокращенные названия месяцев в системе с немецкой или французской локалью будут не такие, как в системе с английской локалью. Полный перечень см. в табл. 11.3.

Таблица 11.3. Спецификации формата даты, зависящие от локали

Спецификатор	Описание
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Полная дата и время, например «Tue 01 May 2012 04:20:57 PM»
%p	Локализованный эквивалент AM или PM
%x	Дата в формате, соответствующем локали. Например, в США 1 мая 2012 будет представлена в виде «05/01/2012»
%X	Время в формате, соответствующем локали, например «04:24:12 PM»

11.2. Основы работы с временными рядами

Самый простой вид временного ряда в pandas – объект `Series`, индексированный временными метками, которые часто представляются внешними по отношению к pandas Python-строками или объектами `datetime`:

```
In [39]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
....:             datetime(2011, 1, 7), datetime(2011, 1, 8),
....:             datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [40]: ts = pd.Series(np.random.standard_normal(6), index=dates)
```

```
In [41]: ts
Out[41]:
2011-01-02    -0.204708
2011-01-05     0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
2011-01-12     1.393406
dtype: float64
```

Под капотом объекты `datetime` помещаются в объект типа `DatetimeIndex`:

```
In [42]: ts.index
Out[42]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

Как и для других объектов `Series`, арифметические операции над временными рядами с различными индексами автоматически приводят к выравниванию дат:

```
In [43]: ts + ts[::-2]
Out[43]:
2011-01-02    -0.409415
2011-01-05         NaN
2011-01-07    -1.038877
2011-01-08         NaN
2011-01-10     3.931561
2011-01-12         NaN
dtype: float64
```

Напомним, что конструкция `ts[::-2]` выбирает каждый второй элемент `ts`.

В `pandas` временные метки хранятся в типе данных NumPy `datetime64` с наносекундным разрешением:

```
In [44]: ts.index.dtype
Out[44]: dtype('<M8[ns]')
```

Скалярные значения в индексе `DatetimeIndex` – это объекты `pandas` типа `Timestamp`:

```
In [45]: stamp = ts.index[0]

In [46]: stamp
Out[46]: Timestamp('2011-01-02 00:00:00')
```

Объект типа `pandas.Timestamp` можно использовать всюду, где допустим объект `datetime`. Обратное, однако, неверно, потому что в `pandas.Timestamp` можно хранить данные с наносекундной точностью, а в `datetime` – только с микросекундной. Кроме того, в `pandas.Timestamp` можно хранить информацию о частоте (если име-

ется), и он умеет преобразовывать часовые пояса и производить другие манипуляции. Подробнее об этом будет рассказано в разделе 11.4.

Индексирование, выборка, подмножества

TimeSeries – подкласс Series и потому ведет себя точно так же по отношению к индексированию и выборке данных по метке:

```
In [47]: stamp = ts.index[2]
```

```
In [48]: ts[stamp]
Out[48]: -0.5194387150567381
```

В качестве дополнительного удобства можно передать строку, допускающую интерпретацию в виде даты:

```
In [49]: ts["2011-01-10"]
Out[49]: 1.9657805725027142
```

Для выборки срезов из длинных временных рядов можно передать только год или год и месяц (тип `pandas.date_range` обсуждается ниже):

```
In [50]: longer_ts = pd.Series(np.random.standard_normal(1000),
....: index=pd.date_range("2000-01-01", periods=1000))
```

```
In [51]: longer_ts
Out[51]:
2000-01-01    0.092908
2000-01-02    0.281746
2000-01-03    0.769023
2000-01-04    1.246435
2000-01-05    1.007189
...
2002-09-22    0.930944
2002-09-23   -0.811676
2002-09-24   -1.830156
2002-09-25   -0.138730
2002-09-26    0.334088
Freq: D, Length: 1000, dtype: float64
```

```
In [52]: longer_ts["2001"]
Out[52]:
2001-01-01    1.599534
2001-01-02    0.474071
2001-01-03    0.151326
2001-01-04   -0.542173
2001-01-05   -0.475496
...
2001-12-27    0.057874
2001-12-28   -0.433739
2001-12-29    0.092698
2001-12-30   -1.397820
2001-12-31    1.457823
Freq: D, Length: 365, dtype: float64
```

Здесь строка '2001' интерпретируется как год, и выбирается такой период времени. Это будет работать и тогда, когда указан месяц:

```
In [53]: longer_ts["2001-05"]
Out[53]:
2001-05-01    -0.622547
2001-05-02     0.936289
2001-05-03     0.750018
2001-05-04    -0.056715
2001-05-05     2.300675
...
2001-05-27     0.235477
2001-05-28     0.111835
2001-05-29    -1.251504
2001-05-30    -2.949343
2001-05-31     0.634634
Freq: D, Length: 31, dtype: float64
```

Выборка срезов с помощью объектов `datetime` тоже работает:

```
In [54]: ts[datetime(2011, 1, 7):]
Out[54]:
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
2011-01-12     1.393406
dtype: float64

In [55]: ts[datetime(2011, 1, 7):datetime(2011, 1, 10)]
Out[55]:
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
dtype: float64
```

Поскольку временные ряды обычно упорядочены хронологически, при формировании срезов можно указывать временные метки, отсутствующие в самом ряду, чтобы выполнить запрос по диапазону:

```
In [56]: ts
Out[56]:
2011-01-02    -0.204708
2011-01-05     0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
2011-01-12     1.393406
dtype: float64

In [57]: ts["2011-01-06":"2011-01-11"]
Out[57]:
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
dtype: float64
```

Как и раньше, можно задать дату в виде строки, объекта `datetime` или `Timestamp`. Напомню, что такое формирование среза порождает представление исходного временного ряда, как и для массивов NumPy. Это означает, что никакие данные не копируются, а модификация среза отражается на исходных данных.

Существует эквивалентный метод экземпляра `truncate`, который возвращает срез Series между двумя датами:

```
In [58]: ts.truncate(after="2011-01-09")
Out[58]:
2011-01-02    -0.204708
2011-01-05     0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
dtype: float64
```

Все вышеперечисленное справедливо и для объекта DataFrame, индексированного своими строками:

```
In [59]: dates = pd.date_range("2000-01-01", periods=100, freq="W-WED")

In [60]: long_df = pd.DataFrame(np.random.standard_normal((100, 4)),
.....:                        index=dates,
.....:                        columns=["Colorado", "Texas",
.....:                               "New York", "Ohio"])

In [61]: long_df.loc["2001-05"]
Out[61]:
```

	Colorado	Texas	New York	Ohio
2001-05-02	-0.006045	0.490094	-0.277186	-0.707213
2001-05-09	-0.560107	2.735527	0.927335	1.513906
2001-05-16	0.538600	1.273768	0.667876	-0.969206
2001-05-23	1.676091	-0.817649	0.050188	1.951312
2001-05-30	3.260383	0.963301	1.201206	-1.852001

Временные ряды с неуникальными индексами

В некоторых приложениях бывает, что несколько результатов измерений имеют одну и ту же временную метку, например:

```
In [62]: dates = pd.DatetimeIndex(["2000-01-01", "2000-01-02", "2000-01-02",
.....:                             "2000-01-02", "2000-01-03"])

In [63]: dup_ts = pd.Series(np.arange(5), index=dates)

In [64]: dup_ts
Out[64]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

Узнать о том, что индекс не уникален, можно, опросив его свойство `is_unique`:

```
In [65]: dup_ts.index.is_unique
Out[65]: False
```

При доступе к такому временному ряду по индексу будет возвращено либо скалярное значение, либо срез – в зависимости от того, является временная метка уникальной или нет:

```
In [66]: dup_ts["2000-01-03"] # метка уникальна
Out[66]: 4

In [67]: dup_ts["2000-01-02"] # метка повторяется
Out[67]:
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

Пусть требуется агрегировать данные с неуникальными временными метками. Одно из возможных решений – воспользоваться методом `groupby` с параметром `level=0` (один-единственный уровень):

```
In [68]: grouped = dup_ts.groupby(level=0)

In [69]: grouped.mean()
Out[69]:
2000-01-01    0.0
2000-01-02    2.0
2000-01-03    4.0
dtype: float64

In [70]: grouped.count()
Out[70]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

11.3. Диапазоны дат, частоты и сдвиг

Вообще говоря, временные ряды `pandas` не предполагаются регулярными, т. е. частота в них не фиксирована. Для многих приложений это вполне приемлемо. Но иногда желательно работать с постоянной частотой, например день, месяц, 15 минут, даже если для этого приходится вставлять в ряд отсутствующие значения. По счастью, `pandas` поддерживает полный набор частот и средства для передискретизации (подробно обсуждается в разделе 11.6), выведения частот и генерации диапазонов дат с фиксированной частотой. Например, временной ряд из нашего примера можно преобразовать в ряд с частотой один день с помощью метода `resample`:

```
In [71]: ts
Out[71]:
2011-01-02    -0.204708
2011-01-05     0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
2011-01-12     1.393406
dtype: float64

In [72]: resampler = ts.resample("D")

In [73]: resampler
Out[73]: <pandas.core.resample.DatetimeIndexResampler object at 0x7febd896bc40>
```

Строка 'D' интерпретируется как частота раз в сутки (daily).

Преобразование частоты, или *передискретизация*, – настолько обширная тема, что мы посвятим ей отдельный раздел 11.6 ниже. А сейчас я покажу, как работать с базовой частотой и кратными ей.

Генерирование диапазонов дат

Раньше я уже пользовался методом `pandas.date_range` без объяснений, и вы, наверное, догадались, что он порождает объект `DatetimeIndex` указанной длины с определенной частотой:

```
In [74]: index = pd.date_range("2012-04-01", "2012-06-01")

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
                '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
                '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
                '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
                '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
                '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
                '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
                '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
                '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
                '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
                '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
                '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
                '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
                '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
                '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
                '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

По умолчанию метод `date_range` генерирует временные метки с частотой один день. Если вы передаете ему только начальную или конечную дату, то должны задать также количество генерируемых периодов:

```
In [76]: pd.date_range(start="2012-04-01", periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
                '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
                '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
                '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
                '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end="2012-06-01", periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
                '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
                '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
                '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
                '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

Начальная и конечная даты определяют строгие границы для сгенерированного индекса по датам. Например, если требуется индекс по датам, содержа-

щий последний рабочий день каждого месяца, то следует передать в качестве частоты значение 'BM', и тогда будут включены только даты, попадающие внутрь или на границу интервала:

```
In [78]: pd.date_range("2000-01-01", "2000-12-01", freq="BM")
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

Таблица 11.4. Базовые частоты временных рядов (список неполный)

Обозначение	Тип смещения	Описание
D	Day	Ежедневно
B	BusinessDay	Каждый рабочий день
H	Hour	Ежечасно
T или min	Minute	Ежеминутно
S	Second	Ежесекундно
L или ms	Milli	Каждую миллисекунду
U	Micro	Каждую микросекунду
M	MonthEnd	Последний календарный день месяца
BM	BusinessMonthEnd	Последний рабочий день месяца
MS	MonthBegin	Первый календарный день месяца
BMS	BusinessMonthBegin	Первый рабочий день месяца
W-MON, W-TUE, ...	Week	Еженедельно в указанный день: MON, TUE, WED, THU, FRI, SAT, SUN
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Указанный день первой, второй, третьей или четвертой недели месяца. Например, WOM-3FRI означает третью пятницу каждого месяца
Q-JAN, Q-FEB, ...	QuarterEnd	Ежеквартально с привязкой к последнему календарному дню каждого месяца, считая, что год заканчивается в указанном месяце: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Ежеквартально с привязкой к последнему рабочему дню каждого месяца, считая, что год заканчивается в указанном месяце
QS-JAN, QS-FEB, ...	QuarterBegin	Ежеквартально с привязкой к первому календарному дню каждого месяца, считая, что год заканчивается в указанном месяце

Обозначение	Тип смещения	Описание
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Ежеквартально с привязкой к первому рабочему дню каждого месяца, считая, что год заканчивается в указанном месяце
A-JAN, A-FEB, ...	YearEnd	Ежегодно с привязкой к последнему календарному дню указанного месяца: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
BA-JAN, BA-FEB, ...	BusinessYearEnd	Ежегодно с привязкой к последнему рабочему дню указанного месяца
AS-JAN, AS-FEB, ...	YearBegin	Ежегодно с привязкой к первому календарному дню указанного месяца
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Ежегодно с привязкой к первому рабочему дню указанного месяца

По умолчанию метод `pandas.date_range` сохраняет время (если оно было задано) начальной и конечной временной метки:

```
In [79]: pd.date_range("2012-05-02 12:56:31", periods=5)
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
              '2012-05-04 12:56:31', '2012-05-05 12:56:31',
              '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

Иногда начальная или конечная дата содержит время, но требуется сгенерировать нормализованный набор временных меток, в которых время совпадает с полуночью. Для этого задайте параметр `normalize`:

```
In [80]: pd.date_range("2012-05-02 12:56:31", periods=5, normalize=True)
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
              '2012-05-06'],
              dtype='datetime64[ns]', freq='D')
```

Частоты и смещения дат

Частота в `pandas` состоит из базовой частоты и кратности. Базовая частота обычно обозначается строкой, например 'M' означает раз в месяц, а 'H' – раз в час. Для каждой базовой частоты определен объект, называемый *смещением даты* (date offset). Так, частоту «раз в час» можно представить классом `Hour`:

```
In [81]: from pandas.tseries.offsets import Hour, Minute

In [82]: hour = Hour()

In [83]: hour
Out[83]: <Hour>
```

Для определения кратности смещения нужно задать целое число:

```
In [84]: four_hours = Hour(4)
```

```
In [85]: four_hours
Out[85]: <4 * Hours>
```

В большинстве приложений не приходится создавать такие объекты явно, достаточно использовать их строковые обозначения вида 'H' или '4H'. Наличие целого числа перед базовой частотой создает кратную частоту:

```
In [86]: pd.date_range("2000-01-01", "2000-01-03 23:59", freq="4H")
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

Операция сложения позволяет объединить несколько смещений:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

Можно также задать частоту в виде строки '1h30min', что приводит к тому же результату, что и выше:

```
In [88]: pd.date_range("2000-01-01", periods=10, freq="1h30min")
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
               '2000-01-01 09:00:00', '2000-01-01 10:30:00',
               '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

Некоторые частоты описывают неравноотстоящие моменты времени. Например, значения частот 'M' (конец календарного месяца) и 'BМ' (последний рабочий день месяца) зависят от числа дней в месяце, а в последнем случае также от того, заканчивается месяц рабочим или выходным днем. За неимением лучшего термина я называю такие смещения *привязанными*.

В табл. 11.4 перечислены имеющиеся в pandas коды частот и классы смещений дат.



Пользователь может определить собственный класс частоты для реализации логики работы с датами, отсутствующей в pandas, однако подробности этого процесса выходят за рамки книги.

Даты, связанные с неделей месяца

Полезный класс частот – «неделя месяца», обозначается строкой, начинающейся с **WOM**. Он позволяет получить, например, третью пятницу каждого месяца:

```
In [89]: monthly_dates = pd.date_range("2012-01-01", "2012-09-01", freq="WOM-3FRI")
Out[89]:
In [90]: list(monthly_dates)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

Сдвиг данных (с опережением и с запаздыванием)

Под «сдвигом» понимается перемещение данных назад и вперед по временной оси. У объектов Series и DataFrame имеется метод `shift` для «наивного» сдвига в обе стороны без модификации индекса:

```
In [91]: ts = pd.Series(np.random.standard_normal(4),
.....:                  index=pd.date_range("2000-01-01", periods=4, freq="M"))

In [92]: ts
Out[92]:
2000-01-31    -0.066748
2000-02-29     0.838639
2000-03-31    -0.117388
2000-04-30    -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31    -0.066748
2000-04-30     0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31    -0.117388
2000-02-29    -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

При таком сдвиге отсутствующие данные вставляются в начало или в конец временного ряда.

Типичное применение `shift` – вычисление относительных изменений временного ряда или нескольких временных рядов и представление их в виде столбцов объекта DataFrame. Это выражается следующим образом:

```
ts / ts.shift(1) - 1
```

Поскольку наивный сдвиг не изменяет индекс, некоторые данные отбрасываются. Но если известна частота, то ее можно передать методу `shift`, чтобы сдвинуть вперед временные метки, а не сами данные:

```
In [95]: ts.shift(2, freq="M")
Out[95]:
2000-03-31    -0.066748
2000-04-30     0.838639
2000-05-31    -0.117388
2000-06-30    -0.517795
Freq: M, dtype: float64
```

Можно указывать и другие частоты, что позволяет гибко смещать данные в прошлое и в будущее:

```
In [96]: ts.shift(3, freq="D")
Out[96]:
2000-02-03    -0.066748
2000-03-03     0.838639
2000-04-03    -0.117388
2000-05-03    -0.517795
dtype: float64

In [97]: ts.shift(1, freq="90T")
Out[97]:
2000-01-31 01:30:00    -0.066748
2000-02-29 01:30:00     0.838639
2000-03-31 01:30:00    -0.117388
2000-04-30 01:30:00    -0.517795
dtype: float64
```

Здесь `T` обозначает минуты. Отметим, что параметр `freq` задает смещение, применяемое к временным меткам, но истинная частота данных (если таковая присутствует) при этом не изменяется.

Сдвиг дат с помощью смещений

Смещения дат можно также использовать совместно с объектами `datetime` и `Timestamp`:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd

In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

В случае привязанного смещения, например `MonthEnd`, первое сложение с ним *продвинет* дату до следующей даты с соответствующей привязкой:

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')

In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Привязанные смещения можно использовать и для явного сдвига даты вперед и назад с помощью методов `rollforward` и `rollback` соответственно:

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

У смещений дат есть интересное применение совместно с функцией `groupby`:

```
In [106]: ts = pd.Series(np.random.standard_normal(20),
.....: index=pd.date_range("2000-01-15", periods=20, freq="4D")
)
In [107]: ts
Out[107]:
2000-01-15    -0.116696
2000-01-19     2.389645
2000-01-23    -0.932454
2000-01-27    -0.229331
2000-01-31    -1.140330
2000-02-04     0.439920
2000-02-08    -0.823758
2000-02-12    -0.520930
2000-02-16     0.350282
2000-02-20     0.204395
2000-02-24     0.133445
2000-02-28     0.327905
2000-03-03     0.072153
2000-03-07     0.131678
2000-03-11    -1.297459
2000-03-15     0.997747
2000-03-19     0.870955
2000-03-23    -0.991253
2000-03-27     0.151699
2000-03-31     1.266151
Freq: 4D, dtype: float64

In [108]: ts.groupby(MonthEnd()).rollforward().mean()
Out[108]:
2000-01-31    -0.005833
2000-02-29     0.015894
2000-03-31     0.150209
dtype: float64
```

Разумеется, то же самое можно проделать проще и быстрее с помощью метода `resample` (мы обсудим его в разделе 11.6 ниже):

```
In [109]: ts.resample("M").mean()
Out[109]:
2000-01-31    -0.005833
2000-02-29     0.015894
2000-03-31     0.150209
Freq: M, dtype: float64
```

11.4. ЧАСОВЫЕ ПОЯСА

Работа с часовыми поясами традиционно считается одной из самых неприятных сторон манипулирования временными рядами. Поэтому многие пользователи предпочитают иметь дело с временными рядами в *координированном универсальном времени (UTC)*, не зависящем от географического местоположения международном стандарте. Часовые пояса выражаются в виде смещений от UTC; например, в Нью-Йорке время отстает от UTC на 4 часа в летний период и на 5 часов в остальное время года.

В Python информация о часовых поясах берется из сторонней библиотеки `pytz` (ее можно установить с помощью `pip` или `conda`), которая является оберткой вокруг *базы данных Олсона*, где собраны все сведения о мировых часовых поясах. Это особенно важно для исторических данных, потому что даты перехода на летнее время (и даже смещения от UTC) многократно менялись по прихоти местных правительств. В США даты перехода на летнее время с 1900 года менялись много раз!

Подробные сведения о библиотеке `pytz` можно найти в документации к ней. Но поскольку `pandas` инкапсулирует функциональность `pytz`, то можете спокойно игнорировать весь ее API, кроме названий часовых поясов. Поскольку `pandas` в любом случае зависит от библиотеки `pytz`, устанавливать ее отдельно необязательно. А названия поясов можно узнать как интерактивно, так и из документации:

```
In [110]: import pytz

In [111]: pytz.common_timezones[-5:]
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
Получить объект часового пояса от pytz позволяет функция pytz.timezone:
In [112]: tz = pytz.timezone("America/New_York")

In [113]: tz
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Методы из библиотеки `pandas` принимают как названия часовых зон, так и эти объекты.

Локализация и преобразование

По умолчанию временные ряды в `pandas` не учитывают часовые пояса. Рассмотрим следующий ряд:

```
In [114]: dates = pd.date_range("2012-03-09 09:30", periods=6)

In [115]: ts = pd.Series(np.random.standard_normal(len(dates)), index=dates)

In [116]: ts
Out[116]:
2012-03-09 09:30:00    -0.202469
2012-03-10 09:30:00     0.050718
2012-03-11 09:30:00     0.639869
2012-03-12 09:30:00     0.597594
2012-03-13 09:30:00    -0.797246
2012-03-14 09:30:00     0.472879
Freq: D, dtype: float64
```

Поле `tz` в индексе равно `None`:

```
In [117]: print(ts.index.tz)
None
```

Но при генерировании диапазонов дат можно и указать часовой пояс:

```
In [118]: pd.date_range("2012-03-09 09:30", periods=10, tz="UTC")
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

Для преобразования даты из инвариантного формата в локализованный (т. е. интерпретации в конкретном часовом поясе) служит метод `tz_localize`:

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00    -0.202469
2012-03-10 09:30:00     0.050718
2012-03-11 09:30:00     0.639869
2012-03-12 09:30:00     0.597594
2012-03-13 09:30:00    -0.797246
2012-03-14 09:30:00     0.472879
Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize("UTC")
```

```
In [121]: ts_utc
Out[121]:
2012-03-09 09:30:00+00:00    -0.202469
2012-03-10 09:30:00+00:00     0.050718
2012-03-11 09:30:00+00:00     0.639869
2012-03-12 09:30:00+00:00     0.597594
2012-03-13 09:30:00+00:00    -0.797246
2012-03-14 09:30:00+00:00     0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

После локализации временного ряда для его преобразования в другой часовой пояс нужно вызвать метод `tz_convert`:

```
In [123]: ts_utc.tz_convert("America/New_York")
Out[123]:
2012-03-09 04:30:00-05:00    -0.202469
2012-03-10 04:30:00-05:00     0.050718
2012-03-11 05:30:00-04:00     0.639869
2012-03-12 05:30:00-04:00     0.597594
2012-03-13 05:30:00-04:00    -0.797246
```

```
2012-03-14 05:30:00-04:00    0.472879
Freq: D, dtype: float64
```

Приведенный выше временной ряд охватывает дату перехода на летнее время в часовом поясе `America/New_York`, мы могли бы локализовать его для часового пояса восточного побережья США, а затем преобразовать, скажем, в UTC или в берлинское время:

```
In [124]: ts_eastern = ts.tz_localize("America/New_York")

In [125]: ts_eastern.tz_convert("UTC")
Out[125]:
2012-03-09 14:30:00+00:00    -0.202469
2012-03-10 14:30:00+00:00     0.050718
2012-03-11 13:30:00+00:00     0.639869
2012-03-12 13:30:00+00:00     0.597594
2012-03-13 13:30:00+00:00    -0.797246
2012-03-14 13:30:00+00:00     0.472879
dtype: float64

In [126]: ts_eastern.tz_convert("Europe/Berlin")
Out[126]:
2012-03-09 15:30:00+01:00    -0.202469
2012-03-10 15:30:00+01:00     0.050718
2012-03-11 14:30:00+01:00     0.639869
2012-03-12 14:30:00+01:00     0.597594
2012-03-13 14:30:00+01:00    -0.797246
2012-03-14 14:30:00+01:00     0.472879
dtype: float64
tz_localize и tz_convert являются также методами экземпляра DatetimeIndex:
In [127]: ts.index.tz_localize("Asia/Shanghai")
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq=None)
```



При локализации наивных временных меток проверяется также однозначность и существование моментов времени в окрестности даты перехода на летнее время.

Операции над объектами `Timestamp` с учетом часового пояса

По аналогии с временными рядами и диапазонами дат можно локализовать и отдельные объекты `Timestamp`, включив в них информацию о часовом поясе, а затем преобразовывать из одного пояса в другой:

```
In [128]: stamp = pd.Timestamp("2011-03-12 04:00")

In [129]: stamp_utc = stamp.tz_localize("utc")

In [130]: stamp_utc.tz_convert("America/New_York")
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York')
```


Часовой пояс можно задать и при создании объекта `Timestamp`:

```
In [131]: stamp_moscow = pd.Timestamp("2011-03-12 04:00", tz="Europe/Moscow")

In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

В объектах `Timestamp`, учитывающих часовой пояс, хранится временной штамп UTC в виде числа секунд от «эпохи» UNIX (1 января 1970); это значение инвариантно относительно преобразования из одного пояса в другой:

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert("America/New_York").value
Out[134]: 1299902400000000000
```

При выполнении арифметических операций над объектами `pandas DateOffset` всюду, где возможно, учитывается переход на летнее время. Ниже мы конструируем временные метки, соответствующие моментам, которые предшествуют переходу на летнее время и возврату в обычный режим. Сначала за 30 минут до перехода:

```
In [135]: stamp = pd.Timestamp("2012-03-11 01:30", tz="US/Eastern")

In [136]: stamp
Out[136]: Timestamp('2012-03-11 01:30:00-0500', tz='US/Eastern')

In [137]: stamp + Hour()
Out[137]: Timestamp('2012-03-11 03:30:00-0400', tz='US/Eastern')
Затем за 90 минут до перехода с летнего времени на обычное:
In [138]: stamp = pd.Timestamp("2012-11-04 00:30", tz="US/Eastern")

In [139]: stamp
Out[139]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')

In [140]: stamp + 2 * Hour()
Out[140]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

Операции над датами из разных часовых поясов

Если комбинируются два временных ряда с разными часовыми поясами, то в результате получится UTC. Поскольку во внутреннем представлении временные метки хранятся в UTC, то операция не требует никаких преобразований:

```
In [141]: dates = pd.date_range("2012-03-07 09:30", periods=10, freq="B")

In [142]: ts = pd.Series(np.random.standard_normal(len(dates)), index=dates)

In [143]: ts
Out[143]:
2012-03-07 09:30:00    0.522356
2012-03-08 09:30:00   -0.546348
2012-03-09 09:30:00   -0.733537
2012-03-12 09:30:00    1.302736
2012-03-13 09:30:00    0.022199
2012-03-14 09:30:00    0.364287
```

```

2012-03-15 09:30:00    -0.922839
2012-03-16 09:30:00     0.312656
2012-03-19 09:30:00    -1.128497
2012-03-20 09:30:00    -0.333488
Freq: B, dtype: float64

```

```
In [144]: ts1 = ts[:7].tz_localize("Europe/London")
```

```
In [145]: ts2 = ts1[2:].tz_convert("Europe/Moscow")
```

```
In [146]: result = ts1 + ts2
```

```
In [147]: result.index
```

```
Out[147]:
```

```

DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
              '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
              '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
              '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq=None)

```

Операции между объектами, учитывающими и не учитывающими часовой пояс, не поддерживаются и приводят к исключению.

11.5. ПЕРИОДЫ И АРИФМЕТИКА ПЕРИОДОВ

Периоды – это промежутки времени: дни, месяцы, кварталы, годы. Этот тип данных представлен классом `pandas.Period`, конструктор которого принимает строку или число и поддерживаемую частоту из табл. 11.4:

```
In [148]: p = pd.Period("2011", freq="A-DEC")
```

```
In [149]: p
```

```
Out[149]: Period('2011', 'A-DEC')
```

В данном случае объект `Period` представляет промежуток времени от 1 января 2007 до 31 декабря 2011 включительно. Сложение и вычитание периода и целого числа дают тот же результат, что сдвиг на величину, кратную частоте периода:

```
In [150]: p + 5
```

```
Out[150]: Period('2016', 'A-DEC')
```

```
In [151]: p - 2
```

```
Out[151]: Period('2009', 'A-DEC')
```

Если у двух периодов одинаковая частота, то их разностью является количество единиц между ними, выраженное в виде смещения даты:

```
In [152]: pd.Period("2014", freq="A-DEC") - p
```

```
Out[152]: <3 * YearEnds: month=12>
```

Регулярные диапазоны периодов строятся с помощью функции `period_range`:

```
In [153]: periods = pd.period_range("2000-01-01", "2000-06-30", freq="M")
```

```
In [154]: periods
```

```

Out[154]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '2000-06'],
                      dtype='period[M]')

```

В классе `PeriodIndex` хранится последовательность периодов, он может служить осевым индексом в любой структуре данных `pandas`:

```
In [155]: pd.Series(np.random.standard_normal(6), index=periods)
Out[155]:
2000-01    -0.514551
2000-02    -0.559782
2000-03    -0.783408
2000-04    -1.797685
2000-05    -0.172670
2000-06     0.680215
Freq: M, dtype: float64
```

Если имеется массив строк, то можно также использовать класс `PeriodIndex`, значениями которого являются периоды:

```
In [156]: values = ["2001Q3", "2002Q2", "2003Q1"]

In [157]: index = pd.PeriodIndex(values, freq="Q-DEC")

In [158]: index
Out[158]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]')
```

Преобразование частоты периода

Периоды и объекты `PeriodIndex` можно преобразовать с изменением частоты, воспользовавшись методом `asfreq`. Для примера предположим, что имеется годовой период, который мы хотим преобразовать в месячный, начинающийся или заканчивающийся на границе года. Это довольно просто:

```
In [159]: p = pd.Period("2011", freq="A-DEC")

In [160]: p
Out[160]: Period('2011', 'A-DEC')

In [161]: p.asfreq("M", how="start")
Out[161]: Period('2011-01', 'M')

In [162]: p.asfreq("M", how="end")
Out[162]: Period('2011-12', 'M')

In [163]: p.asfreq("M")
Out[163]: Period('2011-12', 'M')
```

Можно рассматривать `Period("2011", "A-DEC")` как курсор, указывающий на промежуток времени, поделенный на периоды продолжительностью один месяц. Это проиллюстрировано на рис. 11.1. Для финансового года, заканчивающегося в любом месяце, кроме декабря, месячные подпериоды вычисляются по-другому:

```
In [164]: p = pd.Period("2011", freq="A-JUN")

In [165]: p
Out[165]: Period('2011', 'A-JUN')

In [166]: p.asfreq("M", how="start")
Out[166]: Period('2010-07', 'M')
```

```
In [167]: p.asfreq("M", how="end")
Out[167]: Period('2011-06', 'M')
```

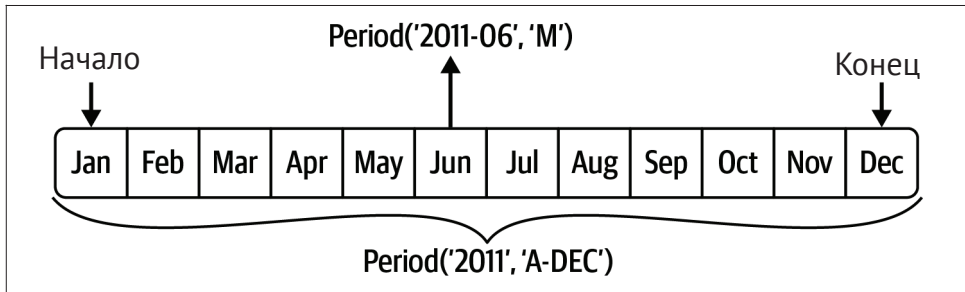


Рис. 11.1. Иллюстрация на тему преобразования частоты периода

Когда производится преобразование из большей частоты в меньшую, объемлющий период определяется в зависимости от того, куда попадает подпериод. Например, если частота равна `A-JUN`, то месяц `Aug-2011` фактически является частью периода `2012`:

```
In [168]: p = pd.Period("Aug-2011", "M")
```

```
In [169]: p.asfreq("A-JUN")
Out[169]: Period('2012', 'A-JUN')
```

Эта семантика сохраняется и в случае преобразования целых объектов `PeriodIndex` или временных рядов:

```
In [170]: periods = pd.period_range("2006", "2009", freq="A-DEC")
```

```
In [171]: ts = pd.Series(np.random.standard_normal(len(periods)), index=periods)
```

```
In [172]: ts
```

```
Out[172]:
```

```
2006    1.607578
```

```
2007     0.200381
```

```
2008    -0.834068
```

```
2009    -0.302988
```

```
Freq: A-DEC, dtype: float64
```

```
In [173]: ts.asfreq("M", how="start")
```

```
Out[173]:
```

```
2006-01    1.607578
```

```
2007-01     0.200381
```

```
2008-01    -0.834068
```

```
2009-01    -0.302988
```

```
Freq: M, dtype: float64
```

Здесь каждый годичный период заменен месячным, соответствующим первому попадающему в него месяцу. Если бы мы вместо этого захотели получить последний рабочий день каждого года, то должны были бы задать частоту `'B'` и указать, что нам нужен конец периода:

```
In [174]: ts.asfreq("B", how="end")
Out[174]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```

Квартальная частота периода

Квартальные данные стандартно применяются в бухгалтерии, финансах и других областях. Обычно квартальные итоги подводятся относительно *конца финансового года*, каковым считается последний календарный или рабочий день одного из 12 месяцев. Следовательно, период 2012Q4 интерпретируется по-разному в зависимости от того, что понимать под концом финансового года. Библиотека pandas поддерживает все 12 возможных значений квартальной частоты от Q-JAN до Q-DEC:

```
In [175]: p = pd.Period("2012Q4", freq="Q-JAN")

In [176]: p
Out[176]: Period('2012Q4', 'Q-JAN')
```

Если финансовый год заканчивается в январе, то период 2012Q4 охватывает месяцы с ноября 2011 по январь 2012 года, и это легко проверить, преобразовав квартальную частоту в суточную.

```
In [177]: p.asfreq("D", how="start")
Out[177]: Period('2011-11-01', 'D')

In [178]: p.asfreq("D", how="end")
Out[178]: Period('2012-01-31', 'D')
```

См. иллюстрацию на рис. 11.2.

Year 2012													
M	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	
Q-DEC	2012Q1			2012Q2			2012Q3			2012Q4			
Q-SEP	2012Q2			2012Q3			2012Q4			2013Q1			
Q-FEB	2012Q4			2013Q1			2013Q2			2013Q3			Q4

Рис. 11.2. Различные соглашения о квартальной частоте

Таким образом, арифметические операции с периодами выполняются очень просто; например, чтобы получить временную метку для момента «4 часа полудни предпоследнего рабочего дня квартала», можно было бы написать:

```
In [179]: p4pm = (p.asfreq("B", how="end") - 1).asfreq("T", how="start") + 16 * 60

In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')

In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

Метод `to_timestamp` по умолчанию возвращает объект `Timestamp`, соответствующий началу периода.

Для генерирования квартальных диапазонов применяется функция `pandas.period_range`. Арифметические операции такие же:

```
In [182]: periods = pd.period_range("2011Q3", "2012Q4", freq="Q-JAN")

In [183]: ts = pd.Series(np.arange(len(periods)), index=periods)

In [184]: ts
Out[184]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64

In [185]: new_periods = (periods.asfreq("B", "end") - 1).asfreq("H", "start") + 16

In [186]: ts.index = new_periods.to_timestamp()

In [187]: ts
Out[187]:
2010-10-28 16:00:00    0
2011-01-28 16:00:00    1
2011-04-28 16:00:00    2
2011-07-28 16:00:00    3
2011-10-28 16:00:00    4
2012-01-30 16:00:00    5
dtype: int64
```

Преобразование временных меток в периоды и обратно

Объекты `Series` и `DataFrame`, индексированные временными метками, можно преобразовать в периоды методом `to_period`:

```
In [188]: dates = pd.date_range("2000-01-01", periods=3, freq="M")

In [189]: ts = pd.Series(np.random.standard_normal(3), index=dates)

In [190]: ts
Out[190]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64
```

```
In [191]: pts = ts.to_period()
```

```
In [192]: pts
```

```
Out[192]:
```

```
2000-01    1.663261
```

```
2000-02   -0.996206
```

```
2000-03    1.521760
```

```
Freq: M, dtype: float64
```

Поскольку периоды всегда ссылаются на непересекающиеся временные промежутки, то временная метка может принадлежать только одному периоду данной частоты. Частота нового объекта `PeriodIndex` по умолчанию выводится из временных меток, но можно задать любую поддерживаемую частоту (большинство частот в табл. 11.4 поддерживаются). Наличие повторяющихся периодов в результате также не приводит ни к каким проблемам:

```
In [193]: dates = pd.date_range("2000-01-29", periods=6)
```

```
In [194]: ts2 = pd.Series(np.random.standard_normal(6), index=dates)
```

```
In [195]: ts2
```

```
Out[195]:
```

```
2000-01-29    0.244175
```

```
2000-01-30    0.423331
```

```
2000-01-31   -0.654040
```

```
2000-02-01    2.089154
```

```
2000-02-02   -0.060220
```

```
2000-02-03   -0.167933
```

```
Freq: D, dtype: float64
```

```
In [196]: ts2.to_period("M")
```

```
Out[196]:
```

```
2000-01    0.244175
```

```
2000-01    0.423331
```

```
2000-01   -0.654040
```

```
2000-02    2.089154
```

```
2000-02   -0.060220
```

```
2000-02   -0.167933
```

```
Freq: M, dtype: float64
```

Для обратного преобразования во временные метки служит метод `to_timestamp`, который возвращает объект типа `DatetimeIndex`:

```
In [197]: pts = ts2.to_period()
```

```
In [198]: pts
```

```
Out[198]:
```

```
2000-01-29    0.244175
```

```
2000-01-30    0.423331
```

```
2000-01-31   -0.654040
```

```
2000-02-01    2.089154
```

```
2000-02-02   -0.060220
```

```
2000-02-03   -0.167933
```

```
Freq: D, dtype: float64
```

```
In [199]: pts.to_timestamp(how="end")
```

```
Out[199]:
```

```

2000-01-29 23:59:59.999999999    0.244175
2000-01-30 23:59:59.999999999    0.423331
2000-01-31 23:59:59.999999999   -0.654040
2000-02-01 23:59:59.999999999    2.089154
2000-02-02 23:59:59.999999999   -0.060220
2000-02-03 23:59:59.999999999   -0.167933
Freq: D, dtype: float64

```

Создание PeriodIndex из массивов

В наборах данных с фиксированной частотой информация о промежутках времени иногда хранится в нескольких столбцах. Например, в следующем макроэкономическом наборе данных год и квартал находятся в разных столбцах:

```
In [200]: data = pd.read_csv("examples/macrodta.csv")
```

```
In [201]: data.head(5)
```

```
Out[201]:
```

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi \
0	1959	1	2710.349	1707.4	286.898	470.045	1886.9	28.98
1	1959	2	2778.801	1733.7	310.859	481.301	1919.7	29.15
2	1959	3	2775.488	1751.8	289.226	491.260	1916.4	29.35
3	1959	4	2785.204	1753.7	299.356	484.052	1931.3	29.37
4	1960	1	2847.699	1770.5	331.722	462.199	1955.5	29.54

	m1	tbilrate	unemp	pop	infl	realint
0	139.7	2.82	5.8	177.146	0.00	0.00
1	141.7	3.08	5.1	177.830	2.34	0.74
2	140.5	3.82	5.3	178.657	2.74	1.09
3	140.0	4.33	5.6	179.386	0.27	4.06
4	139.6	3.50	5.2	180.007	2.31	1.19

```
In [202]: data["year"]
```

```
Out[202]:
```

```

0      1959
1      1959
2      1959
3      1959
4      1960
...
198    2008
199    2008
200    2009
201    2009
202    2009

```

```
Name: year, Length: 203, dtype: int64
```

```
In [203]: data["quarter"]
```

```
Out[203]:
```

```

0      1
1      2
2      3
3      4
4      1
...
198    3
199    4

```



```

200    1
201    2
202    3
Name: quarter, Length: 203, dtype: int64

```

Передав эти массивы конструктору `PeriodIndex` вместе с частотой, мы сможем объединить их для построения индекса `DataFrame`:

```

In [204]: index = pd.PeriodIndex(year=data["year"], quarter=data["quarter"],
.....:                          freq="Q-DEC")

```

```

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
            '1960Q3', '1960Q4', '1961Q1', '1961Q2',
            ...
            '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
            '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203)

```

```

In [206]: data.index = index

```

```

In [207]: data["infl"]
Out[207]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
1960Q1    2.31
...
2008Q3   -3.16
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64

```

11.6. ПЕРЕДИСКРЕТИЗАЦИЯ И ПРЕОБРАЗОВАНИЕ ЧАСТОТЫ

Под *передискретизацией* понимается процесс изменения частоты временно-го ряда. Агрегирование с переходом от высокой частоты к низкой называется *понижающей передискретизацией*, а переход от низкой частоты к более высокой – *повышающей передискретизацией*. Не всякая передискретизация попадает в одну из этих категорий; например, преобразование частоты *W-WED* (еженедельно по средам) в *W-FRI* не повышает и не понижает частоту.

Все объекты `pandas` имеют метод `resample`, отвечающий за любые преобразования частоты. API метода `resample` примерно такой же, как у `groupby`; мы сначала вызываем `resample` для группировки данных, а затем обращаемся к функции агрегирования:

```

In [208]: dates = pd.date_range("2000-01-01", periods=100)

In [209]: ts = pd.Series(np.random.standard_normal(len(dates)), index=dates)

```

```

In [210]: ts
Out[210]:
2000-01-01    0.631634
2000-01-02   -1.594313
2000-01-03   -1.519937
2000-01-04    1.108752
2000-01-05    1.255853
...
2000-04-05   -0.423776
2000-04-06    0.789740
2000-04-07    0.937568
2000-04-08   -2.253294
2000-04-09   -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample("M").mean()
Out[211]:
2000-01-31   -0.165893
2000-02-29    0.078606
2000-03-31    0.223811
2000-04-30   -0.063643
Freq: M, dtype: float64

In [212]: ts.resample("M", kind="period").mean()
Out[212]:
2000-01   -0.165893
2000-02    0.078606
2000-03    0.223811
2000-04   -0.063643
Freq: M, dtype: float64

```

Метод `resample` обладает большой гибкостью и применим к большим временным рядам. Примеры в следующих разделах иллюстрируют его семантику и использование, а в табл. 11.5 перечислены некоторые его аргументы.

Таблица 11.5. Аргументы метода `resample`

Аргумент	Описание
<code>rule</code>	Строка или объект <code>DateOffset</code> , задающий новую частоту, например <code>'M'</code> , <code>'5min'</code> или <code>Second(15)</code>
<code>axis</code>	Ось передискретизации, по умолчанию 0
<code>fill_method</code>	Способ интерполяции при повышающей передискретизации, например <code>"ffill"</code> или <code>"bfill"</code> . По умолчанию интерполяция не производится
<code>closed</code>	При понижающей передискретизации определяет, какой конец интервала должен включаться: <code>"right"</code> (правый) или <code>"left"</code> (левый)
<code>label</code>	При понижающей передискретизации определяет, следует ли помечать агрегированный результат меткой правого или левого конца интервала. Например, пятиминутный интервал от 9:30 до 9:35 можно пометить меткой 9:30 или 9:35. По умолчанию <code>"right"</code> (т. е. 9:35 в этом примере)
<code>limit</code>	При прямом или обратном восполнении максимальное количество восполняемых периодов

Аргумент	Описание
<code>kind</code>	Агрегировать в периоды (" <code>period</code> ") или временные метки (" <code>timestamp</code> "); по умолчанию определяется видом индекса, связанного с данным временным рядом
<code>convention</code>	При передискретизации периодов соглашение (" <code>start</code> " или " <code>end</code> ") о преобразовании периода низкой частоты в период высокой частоты. По умолчанию " <code>start</code> "
<code>origin</code>	«Базовая» временная метка, от которой отсчитываются границы интервала передискретизации; может также принимать значения " <code>epoch</code> ", " <code>start</code> ", " <code>start_day</code> ", " <code>end</code> ", " <code>end_day</code> "; полное описание см. в строке документации для <code>resample</code>
<code>offset</code>	Смещение (объект <code>timedelta</code>), прибавляемое к начальному моменту; по умолчанию <code>None</code>

Понижающая передискретизация

Понижающей передискретизацией называется агрегирование данных с целью понижения и регуляризации частоты. Агрегируемые данные необязательно исправлять часто; желаемая частота определяет *границы интервалов*, разбивающих агрегируемые данные на порции. Например, для преобразования к месячному периоду, "`М`" или "`ВМ`", данные нужно разбить на интервалы продолжительностью один месяц. Говорят, что каждый интервал *полуоткрыт*; любая точка может принадлежать только одному интервалу, а их объединение должно покрывать всю протяженность временного ряда. Перед тем как выполнять понижающую передискретизацию данных методом `resample`, нужно решить для себя следующие вопросы:

- какой конец интервала будет включаться;
- помечать ли агрегированный интервал меткой его начала или конца.

Для иллюстрации рассмотрим данные с частотой одна минута:

```
In [213]: dates = pd.date_range("2000-01-01", periods=12, freq="T")
```

```
In [214]: ts = pd.Series(np.arange(len(dates)), index=dates)
```

```
In [215]: ts
```

```
Out[215]:
```

```
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
2000-01-01 00:09:00    9
2000-01-01 00:10:00   10
2000-01-01 00:11:00   11
Freq: T, dtype: int64
```

Пусть требуется агрегировать данные в пятиминутные группы, или *столбики*, вычислив сумму по каждой группе:

```
In [216]: ts.resample("5min").sum()
Out[216]:
2000-01-01 00:00:00    10
2000-01-01 00:05:00    35
2000-01-01 00:10:00    21
Freq: 5T, dtype: int64
```

Переданная частота определяет границы интервалов с пятиминутным приращением. Для этой частоты левый конец интервала по умолчанию включается, т. е. значение 00:00 включается в интервал от 00:00 до 00:05, а значение 00:05 исключается⁵. Если задать параметр `closed='right'`, то будет включаться правый конец интервала:

```
In [217]: ts.resample("5min", closed="right").sum()
Out[217]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int64
```

Результирующий временной ряд помечен временными метками, соответствующими левым концам интервалов. Параметр `label='right'` позволяет использовать для этой цели метки правых концов:

```
In [218]: ts.resample("5min", closed="right", label="right").sum()
Out[218]:
2000-01-01 00:00:00     0
2000-01-01 00:05:00    15
2000-01-01 00:10:00    40
2000-01-01 00:15:00    11
Freq: 5T, dtype: int64
```

На рис. 11.3 показано, как данные с минутной частотой агрегируются в пятиминутные группы.

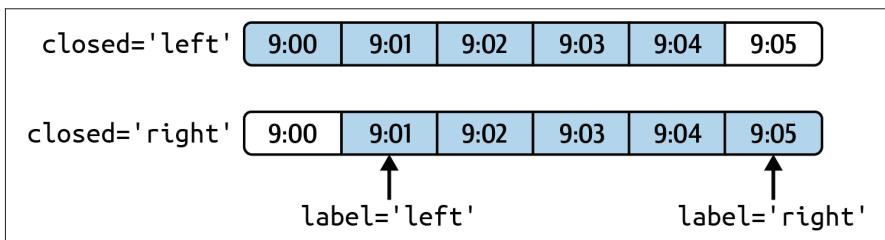


Рис. 11.3. Соглашения о включении конца и о метках интервалов на примере передискретизации с частотой 5 минут

⁵ Выбор значений `closed` и `label` по умолчанию некоторым пользователям может показаться странным. Значение `closed="left"` подразумевается по умолчанию для всех частот, кроме "M", "A", "Q", "BM", "BQ" и "W", для которых по умолчанию подразумевается `closed="right"`. Значения по умолчанию были выбраны, так чтобы результат казался интуитивно очевидным, но следует помнить, что они не всегда одинаковы.

Наконец, иногда желательно сдвинуть индекс результата на какую-то величину, скажем вычесть одну секунду из правого конца, чтобы было понятнее, к какому интервалу относится временная метка. Для этого следует прибавить смещение к результирующему индексу:

```
In [219]: from pandas.tseries.frequencies import to_offset

In [220]: result = ts.resample("5min", closed="right", label="right").sum()

In [221]: result.index = result.index + to_offset("-1s")

In [222]: result
Out[222]:
1999-12-31 23:59:59    0
2000-01-01 00:04:59   15
2000-01-01 00:09:59   40
2000-01-01 00:14:59   11
Freq: 5T, dtype: int64
```

Передискретизация OHLC

В финансовых приложениях очень часто временной ряд агрегируют, вычисляя четыре значения для каждого интервала: первое (открытие – open), последнее (закрытие – close), максимальное (high) и минимальное (low). Задав параметр `how='ohlc'`, мы получим объект `DataFrame`, в столбцах которого находятся эти четыре агрегата, которые эффективно вычисляются одним вызовом функции:

```
In [223]: ts = pd.Series(np.random.permutation(np.arange(len(dates))), index=dates)

In [224]: ts.resample("5min").ohlc()
Out[224]:
```

	open	high	low	close
2000-01-01 00:00:00	8	8	1	5
2000-01-01 00:05:00	6	11	2	2
2000-01-01 00:10:00	0	7	0	7

Повышающая передискретизация и интерполяция

Повышающая передискретизация – это преобразование от низкой частоты к более высокой, агрегирование при этом не требуется. Рассмотрим объект `DataFrame`, содержащий недельные данные:

```
In [225]: frame = pd.DataFrame(np.random.standard_normal((2, 4)),
.....:                        index=pd.date_range("2000-01-01", periods=2,
.....:                        freq="W-WED"),
.....:                        columns=["Colorado", "Texas", "New York", "Ohio"])

In [226]: frame
Out[226]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Если мы используем с этими данными функцию агрегирования, то на каждую группу получается только одно значение, а отсутствующие значения приводят к лакунам. Чтобы перейти к более высокой частоте без агрегирования, применяется метод `asfreq`:

```
In [227]: df_daily = frame.resample("D").asfreq()
```

```
In [228]: df_daily
```

```
Out[228]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-06	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Допустим, мы хотим восполнить значения для дней, отличных от среды. Для этого применимы те же способы восполнения или интерполяции, что в методах `fillna` и `reindex`:

```
In [229]: frame.resample("D").ffill()
```

```
Out[229]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-06	-0.896431	0.927238	0.482284	-0.867130
2000-01-07	-0.896431	0.927238	0.482284	-0.867130
2000-01-08	-0.896431	0.927238	0.482284	-0.867130
2000-01-09	-0.896431	0.927238	0.482284	-0.867130
2000-01-10	-0.896431	0.927238	0.482284	-0.867130
2000-01-11	-0.896431	0.927238	0.482284	-0.867130
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Можно восполнить отсутствующие значения не во всех последующих периодах, а только в заданном числе:

```
In [230]: frame.resample("D").ffill(limit=2)
```

```
Out[230]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-06	-0.896431	0.927238	0.482284	-0.867130
2000-01-07	-0.896431	0.927238	0.482284	-0.867130
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Важно отметить, что новый индекс дат может не совпадать со старым:

```
In [231]: frame.resample("W-THU").ffill()
```

```
Out[231]:
```

	Colorado	Texas	New York	Ohio
2000-01-06	-0.896431	0.927238	0.482284	-0.867130
2000-01-13	0.493841	-0.155434	1.397286	1.507055

Передискретизация периодов

Передискретизация данных, индексированных периодами, похожа на индексирование временными метками:

```
In [232]: frame = pd.DataFrame(np.random.standard_normal((24, 4)),
.....:                        index=pd.period_range("1-2000", "12-2001",
.....:                        freq="M"),
.....:                        columns=["Colorado", "Texas", "New York", "Ohio"])
```

```
In [233]: frame.head()
```

```
Out[233]:
```

	Colorado	Texas	New York	Ohio
2000-01	-1.179442	0.443171	1.395676	-0.529658
2000-02	0.787358	0.248845	0.743239	1.267746
2000-03	1.302395	-0.272154	-0.051532	-0.467740
2000-04	-1.040816	0.426419	0.312945	-1.115689
2000-05	1.234297	-1.893094	-1.661605	-0.005477

```
In [234]: annual_frame = frame.resample("A-DEC").mean()
```

```
In [235]: annual_frame
```

```
Out[235]:
```

	Colorado	Texas	New York	Ohio
2000	0.487329	0.104466	0.020495	-0.273945
2001	0.203125	0.162429	0.056146	-0.103794

Повышающая передискретизация чуть сложнее, потому что необходимо принять решение о том, в какой конец промежутка времени для новой частоты помещать значения до передискретизации, как в случае метода `asfreq`. Аргумент `convention` по умолчанию равен `"start"`, но можно задать и значение `"end"`:

Q-DEC: поквартально, год заканчивается в декабре

```
In [236]: annual_frame.resample("Q-DEC").ffill()
```

```
Out[236]:
```

	Colorado	Texas	New York	Ohio
2000Q1	0.487329	0.104466	0.020495	-0.273945
2000Q2	0.487329	0.104466	0.020495	-0.273945
2000Q3	0.487329	0.104466	0.020495	-0.273945
2000Q4	0.487329	0.104466	0.020495	-0.273945
2001Q1	0.203125	0.162429	0.056146	-0.103794
2001Q2	0.203125	0.162429	0.056146	-0.103794
2001Q3	0.203125	0.162429	0.056146	-0.103794
2001Q4	0.203125	0.162429	0.056146	-0.103794

```
In [237]: annual_frame.resample("Q-DEC", convention="end").asfreq()
```

```
Out[237]:
```

	Colorado	Texas	New York	Ohio
2000Q4	0.487329	0.104466	0.020495	-0.273945
2001Q1	NaN	NaN	NaN	NaN
2001Q2	NaN	NaN	NaN	NaN
2001Q3	NaN	NaN	NaN	NaN
2001Q4	0.203125	0.162429	0.056146	-0.103794

Поскольку периоды ссылаются на промежутки времени, правила повышающей и понижающей передискретизаций более строгие:

- в случае понижающей передискретизации конечная частота должна быть *подпериодом* начальной;
- в случае повышающей передискретизации конечная частота должна быть *надпериодом* начальной.

Если эти правила не выполнены, то будет возбуждено исключение. Это относится главным образом к квартальной, годовой и недельной частотам; например, промежутки времени, определенные частотой **Q-MAR**, выровнены только с периодами **A-MAR**, **A-JUN**, **A-SEP** и **A-DEC**:

```
In [238]: annual_frame.resample("Q-MAR").ffill()
Out[238]:
```

	Colorado	Texas	New York	Ohio
2000Q4	0.487329	0.104466	0.020495	-0.273945
2001Q1	0.487329	0.104466	0.020495	-0.273945
2001Q2	0.487329	0.104466	0.020495	-0.273945
2001Q3	0.487329	0.104466	0.020495	-0.273945
2001Q4	0.203125	0.162429	0.056146	-0.103794
2002Q1	0.203125	0.162429	0.056146	-0.103794
2002Q2	0.203125	0.162429	0.056146	-0.103794
2002Q3	0.203125	0.162429	0.056146	-0.103794

Групповая передискретизация по времени

Для временных рядов метод **resample** семантически является групповой операцией, основанной на разбиении времени на интервалы. Рассмотрим пример таблицы:

```
In [239]: N = 15
```

```
In [240]: times = pd.date_range("2017-05-20 00:00", freq="1min", periods=N)
```

```
In [241]: df = pd.DataFrame({"time": times,
.....:                      "value": np.arange(N)})
```

```
In [242]: df
```

```
Out[242]:
```

	time	value
0	2017-05-20 00:00:00	0
1	2017-05-20 00:01:00	1
2	2017-05-20 00:02:00	2
3	2017-05-20 00:03:00	3
4	2017-05-20 00:04:00	4
5	2017-05-20 00:05:00	5
6	2017-05-20 00:06:00	6
7	2017-05-20 00:07:00	7
8	2017-05-20 00:08:00	8
9	2017-05-20 00:09:00	9
10	2017-05-20 00:10:00	10
11	2017-05-20 00:11:00	11
12	2017-05-20 00:12:00	12
13	2017-05-20 00:13:00	13
14	2017-05-20 00:14:00	14

Здесь мы можем индексировать по столбцу `"time"`, а затем выполнить пере-дискретизацию:

```
In [243]: df.set_index("time").resample("5min").count()
Out[243]:
```

time	value
2017-05-20 00:00:00	5
2017-05-20 00:05:00	5
2017-05-20 00:10:00	5

Предположим, что объект `DataFrame` включает несколько временных рядов, которые различаются по дополнительному столбцу, содержащему групповой ключ:

```
In [244]: df2 = pd.DataFrame({"time": times.repeat(3),
.....:                      "key": np.tile(["a", "b", "c"], N),
.....:                      "value": np.arange(N * 3.)})

In [245]: df2.head(7)
Out[245]:
```

	time	key	value
0	2017-05-20 00:00:00	a	0.0
1	2017-05-20 00:00:00	b	1.0
2	2017-05-20 00:00:00	c	2.0
3	2017-05-20 00:01:00	a	3.0
4	2017-05-20 00:01:00	b	4.0
5	2017-05-20 00:01:00	c	5.0
6	2017-05-20 00:02:00	a	6.0

Чтобы выполнить одну и ту же передискретизацию для каждого значения `"key"`, введем объект `pandas.Grouper`:

```
In [246]: time_key = pd.Grouper(freq="5min")
```

Затем можно задать индекс по времени, сгруппировать по `"key"` и `time_key` и агрегировать:

```
In [247]: resampled = (df2.set_index("time")
.....:                  .groupby(["key", time_key])
.....:                  .sum())

In [248]: resampled
Out[248]:
```

	key	time	value
a	a	2017-05-20 00:00:00	30.0
		2017-05-20 00:05:00	105.0
		2017-05-20 00:10:00	180.0
b	b	2017-05-20 00:00:00	35.0
		2017-05-20 00:05:00	110.0
		2017-05-20 00:10:00	185.0
c	c	2017-05-20 00:00:00	40.0
		2017-05-20 00:05:00	115.0
		2017-05-20 00:10:00	190.0

```
In [249]: resampled.reset_index()
Out[249]:
```

	key	time	value
0	a	2017-05-20 00:00:00	30.0
1	a	2017-05-20 00:05:00	105.0
2	a	2017-05-20 00:10:00	180.0
3	b	2017-05-20 00:00:00	35.0
4	b	2017-05-20 00:05:00	110.0
5	b	2017-05-20 00:10:00	185.0
6	c	2017-05-20 00:00:00	40.0
7	c	2017-05-20 00:05:00	115.0
8	c	2017-05-20 00:10:00	190.0

У `pandas.Grouper` есть ограничение – индекс Series или DataFrame должен быть построен по времени.

11.7. СКОЛЬЗЯЩИЕ ОКОННЫЕ ФУНКЦИИ

Важный класс преобразований массива, применяемый для операций с временными рядами, – статистические и иные функции, вычисляемые в скользящем окне или с экспоненциально убывающими весами. Я называю их *скользящими оконными функциями*, хотя сюда относятся также функции, не связанные с окном постоянной ширины, например экспоненциально взвешенное скользящее среднее. Как и во всех статистических функциях, отсутствующие значения автоматически отбрасываются.

Для начала загрузим временной ряд и передискретизируем его на частоту «рабочий день»:

```
In [250]: close_px_all = pd.read_csv("examples/stock_px.csv",
.....: parse_dates=True, index_col=0)
In [251]: close_px = close_px_all[["AAPL", "MSFT", "XOM"]]
In [252]: close_px = close_px.resample("B").ffill()
```

Теперь я введу в рассмотрение оператор `rolling`, который ведет себя как `resample` и `groupby`. Его можно применить к объекту Series или DataFrame, передав аргумент `window` (равный количеству периодов; созданный график показан на рис. 11.4):

```
In [253]: close_px["AAPL"].plot()
Out[253]: <AxesSubplot:>

In [254]: close_px["AAPL"].rolling(250).mean().plot()
```

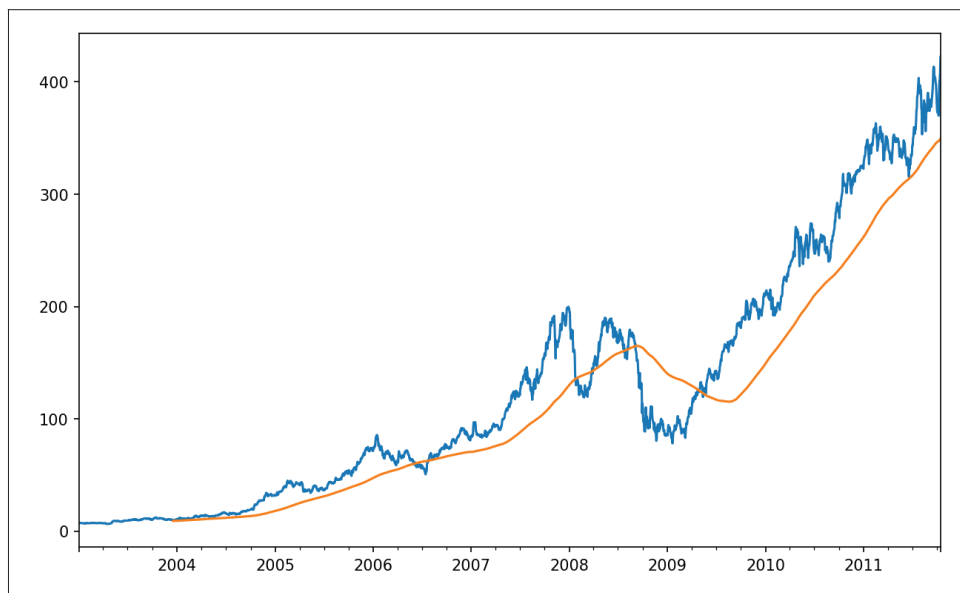


Рис. 11.4. Скользящее среднее котировок акций Apple за 250 дней

Выражение `rolling(250)` ведет себя как `groupby`, но вместо группировки создает объект, который допускает группировку по скользящему окну шириной 250 дней. Таким образом, здесь мы имеем средние котировки акций Apple в скользящем окне шириной 250 дней.

По умолчанию функции создания скользящих окон требуют, чтобы все значения в окне были отличны от NA. Это поведение можно изменить, чтобы учесть возможность отсутствия данных и, в частности, тот факт, что в начале временного ряда количество периодов данных меньше `window` (см. рис. 11.5):

```
In [255]: plt.figure()
Out[255]: <Figure size 1000x600 with 0 Axes>
In [256]: std250 = close_px["AAPL"].pct_change().rolling(250, min_periods=10).std()
()
```

```
In [257]: std250[5:12]
Out[257]:
2003-01-09      NaN
2003-01-10      NaN
2003-01-13      NaN
2003-01-14      NaN
2003-01-15      NaN
2003-01-16    0.009628
2003-01-17    0.013818
Freq: B, Name: AAPL, dtype: float64
```

```
In [258]: std250.plot()
```

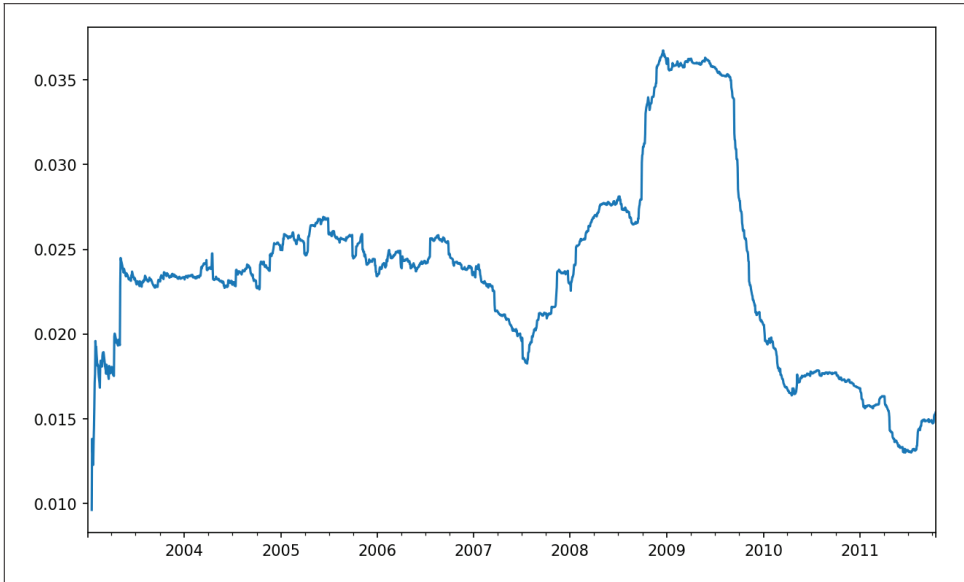


Рис. 11.5. Стандартное отклонение суточного оборота Apple

Чтобы вычислить *среднее с расширяющимся окном*, используйте оператор `expanding` вместо `rolling`. В этом случае начальное окно расположено в начале временного ряда и увеличивается в размере, пока не охватит весь ряд. Среднее с расширяющимся окном для временного ряда `std250` вычисляется следующим образом:

```
In [259]: expanding_mean = std250.expanding().mean()
```

При вызове функции скользящего окна от имени объекта `DataFrame` преобразование применяется к каждому столбцу (см. рис. 11.6):

```
In [261]: plt.style.use('grayscale')
```

```
In [262]: close_px.rolling(60).mean().plot(logy=True)
```

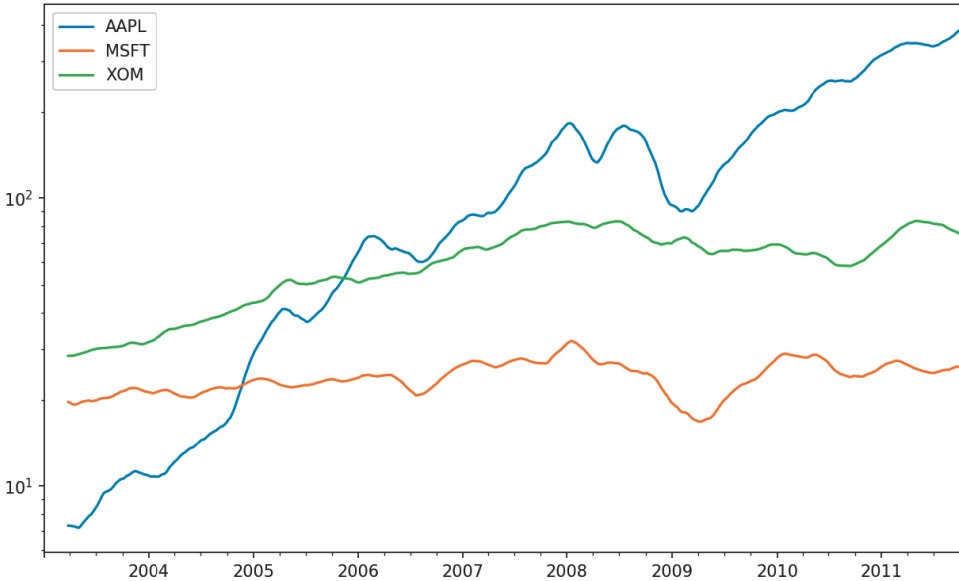


Рис. 11.6. Скользящее среднее котировок акций за 60 дней (по оси у отложен логарифм)

Функция `rolling` принимает также строку, содержащую фиксированное временное смещение, а не количество периодов. Такой вариант может быть полезен для нерегулярных временных рядов. Точно такие же строки передаются `resample`. Например, вот как можно было бы вычислить скользящее среднее за 20 дней:

```
In [263]: close_px.rolling("20D").mean()
Out[263]:
```

	AAPL	MSFT	XOM
2003-01-02	7.400000	21.110000	29.220000
2003-01-03	7.425000	21.125000	29.230000
2003-01-06	7.433333	21.256667	29.473333
2003-01-07	7.432500	21.425000	29.342500
2003-01-08	7.402000	21.402000	29.240000
...	...	...	...
2011-10-10	389.351429	25.602143	72.527857
2011-10-11	388.505000	25.674286	72.835000
2011-10-12	388.531429	25.810000	73.400714
2011-10-13	388.826429	25.961429	73.905000
2011-10-14	391.038000	26.048667	74.185333

[2292 rows x 3 columns]

Экспоненциально взвешенные функции

Вместо использования окна постоянного размера, когда веса всех наблюдений одинаковы, можно задать постоянный коэффициент затухания, чтобы повысить вес последних наблюдений. Есть два способа задать коэффициент затухания, самый популярный – использовать *промежуток* (`span`), потому что результаты в этом случае получаются сравнимыми с применением

простой скользящей оконной функции, для которой размер окна равен промежутку.

Поскольку экспоненциально взвешенная статистика придает больший вес недавним наблюдениям, она быстрее «адаптируется» к изменениям по сравнению с вариантом с равными весами.

В pandas имеется оператор `ewm` (exponentially weighted moving – экспоненциально взвешенный сдвиг), который работает совместно с `rolling` и `expanding`. В примере ниже скользящее среднее котировок акций Apple за 60 дней сравнивается с экспоненциально взвешенным (EW) скользящим средним для `span=60` (рис. 11.7):

```
In [265]: aapl_px = close_px["AAPL"]["2006":"2007"]

In [266]: ma30 = aapl_px.rolling(30, min_periods=20).mean()

In [267]: ewma30 = aapl_px.ewm(span=30).mean()

In [268]: aapl_px.plot(style="k-", label="Price")
Out[268]: <AxesSubplot:>

In [269]: ma30.plot(style="k--", label="Simple Moving Avg")
Out[269]: <AxesSubplot:>

In [270]: ewma30.plot(style="k-", label="EW MA")
Out[270]: <AxesSubplot:>

In [271]: plt.legend()
```

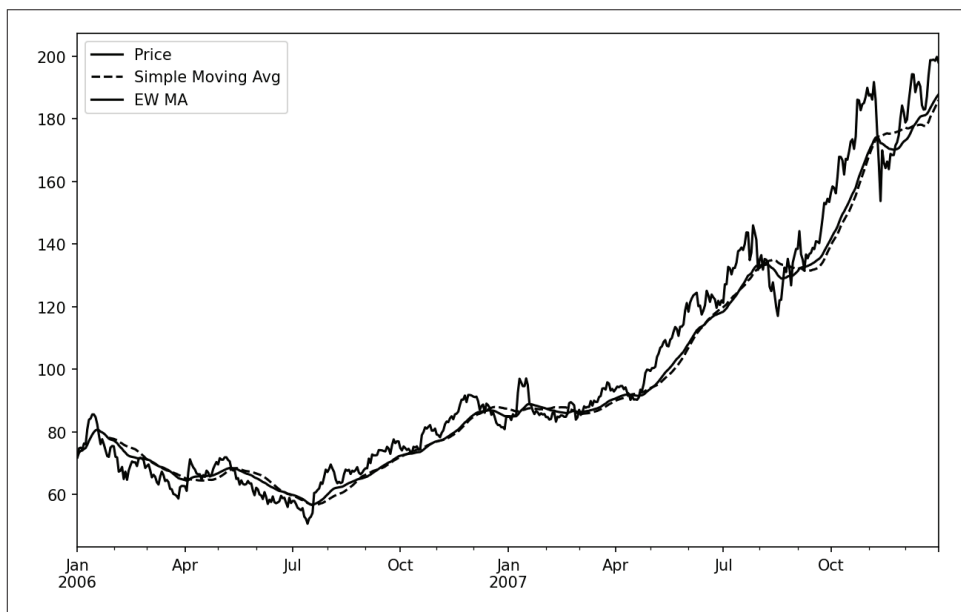


Рис. 11.7. Простое и экспоненциально взвешенное скользящее среднее

Бинарные скользящие оконные функции

Для некоторых статистических операций, в частности корреляции и ковариации, необходимы два временных ряда. Например, финансовых аналитиков часто интересует корреляция цены акции с основным биржевым индексом типа S&P 500. Чтобы найти эту величину, мы сначала вычислим относительные изменения в процентах для всего нашего временного ряда:

```
In [273]: spx_px = close_px_all("SPX")
```

```
In [274]: spx_rets = spx_px.pct_change()
```

```
In [275]: returns = close_px.pct_change()
```

Если теперь вызвать `rolling`, а за ней функцию агрегирования `corr`, то мы сможем вычислить скользящую корреляцию с `spx_rets` (получающийся график показан на рис. 11.8):

```
In [276]: corr = returns["AAPL"].rolling(125, min_periods=100).corr(spx_rets)
```

```
In [277]: corr.plot()
```



Рис. 11.8. Корреляция доходности AAPL с индексом S&P 500, рассчитанная за шесть месяцев

Пусть требуется вычислить корреляцию индекса S&P 500 сразу с несколькими акциями. Можно было бы написать цикл, в котором она вычисляется для каждой акции, как было сделано для Apple выше, но если каждая акция представлена столбцом в одном объекте DataFrame, то можно вычислить все корреляции за один прием, вызвав от имени DataFrame метод `rolling` и передав ему временной ряд `spx_rets`.

Результат показан на рис. 11.9.

```
In [279]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)
```

```
In [280]: corr.plot()
```

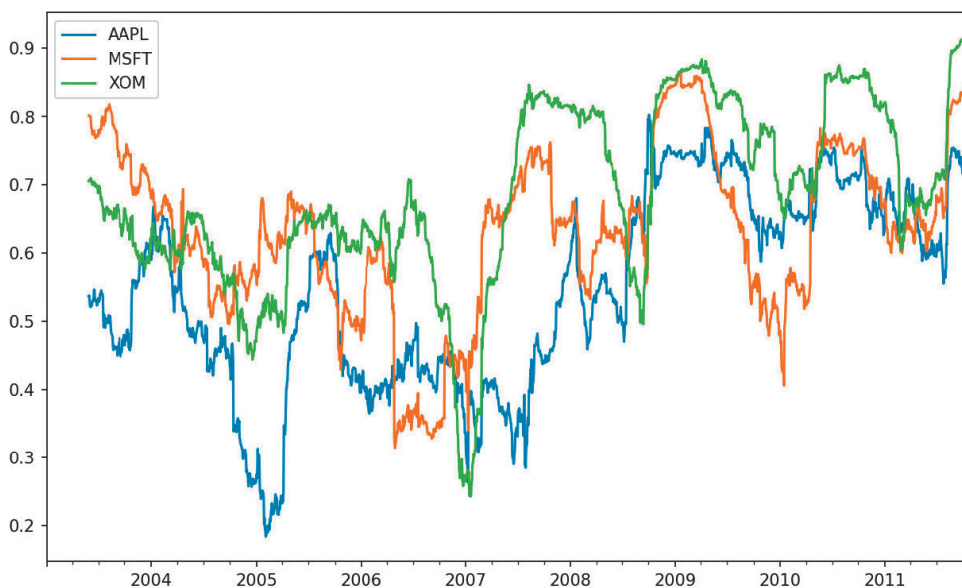


Рис. 11.9. Корреляция доходности нескольких акций с индексом S&P 500, рассчитанная за шесть месяцев

Скользящие оконные функции, определенные пользователем

Метод `apply` скользящего окна `rolling` и других подобных объектов позволяет применить произвольную функцию, принимающую массив, к скользящему окну. Единственное требование заключается в том, что функция должна порождать только одно скалярное значение (производить редукцию) для каждого фрагмента массива. Например, при вычислении выборочных квантилей с помощью `rolling(...).quantile(q)` нам может быть интересен процентильный ранг некоторого значения относительно выборки. Это можно сделать с помощью функции `scipy.stats.percentileofscore` (график показан на рис. 11.10):

```
In [282]: from scipy.stats import percentileofscore
```

```
In [283]: def score_at_2percent(x):
.....:     return percentileofscore(x, 0.02)
```

```
In [284]: result = returns["AAPL"].rolling(250).apply(score_at_2percent)
```

```
In [285]: result.plot()
```

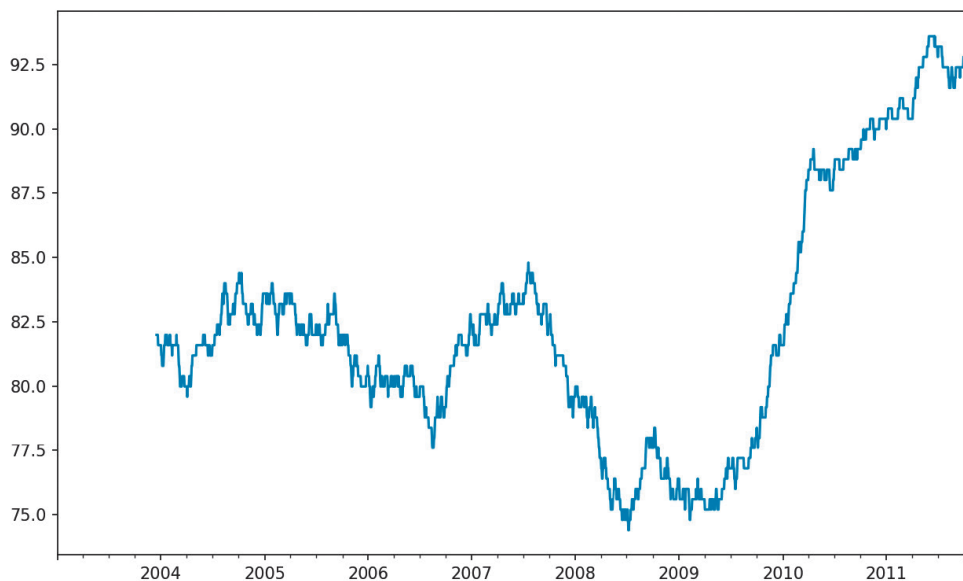



Рис. 11.10. Двухпроцентный процентильный ранг доходности AAPL, рассчитанный по окну протяженностью 1 год

Установить SciPy можно с помощью `conda` или `pip`:

```
conda install scipy
```

11.8. ЗАКЛЮЧЕНИЕ

Для анализа временных рядов нужны специальные средства преобразования данных, отличающиеся от рассмотренных в предыдущих главах.

В следующей главе мы покажем, как приступить к работе с библиотеками моделирования типа `statsmodels` и `scikit-learn`.

Введение в библиотеки моделирования на Python

В этой книге я стремился в первую очередь заложить фундамент для анализа данных на Python. Поскольку специалисты по анализу данных и науке о данных часто жалуются на непомерно высокие временные затраты на переформатирование и подготовку данных, структура книги отражает важность овладения этими навыками.

Какую библиотеку использовать для разработки моделей, зависит от приложения. Многие статистические задачи можно решить, применяя более простые методы, например регрессию методом обыкновенных наименьших квадратов, тогда как для других задач требуются более развитые методы машинного обучения. По счастью, Python стал одним из избранных языков для реализации аналитических методов, поэтому, прочитав эту книгу до конца, вы сможете исследовать многочисленные инструменты, имеющиеся в вашем распоряжении.

В этой главе я сделаю обзор некоторых возможностей `pandas`, которые могут пригодиться, когда вы будете переходить от переформатирования данных к обучению и оцениванию моделей. Затем будет приведено краткое введение в две популярные библиотеки моделирования, `statsmodels` (<http://www.statsmodels.org/stable/index.html>) и `scikit-learn` (<https://scikit-learn.org/>). Поскольку оба проекта настолько велики, что заслуживают отдельной книги, я даже не пытаюсь рассмотреть их во всей полноте, а отсылаю к документации и другим книгам по науке о данных, статистике и машинному обучению.

12.1. ИНТЕРФЕЙС МЕЖДУ PANDAS И КОДОМ МОДЕЛИ

Широко распространенный подход к разработке моделей заключается в том, чтобы с помощью `pandas` произвести загрузку и очистку данных, а затем воспользоваться библиотекой моделирования для построения самой модели. Важная часть процесса разработки модели в машинном обучении называется *конструированием признаков* (feature engineering). Речь идет об описании преобразований данных или аналитических операций, извлекающих из исходного набора данных информацию, которая может оказаться полезной в контексте моделирования. Рассмотренные в этой книге средства группировки и агрегирования часто применяются в контексте конструирования признаков.

Хотя детали «правильного» конструирования признаков выходят за рамки этой книги, я все же продемонстрирую некоторые приемы, позволяющие относительно безболезненно переключаться между манипулированием данными средствами pandas и моделированием.

Переходником между pandas и другими аналитическими библиотеками обычно являются массивы NumPy. Для преобразования объекта DataFrame в массив NumPy используется метод `to_numpy`:

```
In [12]: data = pd.DataFrame({
.....:     'x0': [1, 2, 3, 4, 5],
.....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
.....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})
```

```
In [13]: data
Out[13]:
```

	x0	x1	y
0	1	0.01	-1.5
1	2	-0.01	0.0
2	3	0.25	3.6
3	4	-4.10	1.3
4	5	0.00	-2.0

```
In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')
```

```
In [15]: data.to_numpy()
Out[15]:
array([[ 1. ,  0.01, -1.5 ],
       [ 2. , -0.01,  0. ],
       [ 3. ,  0.25,  3.6 ],
       [ 4. , -4.1 ,  1.3 ],
       [ 5. ,  0. , -2. ]])
```

Для обратного преобразования в DataFrame можно передать двумерный массив ndarray и факультативно имена столбцов:

```
In [16]: df2 = pd.DataFrame(data.to_numpy(), columns=['one', 'two', 'three'])
```

```
In [17]: df2
Out[17]:
```

	one	two	three
0	1.0	0.01	-1.5
1	2.0	-0.01	0.0
2	3.0	0.25	3.6
3	4.0	-4.10	1.3
4	5.0	0.00	-2.0

Метод `to_numpy` предназначен для использования в случае, когда данные однородны, например все имеют числовой тип. При наличии неоднородных данных получится массив ndarray объектов Python:

```
In [18]: df3 = data.copy()
```

```
In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']
```

```
In [20]: df3
```

```

Out[20]:
   x0  x1  y strings
0  1  0.01 -1.5      a
1  2 -0.01  0.0      b
2  3  0.25  3.6      c
3  4 -4.10  1.3      d
4  5  0.00 -2.0      e

In [21]: df3.to_numpy()
Out[21]:
array([[1,  0.01, -1.5, 'a'],
       [2, -0.01,  0.0, 'b'],
       [3,  0.25,  3.6, 'c'],
       [4, -4.1 ,  1.3, 'd'],
       [5,  0.0 , -2.0, 'e']], dtype=object)

```

Для некоторых моделей нужно использовать лишь подмножество столбцов. Я рекомендую `loc`-индексирование в сочетании с `to_numpy`:

```

In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].to_numpy()
Out[23]:
array([[ 1. ,  0.01],
       [ 2. , -0.01],
       [ 3. ,  0.25],
       [ 4. , -4.1 ],
       [ 5. ,  0.  ]])

```

В некоторые библиотеки уже встроена поддержка pandas, и часть работы они делают автоматически: выполняют преобразование в NumPy из DataFrame и присоединяют имена параметров модели к столбцам выходных таблиц или объектов Series. Если это не так, то такое «управление метаданными» ложится на вас.

В разделе 7.5 мы рассматривали тип pandas `Categorical` и функцию `pandas.get_dummies`. Пусть в нашем наборе данных имеется нечисловой столбец:

```

In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
....:                                     categories=['a', 'b'])

In [25]: data
Out[25]:
   x0  x1  y category
0  1  0.01 -1.5      a
1  2 -0.01  0.0      b
2  3  0.25  3.6      a
3  4 -4.10  1.3      a
4  5  0.00 -2.0      b

```

Если бы мы хотели заменить столбец `'category'` индикаторными переменными, то создали бы индикаторные переменные, удалили столбец `'category'` и выполнили бы операцию соединения:

```

In [26]: dummies = pd.get_dummies(data.category, prefix='category')

In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)

In [28]: data_with_dummies

```

```
Out[28]:
   x0  x1  y  category_a  category_b
0    1  0.01 -1.5         1         0
1    2 -0.01  0.0         0         1
2    3  0.25  3.6         1         0
3    4 -4.10  1.3         1         0
4    5  0.00 -2.0         0         1
```

При аппроксимации некоторых статистических моделей с помощью индикаторных переменных возникают нюансы. Если приходится работать не только с числовыми столбцами, то, пожалуй, проще и безопаснее использовать библиотеку Patsy (тема следующего раздела).

12.2. ОПИСАНИЕ МОДЕЛЕЙ С ПОМОЩЬЮ PATSY

Patsy (<https://patsy.readthedocs.io/en/latest/>) – написанная на Python библиотека для описания статистических моделей (особенно линейных) с применением простого строкового «синтаксиса формул», в основу которого положены формулы из языков статистического программирования R и S (с некоторыми изменениями).

Patsy широко используется в statsmodels для задания линейных моделей, поэтому я расскажу об основных чертах, чтобы вам было проще приступить к работе с ней. Для *формул* в Patsy применяется такой синтаксис:

```
y ~ x0 + x1
```

Конструкция `a + b` – это не сложение `a` и `b`, имеется в виду, что это термы созданной для модели матрицы плана. Функция `patsy.dmatrices` принимает строку формулы вместе с набором данных (который может быть представлен в виде объекта DataFrame или словаря массивов) и порождает матрицы плана для линейной модели:

```
In [29]: data = pd.DataFrame({
.....:     'x0': [1, 2, 3, 4, 5],
.....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
.....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})
```

```
In [30]: data
Out[30]:
   x0  x1  y
0    1  0.01 -1.5
1    2 -0.01  0.0
2    3  0.25  3.6
3    4 -4.10  1.3
4    5  0.00 -2.0
```

```
In [31]: import patsy
```

```
In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

Теперь имеем:

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
   y
0 -1.5
```

```

0.0
3.6
1.3
-2.0
Terms:
'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0    x1
         1   1  0.01
         1   2 -0.01
         1   3  0.25
         1   4 -4.10
         1   5  0.00
Terms:
'Intercept' (column 0)
'x0' (column 1)
'x1' (column 2)

```

Эти объекты класса Patsy `DesignMatrix` являются массивами NumPy `ndarray` с дополнительными метаданными:

```

In [35]: np.asarray(y)
Out[35]:
array([[ -1.5],
       [  0. ],
       [  3.6],
       [  1.3],
       [ -2. ]])

In [36]: np.asarray(X)
Out[36]:
array([[ 1. ,  1. ,  0.01],
       [ 1. ,  2. , -0.01],
       [ 1. ,  3. ,  0.25],
       [ 1. ,  4. , -4.1 ],
       [ 1. ,  5. ,  0.  ]])

```

Вам, наверное, интересно, откуда взялся свободный член – терм `Intercept`. Это соглашение, принятое для таких линейных моделей, как регрессия методом обыкновенных наименьших квадратов. Свободный член можно подавить, добавив в модель терм `+ 0`:

```

In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0    x1
   1  0.01
   2 -0.01
   3  0.25
   4 -4.10
   5  0.00
Terms:
'x0' (column 0)
'x1' (column 1)

```

Объекты Patsy можно передать напрямую в алгоритмы типа `numpy.linalg.lstsq`, который выполняет регрессию методом обыкновенных наименьших квадратов:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

Метаданные модели сохраняются в атрибуте `design_info`, так что имена столбцов модели можно связать с найденными коэффициентами для получения объекта Series, например:

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])
```

```
In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)
```

```
In [41]: coef
Out[41]:
Intercept    0.312910
x0           -0.079106
x1           -0.265464
dtype: float64
```

Преобразование данных в формулах Patsy

В формулы Patsy можно включить код на Python; при вычислении формулы библиотека будет искать использованные функции в объемлющей области видимости:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)
```

```
In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
Intercept  x0  np.log(np.abs(x1) + 1)
1  1          0.00995
1  2          0.00995
1  3          0.22314
1  4          1.62924
1  5          0.00000

Terms:
'Intercept' (column 0)
'x0' (column 1)
'np.log(np.abs(x1) + 1)' (column 2)
```

Из часто используемых преобразований отметим стандартизацию (приведение к распределению со средним 0 и дисперсией 1) и центрирование (вычитание среднего). Для этих целей в Patsy имеются встроенные функции:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)
```

```
In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
Intercept  standardize(x0)  center(x1)
```

1	-1.41421	0.78
1	-0.70711	0.76
1	0.00000	1.02
1	0.70711	-3.33
1	1.41421	0.77

Terms:

```
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)
```

В процессе моделирования мы иногда обучаем модель на одном наборе данных, а затем тестируем на другом. В роли другого набора может выступать *зарезервированная* часть данных или новые данные, полученные позже. Применяя преобразования типа центрирования и стандартизации, следует быть осторожным, когда модель используется для предсказания на новых данных. Говорят, что такие преобразования *обладают состоянием*, потому что при преобразовании нового набора данных мы должны использовать статистики, в частности среднее и стандартное отклонение, исходного набора.

Функция `patsy.build_design_matrices` умеет применять преобразования к новым *вневыборочным* данным, используя информацию, сохраненную для исходного *внутривыборочного* набора данных:

```
In [46]: new_data = pd.DataFrame({
.....:     'x0': [6, 7, 8, 9],
.....:     'x1': [3.1, -0.5, 0, 2.3],
.....:     'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
 Intercept  standardize(x0)  center(x1)
         1          2.12132         3.87
         1          2.82843         0.27
         1          3.53553         0.77
         1          4.24264         3.07

Terms:
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)]
```

Поскольку знак `+` в формулах Patsy не означает сложения, в случае если мы хотим сложить именованные столбцы из набора данных, необходимо обернуть операцию специальной функцией `I`:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
 Intercept  I(x0 + x1)
         1          1.01
         1          1.99
         1          3.25
```



```

      1      -0.10
      1      5.00
Terms:
'Intercept' (column 0)
'I(x0 + x1)' (column 1)

```

В Patsy встроены и другие преобразования, находящиеся в модуле `patsy.builtins`. Дополнительные сведения см. в онлайн-официальной документации.

Для категориальных данных имеется специальный класс преобразований, о котором я расскажу ниже.

Категориальные данные и Patsy

Нечисловые данные можно преобразовать для использования в матрице плана модели многими способами. Полное рассмотрение этой темы выходит за рамки книги и относится скорее к курсу математической статистики.

Когда в формуле Patsy встречаются нечисловые члены, они по умолчанию преобразуются в фиктивные переменные. Если имеется свободный член, то один из уровней будет пропущен, чтобы избежать коллинеарности:

```

In [51]: data = pd.DataFrame({
.....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
.....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],
.....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],
.....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
.....: })

```

```

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

```

```

In [53]: X
Out[53]:
DesignMatrix with shape (8, 2)
Intercept  key1[T.b]
      1      0
      1      0
      1      1
      1      0
      1      1
      1      0
      1      1
      1      1
Terms:
'Intercept' (column 0)
'key1' (column 1)

```

Если исключить из модели свободный член, то столбцы, соответствующие каждому значению категории, будут включены в модельную матрицу плана:

```

In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)

In [55]: X
Out[55]:
DesignMatrix with shape (8, 2)
key1[a]  key1[b]
      1      0
      1      0
      0      1

```

```

0      1
1      0
0      1
1      0
0      1
Terms:
'key1' (columns 0:2)

```

Числовые столбцы можно интерпретировать как категориальные с помощью функции `C`:

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)
```

```

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
Intercept  C(key2)[T.1]
1          0
1          1
1          0
1          1
1          0
1          1
1          0
1          0
Terms:
'Intercept' (column 0)
'C(key2)' (column 1)

```

Если в модели несколько категориальных термов, то ситуация осложняется, поскольку можно включать термы взаимодействия вида `key1:key2`, например в моделях дисперсионного анализа (ANOVA):

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})
```

```

In [59]: data
Out[59]:
  key1 key2  v1  v2
0    a  zero   1 -1.0
1    a  one   2  0.0
2    b  zero   3  2.5
3    b  one   4 -0.5
4    a  zero   5  4.0
5    b  one   6 -1.2
6    a  zero   7  0.2
7    b  zero   8 -1.7

```

```
In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)
```

```

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
Intercept  key1[T.b]  key2[T.zero]
1          0          1
1          0          0
1          1          1
1          1          0

```

```

      1      0      1
      1      1      0
      1      0      1
      1      1      1

```

Terms:

```

'Intercept' (column 0)
'key1' (column 1)
'key2' (column 2)

```

```
In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)
```

```
In [63]: X
```

```
Out[63]:
```

```
DesignMatrix with shape (8, 4)
```

Intercept	key1[T.b]	key2[T.zero]	key1[T.b]:key2[T.zero]
1	0	1	0
1	0	0	0
1	1	1	1
1	1	0	0
1	0	1	0
1	1	0	0
1	0	1	0
1	1	1	1

Terms:

```

'Intercept' (column 0)
'key1' (column 1)
'key2' (column 2)
'key1:key2' (column 3)

```

Patsy предлагает и другие способы преобразования категориальных данных, в т.ч. преобразования для термов в определенном порядке. Дополнительные сведения см. в документации.

12.3. ВВЕДЕНИЕ В STATSMODELS

Statsmodels (<https://www.statsmodels.org/>) – написанная на Python библиотека для обучения разнообразных статистических моделей, выполнения статистических тестов, разведки и визуализации данных. В statsmodels представлены в основном «классические» статистические методы на основе частотного подхода, а байесовские методы и модели машинного обучения лучше искать в других библиотеках.

Перечислим несколько видов моделей, имеющих в statsmodels:

- линейные модели, обобщенные линейные модели и робастные линейные модели;
- линейные модели со смешанными эффектами;
- методы дисперсионного анализа (ANOVA);
- временные ряды и модели в пространстве состояний;
- обобщенные методы моментов.

На следующих страницах мы воспользуемся несколькими базовыми средствами statsmodels и посмотрим, как применять интерфейсы библиотеки моделирования с формулами Patsy и объектами pandas DataFrame. Если вы еще не установили statsmodels, сделайте это сейчас командой

```
conda install statsmodels
```

Оценивание линейных моделей

В библиотеке statsmodels имеется несколько видов моделей линейной регрессии, от самых простых (обычный метод наименьших квадратов) до более сложных (метод наименьших квадратов с итерационным повторным взвешиванием).

Линейные модели в statsmodels имеют два основных интерфейса: на основе массивов и на основе формул. Доступ к ним производится путем импорта следующих модулей:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

Чтобы продемонстрировать их использование, сгенерируем линейную модель по случайным данным. Выполните следующий код в ячейке Jupyter:

```
# Для воспроизводимости результатов
rng = np.random.default_rng(seed=12345)

def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * rng.standard_normal(*size)

N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]

y = np.dot(X, beta) + eps
```

Здесь я записал «истинную» модель с известными параметрами `beta`. В данном случае `dnorm` – вспомогательная функция для генерирования нормально распределенных данных с заданными средним и дисперсией. Таким образом, имеем:

```
In [66]: X[:5]
Out[66]:
array([[ -0.9005,  -0.1894,  -1.0279],
       [  0.7993,  -1.546 ,  -0.3274],
       [-0.5507,  -0.1203,   0.3294],
       [-0.1639,   0.824 ,   0.2083],
       [-0.0477,  -0.2131,  -0.0482]])

In [67]: y[:5]
Out[67]: array([-0.5995, -0.5885,  0.1856, -0.0075, -0.0154])
```

При обучении линейной модели обычно присутствует свободный член, как мы видели ранее при изучении Patsy. Функция `sm.add_constant` может добавить столбец свободного члена в существующую матрицу:

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
```

```
array([[ 1. , -0.9005, -0.1894, -1.0279],
       [ 1. ,  0.7993, -1.546 , -0.3274],
       [ 1. , -0.5507, -0.1203,  0.3294],
       [ 1. , -0.1639,  0.824 ,  0.2083],
       [ 1. , -0.0477, -0.2131, -0.0482]])
```

Класс `sm.OLS` реализует линейную регрессию методом обыкновенных наименьших квадратов:

```
In [70]: model = sm.OLS(y, X)
```

Метод модели `fit` возвращает объект с результатами регрессии, содержащий оценки параметров модели и диагностическую информацию:

```
In [71]: results = model.fit()
```

```
In [72]: results.params
```

```
Out[72]: array([0.0668, 0.268 , 0.4505])
```

Метод `summary` объекта `results` печатает подробную диагностическую информацию о модели:

```
In [73]: print(results.summary())
```

OLS Regression Results

```
=====
Dep. Variable:          y      R-squared (uncentered):      0.469
Model:                OLS    Adj. R-squared (uncentered):    0.452
Method:             Least Squares    F-statistic:          28.51
Date:                Fri, 12 Aug 2022    Prob (F-statistic):    2.66e-13
Time:                14:09:18    Log-Likelihood:       -25.611
No. Observations:      100    AIC:                  57.22
Df Residuals:          97    BIC:                  65.04
Df Model:               3
Covariance Type:       nonrobust
=====
```

	coef	std err	t	P> t	[0.025	0.975]
x1	0.0668	0.054	1.243	0.217	-0.040	0.174
x2	0.2680	0.042	6.313	0.000	0.184	0.352
x3	0.4505	0.068	6.605	0.000	0.315	0.586

```
=====
Omnibus:                0.435    Durbin-Watson:          1.869
Prob(Omnibus):          0.805    Jarque-Bera (JB):        0.301
Skew:                   0.134    Prob(JB):                0.860
Kurtosis:               2.995    Cond. No.                 1.64
=====
```

Notes:

[1] R^2 is computed without centering (uncentered) since the model does not contain a constant.

[2] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Параметрам присвоены обобщенные имена `x1`, `x2` и т. д. Но пусть вместо этого все параметры модели хранятся в объекте `DataFrame`:

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
```

```
In [75]: data['y'] = y
```

```
In [76]: data[:5]
Out[76]:
```

	col0	col1	col2	y
0	-0.900506	-0.189430	-1.027870	-0.599527
1	0.799252	-1.545984	-0.327397	-0.588454
2	-0.550655	-0.120254	0.329359	0.185634
3	-0.163916	0.824040	0.208275	-0.007477
4	-0.047651	-0.213147	-0.048244	-0.015374

Теперь можно использовать формульный API statsmodels и строковые формулы Patsy:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
```

```
In [78]: results.params
Out[78]:
```

Intercept	-0.020799
col0	0.065813
col1	0.268970
col2	0.449419

dtype: float64

```
In [79]: results.tvalues
Out[79]:
```

Intercept	-0.652501
col0	1.219768
col1	6.312369
col2	6.567428

dtype: float64

Заметим, что statsmodels вернула результаты в виде Series, присоединив имена столбцов из DataFrame. Кроме того, при работе с формулами и объектами pandas не нужно использовать `add_constant`.

Зная оценки параметров, мы можем вычислить для них предсказанные моделью значения для новых вневыборочных данных:

```
In [80]: results.predict(data[:5])
Out[80]:
```

0	-0.592959
1	-0.531160
2	0.058636
3	0.283658
4	-0.102947

dtype: float64

В statsmodels существует еще много инструментов для анализа, диагностики и визуализации результатов линейных моделей – изучайте, сколько душе угодно. Имеются также другие виды линейных моделей, помимо метода обыкновенных наименьших квадратов.

Оценивание процессов с временными рядами

Еще один класс моделей в statsmodels предназначен для анализа временных рядов. В их числе процессы авторегрессии, фильтры Калмана и другие модели в пространстве состояний, а также многомерные авторегрессионные модели.

Смоделируем данные временного ряда с авторегрессионной структурой и шумом. Выполните следующий код в ячейке Jupyter:

```
init_x = 4

values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

У этих данных структура AR(2) (два лага) с параметрами 0.8 и -0.4. Когда аппроксимируется AR-модель, мы не всегда знаем, сколько членов лага включать, поэтому можем аппроксимировать модель с несколько большим числом лагов:

```
In [82]: from statsmodels.tsa.ar_model import AutoReg

In [83]: MAXLAGS = 5

In [84]: model = AutoReg(values, MAXLAGS)

In [85]: results = model.fit()
```

В оценках параметров в `results` сначала идет свободный член, а затем оценки двух первых лагов:

```
In [86]: results.params
Out[86]: array([ 0.0235, 0.8097, -0.4287, -0.0334, 0.0427, -0.0567])
```

В этой книге я не могу углубиться в детали этих моделей и интерпретации результатов, но в документации по `statsmodels` вас ждет много открытий.

12.4. ВВЕДЕНИЕ В SCIKIT-LEARN

Библиотека `scikit-learn` (<https://scikit-learn.org/>) – один из самых популярных и пользующихся доверием универсальных инструментов машинного обучения на Python. Она содержит широкий спектр стандартных методов машинного обучения с учителем и без учителя, предназначенных для выбора и оценки модели, загрузки и преобразования данных и сохранения моделей. Эти модели можно использовать для классификации, кластеризации, предсказания и решения других типичных задач. Для установки `scikit-learn` выполните команду

```
conda install scikit-learn
```

Существуют прекрасные онлайн-ресурсы, из которых можно узнать о машинном обучении и применении таких библиотек, как `scikit-learn` и `TensorFlow`, к решению реальных задач. В этом разделе я дам лишь краткое представление о стилистических особенностях API `scikit-learn`.

За последние годы интеграция `pandas` `scikit-learn` заметно улучшилась, а к тому моменту, когда вы будете читать эту книгу, она, вероятно, станет еще лучше. Настоятельно рекомендую ознакомиться с актуальной документацией по проекту.

В качестве примера я возьму ставший уже классическим набор данных с конкурса Kaggle (<https://www.kaggle.com/c/titanic>), содержащий сведения о пассажирах «Титаника», затонувшего в 1912 году. Загрузим обучающий и тестовый наборы с помощью pandas:

```
In [87]: train = pd.read_csv('datasets/titanic/train.csv')
```

```
In [88]: test = pd.read_csv('datasets/titanic/test.csv')
```

```
In [89]: train[:4]
```

```
Out[89]:
```

	PassengerId	Survived	Pclass \
0	1	0	3
1	2	1	1
2	3	1	3
3	4	1	1

		Name	Sex	Age	SibSp \
0		Braund, Mr. Owen Harris	male	22.0	1
1	Cumings, Mrs. John Bradley (Florence Briggs Th...		female	38.0	1
2		Heikkinen, Miss. Laina	female	26.0	0
3	Futelle, Mrs. Jacques Heath (Lily May Peel)		female	35.0	1

	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
1	0	PC 17599	71.2833	C85	C
2	0	STON/O2. 3101282	7.9250	NaN	S
3	0	113803	53.1000	C123	S

Библиотеки типа statsmodels и scikit-learn, вообще говоря, не допускают подачи на вход неполных данных, поэтому посмотрим, в каких столбцах есть отсутствующие данные:

```
In [90]: train.isna().sum()
```

```
Out[90]:
```

PassengerId	0
Survived	0
Pclass	0
Name	0
Sex	0
Age	177
SibSp	0
Parch	0
Ticket	0
Fare	0
Cabin	687
Embarked	2

dtype: int64

```
In [91]: test.isna().sum()
```

```
Out[91]:
```

PassengerId	0
Pclass	0
Name	0
Sex	0
Age	86
SibSp	0


```
Parch          0
Ticket         0
Fare           1
Cabin         327
Embarked       0
dtype: int64
```

В статистике и машинном обучении типичная задача заключается в том, чтобы предсказать, выживет ли пассажир, на основе признаков, содержащихся в данных. Модель обучается на *обучающем* наборе данных, а затем оценивается на не пересекающемся с ним *тестовом* наборе данных.

Я хотел бы использовать в качестве предсказательного признака возраст `Age`, но в этом столбце есть отсутствующие данные. Существует несколько способов *восполнить* отсутствующие данные, я выберу самый простой и заменю значения `null` в обеих таблицах медианными значениями для обучающего набора:

```
In [92]: impute_value = train['Age'].median()

In [93]: train['Age'] = train['Age'].fillna(impute_value)

In [94]: test['Age'] = test['Age'].fillna(impute_value)
```

Теперь мы должны описать модель. Я добавлю столбец `IsFemale`, вычисляемый на основе столбца `'Sex'`:

```
In [95]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)

In [96]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

Выберем переменные модели и создадим массивы NumPy:

```
In [97]: predictors = ['Pclass', 'IsFemale', 'Age']

In [98]: X_train = train[predictors].to_numpy()

In [99]: X_test = test[predictors].to_numpy()

In [100]: y_train = train['Survived'].to_numpy()
```

```
In [101]: X_train[:5]
Out[101]:
array([[ 3.,  0., 22.],
       [ 1.,  1., 38.],
       [ 3.,  1., 26.],
       [ 1.,  1., 35.],
       [ 3.,  0., 35.]])
```

```
In [102]: y_train[:5]
Out[102]: array([0, 1, 1, 1, 0])
```

Я вовсе не утверждаю, что это хорошая модель или что признаки сконструированы правильно. Мы воспользуемся моделью `LogisticRegression` из `scikit-learn` и создадим ее экземпляр:

```
In [103]: from sklearn.linear_model import LogisticRegression

In [104]: model = LogisticRegression()
```

Так же, как в statsmodels, мы можем обучить модель на обучающих данных с помощью метода модели `fit`:

```
In [105]: model.fit(X_train, y_train)
Out[105]: LogisticRegression()
```

Теперь можно вычислить предсказания для тестового набора данных методом `model.predict`:

```
In [106]: y_predict = model.predict(X_test)

In [107]: y_predict[:10]
Out[107]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

Если бы мы знали истинные значения для тестового набора, то могли бы вычислить верность в процентах или еще какую-нибудь метрику ошибки:

```
(y_true == y_predict).mean()
```

На практике обучение модели гораздо сложнее. У многих моделей есть настраиваемые параметры, и существуют такие приемы, как *перекрестный контроль*, позволяющие избежать переобучения модели. Их использование часто повышает предсказательную способность или робастность модели на новых данных.

Идея перекрестного контроля состоит в том, чтобы разделить обучающий набор на две части с целью смоделировать предсказание на не предъявлявшихся модели данных. Взяв за основу какую-нибудь оценку верности модели, например среднеквадратическую ошибку, можно произвести подбор параметров модели путем поиска на сетке. Для некоторых моделей, в частности логистической регрессии, существуют классы оценивания, в которые перекрестный контроль уже встроен. Например, класс `LogisticRegressionCV` можно использовать, задав в качестве параметра мелкость сетки, на которой производится поиск регуляризирующего параметра модели `C`:

```
In [108]: from sklearn.linear_model import LogisticRegressionCV

In [109]: model_cv = LogisticRegressionCV(Cs=10)

In [110]: model_cv.fit(X_train, y_train)
Out[110]: LogisticRegressionCV()
```

Чтобы выполнить перекрестный контроль вручную, можно воспользоваться вспомогательной функцией `cross_val_score`, которая отвечает за процесс разделения данных. Например, чтобы подвергнуть нашу модель перекрестному контролю, разбив обучающий набор на четыре непересекающиеся части, нужно написать:

```
In [111]: from sklearn.model_selection import cross_val_score

In [112]: model = LogisticRegression(C=10)

In [113]: scores = cross_val_score(model, X_train, y_train, cv=4)

In [114]: scores
Out[114]: array([0.7758, 0.7982, 0.7758, 0.7883])
```

Подразумеваемая по умолчанию метрика оценивания зависит от модели, но можно задать функцию оценивания явно. Обучение с перекрестным контролем занимает больше времени, но часто дает модель лучшего качества.

12.5. ЗАКЛЮЧЕНИЕ

Я лишь очень поверхностно описал некоторые библиотеки моделирования на Python, в то время как существуют многочисленные системы для различных видов статистической обработки и машинного обучения, либо написанные на Python, либо имеющие интерфейс из Python.

В этой книге речь идет прежде всего о переформатировании данных, но есть много других, посвященных моделированию и инструментарию науки о данных. Перечислим некоторые из них:

- Andreas Mueller and Sarah Guido «Introduction to Machine Learning with Python» (O'Reilly)⁶;
- Jake VanderPlas «Python Data Science Handbook» (O'Reilly)⁷;
- Joel Grus «Data Science from Scratch: First Principles with Python» (O'Reilly)⁸;
- Sebastian Raschka «Python Machine Learning» (Packt Publishing)⁹;
- Aurelien Geron «Hands-On Machine Learning with Scikit-Learn and TensorFlow» (O'Reilly)¹⁰.

Книги, конечно, – ценное подспорье для изучения предмета, но иногда они устаревают, поскольку программное обеспечение с открытым исходным кодом быстро изменяется. Чтобы всегда быть в курсе последних возможностей и версии API, имеет прямой смысл почитать документацию по различным системам статистики и машинного обучения.

⁶ *Андреас Мюллер, Сара Гвидо. Введение в машинное обучение с помощью Python. Вильямс, 2017.*

⁷ *Джейк Вандер Плас. Python для сложных задач. Наука о данных и машинное обучение. Питер, 2018.*

⁸ *Джозел Грас. Data Science. Наука о данных с нуля. БХВ-Петербург, 2019.*

⁹ *Себастьян Рашка. Python и машинное обучение. ДМК Пресс, 2017.*

¹⁰ *Орельен Жерон. Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow. Вильямс, 2018.*

Примеры анализа данных

Вот мы и дошли до последней главы, где рассмотрим несколько реальных наборов данных. Для каждого набора мы применим описанные в книге приемы, чтобы извлечь смысл из исходных данных. Продемонстрированная техника пригодна и для любых других наборов данных, в т. ч. ваших собственных. Я включил различные примеры наборов данных, на которых можно попрактиковаться в применении описанных в книге инструментов.

Все примеры наборов данных имеются в репозитории этой книги на GitHub (<http://github.com/wesm/pydata-book>). Если у вас нет доступа к GitHub, можете скачать их с зеркала на Gitee (<https://gitee.com/wesmckinn/pydata-book>).

13.1. НАБОР ДАННЫХ BITLY С САЙТА 1.U.SA.GOV

В 2011 году служба сокращения URL-адресов bit.ly заключила партнерское соглашение с сайтом правительства США USA.gov (<https://www.usa.gov/>) о синхронном предоставлении анонимных данных о пользователях, которые сокращают ссылки, заканчивающиеся на .gov или .mil. В 2011 году, помимо синхронной ленты, формировались ежечасные мгновенные снимки, доступные в виде текстовых файлов. В 2017 году эта служба уже закрылась, но мы сохранили файлы данных и приводим их в качестве примеров.

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

```
In [5]: path = "datasets/bitly_usagov/example.txt"

In [6]: with open(path) as f:
...:     print(f.readline())
...:
{ "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": "A6q0VH", "h": "wFLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\\\www.facebook.com\\l\\7AQEFzjSi\\1.usa.gov\\wFLQtf", "u":
"http:\\\\www.ncbi.nlm.nih.gov\\pubmed\\22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }
```

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
with open(path) as f:
    records = [json.loads(line) for line in f]
```

Получившийся в результате объект `records` представляет собой список словарей Python:

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
 'gr': 'MA',
 'h': 'wFLQtF',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/L/7AQEFzjSi/1.usa.gov/wFLQtF',
 't': 1331923247,
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

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. В первых, можно извлечь список часовых поясов, снова воспользовавшись списковым включением:

```
In [15]: time_zones = [rec["tz"] for rec in records]
-----
KeyError                                Traceback (most recent call last)
<ipython-input-15-abdeba901c13> in <module>
----> 1 time_zones = [rec["tz"] for rec in records]
<ipython-input-15-abdeba901c13> in <listcomp>(.0)
----> 1 time_zones = [rec["tz"] for rec in records]
KeyError: 'tz'
```

Вот те раз! Оказывается, что не во всех записях есть поле часового пояса. Это легко поправить, добавив проверку `if 'tz' in rec` в конец спискового включения:

```
In [16]: time_zones = [rec["tz"] for rec in records if "tz" in rec]

In [17]: time_zones[:10]
Out[17]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
```

```
''',
''',
''']
```

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

Воспользовавшись более продвинутыми средствами из стандартной библиотеки Python, можно записать то же самое короче:

```
from collections import defaultdict
def get_counts2(sequence):
    counts = defaultdict(int) # 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 [20]: counts = get_counts(time_zones)
```

```
In [21]: counts["America/New_York"]
Out[21]: 1251
```

```
In [22]: len(time_zones)
Out[22]: 3440
```

Чтобы получить только первые 10 часовых поясов со счетчиками, можно построить список кортежей (`count`, `timezone`) и отсортировать его:

```
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 [24]: top_counts(counts)
Out[24]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
```

```
(74, 'Europe/London'),
(191, 'America/Denver'),
(382, 'America/Los_Angeles'),
(400, 'America/Chicago'),
(521, ''),
(1251, 'America/New_York')]
```

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

```
In [25]: from collections import Counter
```

```
In [26]: counts = Counter(time_zones)
```

```
In [27]: counts.most_common(10)
```

```
Out[27]:
```

```
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

Подсчет часовых поясов с помощью pandas

Чтобы создать экземпляр `DataFrame` из исходного набора записей, следует передать список записей функции `pandas.DataFrame`:

```
In [28]: frame = pd.DataFrame(records)
```

Мы можем получить базовую информацию об этом новом объекте `DataFrame`, в частности имена столбцов, выведенные типы столбцов и количество отсутствующих значений, воспользовавшись методом `frame.info()`:

```
In [29]: frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 3560 entries, 0 to 3559
```

```
Data columns (total 18 columns):
```

#	Column	Non-Null Count	Dtype
0	a	3440 non-null	object
1	c	2919 non-null	object
2	nk	3440 non-null	float64
3	tz	3440 non-null	object
4	gr	2919 non-null	object
5	g	3440 non-null	object
6	h	3440 non-null	object
7	l	3440 non-null	object
8	al	3094 non-null	object
9	hh	3440 non-null	object
10	r	3440 non-null	object
11	u	3440 non-null	object
12	t	3440 non-null	float64

```

13 hc          3440 non-null float64
14 cy          2919 non-null object
15 ll          2919 non-null object
16 _heartbeat_ 120 non-null float64
17 kw          93 non-null  object
dtypes: float64(4), object(14)
memory usage: 500.8+ KB

```

```

In [30]: frame["tz"].head()
Out[30]:
0    America/New_York
1    America/Denver
2    America/New_York
3    America/Sao_Paulo
4    America/New_York
Name: tz, dtype: object

```

На выходе по запросу `frame` мы видим *сводное представление*, которое показывается для больших объектов DataFrame. Затем можно воспользоваться методом `value_counts` объекта Series:

```

In [31]: tz_counts = frame["tz"].value_counts()

In [32]: tz_counts.head()
Out[32]:
America/New_York    1251
                  521
America/Chicago      400
America/Los_Angeles  382
America/Denver       191
Name: tz, dtype: int64

```

Эти данные можно визуализировать с помощью библиотеки `matplotlib`. Графики можно сделать немного приятнее, подставив какое-нибудь значение вместо неизвестных или отсутствующих часовых поясов. Мы заменяем отсутствующие значения методом `fillna`, а для пустых строк используем индексирование булевым массивом:

```

In [33]: clean_tz = frame["tz"].fillna("Missing")

In [34]: clean_tz[clean_tz == ""] = "Unknown"

In [35]: tz_counts = clean_tz.value_counts()

In [36]: tz_counts.head()
Out[36]:
America/New_York    1251
Unknown             521
America/Chicago      400
America/Los_Angeles  382
America/Denver       191
Name: tz, dtype: int64

```

Теперь можно воспользоваться пакетом `seaborn` для построения горизонтальной столбчатой диаграммы (результат показан на рис. 13.1):


```
In [38]: import seaborn as sns

In [39]: subset = tz_counts.head()

In [40]: sns.barplot(y=subset.index, x=subset.to_numpy())
```

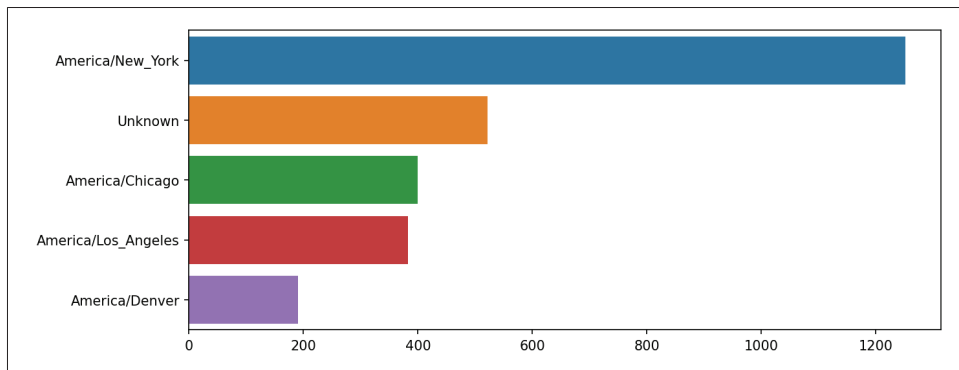


Рис. 13.1. Первые 10 часовых поясов из набора данных 1.usa.gov

Поле `a` содержит информацию о браузере, устройстве или приложении, выполнившем сокращение URL:

```
In [41]: frame["a"][1]
Out[41]: 'GoogleMaps/RochesterNY'

In [42]: frame["a"][50]
Out[42]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [43]: frame["a"][51][:50] # long line
Out[43]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

Выделение всей интересной информации из таких строк «пользовательских агентов» поначалу может показаться пугающей задачей. Одна из возможных стратегий – вырезать из строки первую лексему (грубо описывающую возможности браузера) и представить поведение пользователя в другом разрезе:

```
In [44]: results = pd.Series([x.split()[0] for x in frame["a"].dropna()])

In [45]: results.head(5)
Out[45]:
0      Mozilla/5.0
1  GoogleMaps/RochesterNY
2      Mozilla/4.0
3      Mozilla/5.0
4      Mozilla/5.0
dtype: object

In [46]: results.value_counts().head(8)
Out[46]:
Mozilla/5.0      2594
Mozilla/4.0       601
GoogleMaps/RochesterNY    121
```

```
Opera/9.80          34
TEST_INTERNET_AGENT 24
GoogleProducer      21
Mozilla/6.0         5
BlackBerry8520/5.0.0.681 4
dtype: int64
```

Предположим теперь, что требуется разделить пользователей в первых 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 исключю:

```
In [47]: cframe = frame[frame["a"].notna()].copy()
```

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

```
In [48]: cframe["os"] = np.where(cframe["a"].str.contains("Windows"),
.....:                          "Windows", "Not Windows")
```

```
In [49]: cframe["os"].head(5)
```

```
Out[49]:
```

```
0    Windows
1    Not Windows
2    Windows
3    Not Windows
4    Windows
```

```
Name: os, dtype: object
```

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

```
In [50]: by_tz_os = cframe.groupby(["tz", "os"])
```

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 [51]: agg_counts = by_tz_os.size().unstack().fillna(0)
```

```
In [52]: agg_counts.head()
```

```
Out[52]:
```

os	Not Windows	Windows
tz		
	245.0	276.0
Africa/Cairo	0.0	3.0
Africa/Casablanca	0.0	1.0
Africa/Ceuta	0.0	2.0
Africa/Johannesburg	0.0	1.0

Наконец, выберем из полученной таблицы первые 10 часовых поясов. Для этого я построю косвенный индексный массив по счетчикам строк в `agg_counts`. После вычисления счетчиков строк с помощью `agg_counts.sum("columns")` я могу вызвать `argsort()` и получить индексный массив, который можно будет использовать для сортировки в порядке возрастания:

```
In [53]: indexer = agg_counts.sum("columns").argsort()

In [54]: indexer.values[:10]
Out[54]: array([24, 20, 21, 92, 87, 53, 54, 57, 26, 55])
```

Я воспользуюсь методом `take`, чтобы выбрать строки в этом порядке, и оставлю только последние 10 строк (с наибольшими значениями):

```
In [55]: count_subset = agg_counts.take(indexer[-10:])

In [56]: count_subset
Out[56]:
os                Not Windows  Windows
tz
America/Sao_Paulo          13.0     20.0
Europe/Madrid              16.0     19.0
Pacific/Honolulu           0.0     36.0
Asia/Tokyo                  2.0     35.0
Europe/London              43.0     31.0
America/Denver            132.0     59.0
America/Los_Angeles       130.0    252.0
America/Chicago           115.0    285.0
                        245.0    276.0
America/New_York          339.0    912.0
```

В `pandas` имеется вспомогательный метод `nlargest`, который делает то же самое:

```
In [57]: agg_counts.sum(axis="columns").nlargest(10)
Out[57]:
tz
America/New_York          1251.0
                        521.0
America/Chicago           400.0
America/Los_Angeles       382.0
America/Denver            191.0
Europe/London              74.0
Asia/Tokyo                 37.0
Pacific/Honolulu           36.0
Europe/Madrid              35.0
America/Sao_Paulo          33.0
dtype: float64
```

Теперь это можно визуализировать с помощью групповой столбчатой диаграммы для сравнения количества пользователей Windows и прочих; воспользуемся функцией `barplot` из библиотеки `seaborn` (см. рис. 13.2). Сначала я вызываю `count_subset.stack()`, а затем перестраиваю индекс, чтобы привести данные к форме, лучше совместимой с `seaborn`:

```
In [59]: count_subset = count_subset.stack()

In [60]: count_subset.name = "total"

In [61]: count_subset = count_subset.reset_index()

In [62]: count_subset.head(10)
Out[62]:
```

	tz	os	total
0	America/Sao_Paulo	Not Windows	13.0
1	America/Sao_Paulo	Windows	20.0
2	Europe/Madrid	Not Windows	16.0
3	Europe/Madrid	Windows	19.0
4	Pacific/Honolulu	Not Windows	0.0
5	Pacific/Honolulu	Windows	36.0
6	Asia/Tokyo	Not Windows	2.0
7	Asia/Tokyo	Windows	35.0
8	Europe/London	Not Windows	43.0
9	Europe/London	Windows	31.0

```
In [63]: sns.barplot(x='total', y='tz', hue='os', data=count_subset)
```

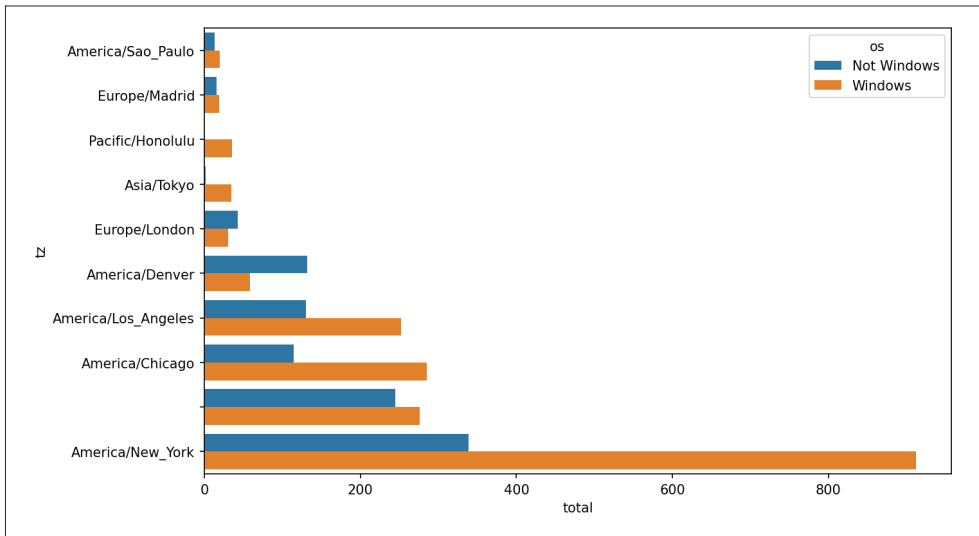


Рис. 13.2. Первые 10 часовых поясов с выделением пользователей Windows и прочих

Из этой диаграммы трудно понять, какова процентная доля пользователей Windows в небольших группах, поэтому нормируем процентные доли групп, так чтобы в сумме получилась 1:

```
def norm_total(group):
    group["normed_total"] = group["total"] / group["total"].sum()
    return group
```

```
results = count_subset.groupby("tz").apply(norm_total)
```

Новая диаграмма показана на рис. 13.3:

```
In [66]: sns.barplot(x="normed_total", y="tz", hue="os", data=results)
```

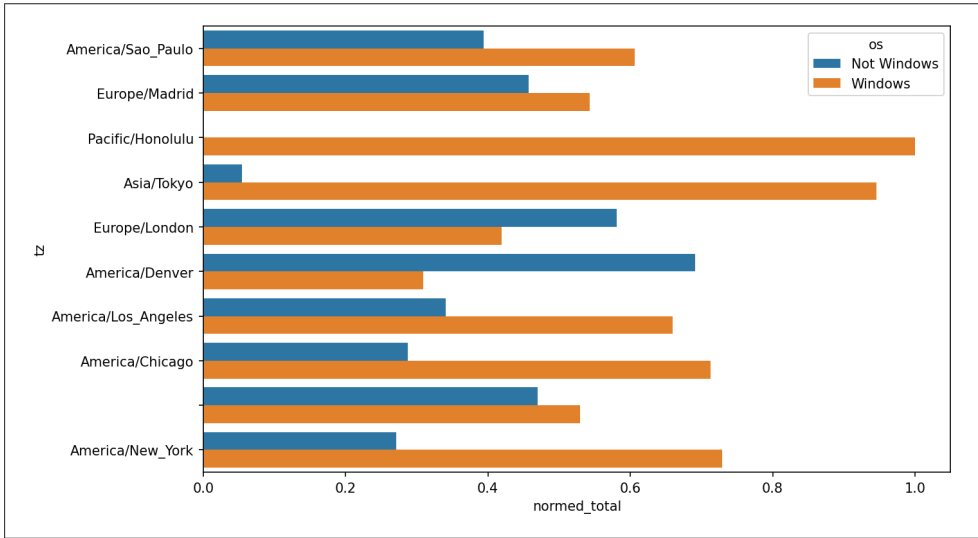


Рис. 13.3. Процентная доля пользователей Windows и прочих в первых 10 часовых поясах

Нормированную сумму можно было бы вычислить эффективнее, воспользовавшись методом `transform` в сочетании с `groupby`:

```
In [67]: g = count_subset.groupby("tz")
```

```
In [68]: results2 = count_subset["total"] / g["total"].transform("sum")
```

13.2. НАБОР ДАННЫХ MOVIELENS 1M

Исследовательская группа GroupLens Research (<https://grouplens.org/datasets/movielens/>) предлагает несколько наборов данных о рейтингах фильмов, предоставленных 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 рекомендаций, основанных на алгоритмах машинного обучения. И хотя в этой книге методы машинного обучения не рассматриваются, я все же покажу, как формировать продольные и поперечные разрезы таких наборов данных с целью привести их к нужному виду.

Набор MovieLens 1M содержит 1 млн рейтингов 4000 фильмов, предоставленных 6000 пользователями. Данные распределены по трем таблицам: рейтинги, информация о пользователях и информация о фильмах. Каждую таблицу можно загрузить в отдельный объект DataFrame с помощью метода `pandas.read_table`. Выполните следующий код в ячейке Jupyter:

```
unames = ["user_id", "gender", "age", "occupation", "zip"]
users = pd.read_table("datasets/movielens/users.dat", sep="::",
                     header=None, names=unames, engine="python")
rnames = ["user_id", "movie_id", "rating", "timestamp"]
```

```

ratings = pd.read_table("datasets/movielens/ratings.dat", sep="::",
                        header=None, names=rnames, engine="python")
mnames = ["movie_id", "title", "genres"]
movies = pd.read_table("datasets/movielens/movies.dat", sep="::",
                       header=None, names=mnames, engine="python")

```

Проверить, что все получилось, можно, взглянув на каждый объект DataFrame:

```
In [70]: users.head(5)
```

```
Out[70]:
```

	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 [71]: ratings.head(5)
```

```
Out[71]:
```

	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 [72]: movies.head(5)
```

```
Out[72]:
```

	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 [73]: ratings
```

```
Out[73]:
```

	user_id	movie_id	rating	timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291
...	...	...	...	...
1000204	6040	1091	1	956716541
1000205	6040	1094	5	956704887
1000206	6040	562	5	956704746
1000207	6040	1096	4	956715648
1000208	6040	1097	4	956715569

[1000209 rows x 4 columns]

Отметим, что возраст и род занятий кодируются целыми числами, а расшифровка приведена в прилагаемом к набору данных файлу [README](#). Анализ данных, хранящихся в трех таблицах, – непростая задача. Пусть, например, требуется вычислить средние рейтинги для конкретного фильма в разрезе

пола и возраста. Как мы увидим, это гораздо легче сделать, если предварительно 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` определяет, по каким столбцам объединять (или *соединять*), ориентируясь на совпадение имен:

```
In [74]: data = pd.merge(pd.merge(ratings, users), movies)
```

```
In [75]: data
```

```
Out[75]:
```

	user_id	movie_id	rating	timestamp	gender	age	occupation	zip \
0	1	1193	5	978300760	F	1	10	48067
1	2	1193	5	978298413	M	56	16	70072
2	12	1193	4	978220179	M	25	12	32793
3	15	1193	4	978199279	M	25	7	22903
4	17	1193	5	978158471	M	50	1	95350
...	...	...	...	...	...	...	...	...
1000204	5949	2198	5	958846401	M	18	17	47901
1000205	5675	2703	3	976029116	M	35	14	30030
1000206	5780	2845	1	958153068	M	18	17	92886
1000207	5851	3607	5	957756608	F	18	20	55410
1000208	5938	2909	4	957273353	M	25	1	35401
0	One Flew Over the Cuckoo's Nest (1975)						genres	
1	One Flew Over the Cuckoo's Nest (1975)						Drama	
2	One Flew Over the Cuckoo's Nest (1975)						Drama	
3	One Flew Over the Cuckoo's Nest (1975)						Drama	
4	One Flew Over the Cuckoo's Nest (1975)						Drama	
...	...						...	
1000204	Modulations (1998)						Documentary	
1000205	Broken Vessels (1998)						Drama	
1000206	White Boys (1999)						Drama	
1000207	One Little Indian (1973)						Comedy Drama Western	
1000208	Five Wives, Three Secretaries and Me (1998)						Documentary	

```
[1000209 rows x 10 columns]
```

```
In [76]: data.iloc[0]
```

```
Out[76]:
```

user_id	1
movie_id	1193
rating	5
timestamp	978300760
gender	F
age	1
occupation	10
zip	48067
title	One Flew Over the Cuckoo's Nest (1975)
genres	Drama

```
Name: 0, dtype: object
```

Чтобы получить средние рейтинги каждого фильма по группам зрителей одного пола, воспользуемся методом `pivot_table`:

```
In [77]: mean_ratings = data.pivot_table("rating", index="title",
....:                                   columns="gender", aggfunc="mean")

In [78]: mean_ratings.head(5)
Out[78]:
gender                F                M
title
$1,000,000 Duck (1971)    3.375000  2.761905
'Night Mother (1986)    3.388889  3.352941
'Til There Was You (1997) 2.675676  2.733333
'burbs, The (1989)      2.793478  2.962085
...And Justice for All (1979) 3.828571  3.689024
```

В результате получается еще один объект DataFrame, содержащий средние рейтинги, в котором метками строк («индексом») являются названия фильмов, а метками столбцов – обозначения полов. Сначала я оставлю только фильмы, получившие не менее 250 оценок (число выбрано совершенно произвольно); для этого сгруппирую данные по названию и с помощью метода `size()` получу объект Series, содержащий размеры групп для каждого наименования:

```
In [79]: ratings_by_title = data.groupby("title").size()

In [80]: ratings_by_title.head()
Out[80]:
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
dtype: int64

In [81]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [82]: active_titles
Out[82]:
Index([''burbs, The (1989)', '10 Things I Hate About You (1999)',
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Men (1957)',
       '13th Warrior, The (1999)', '2 Days in the Valley (1996)',
       '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey (1968)',
       '2010 (1984)',
       ...,
       'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1998)',
       'Young Frankenstein (1974)', 'Young Guns (1988)',
       'Young Guns II (1990)', 'Young Sherlock Holmes (1985)',
       'Zero Effect (1998)', 'eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

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

```
In [83]: mean_ratings = mean_ratings.loc[active_titles]

In [84]: mean_ratings
Out[84]:
```



```

gender F M
title
'burbs, The (1989)          2.793478 2.962085
10 Things I Hate About You (1999) 3.646552 3.311966
101 Dalmatians (1961)       3.791444 3.500000
101 Dalmatians (1996)       3.240000 2.911215
12 Angry Men (1957)         4.184397 4.328421
...
Young Guns (1988)           3.371795 3.425620
Young Guns II (1990)        2.934783 2.904025
Young Sherlock Holmes (1985) 3.514706 3.363344
Zero Effect (1998)          3.864407 3.723140
eXistenZ (1999)             3.098592 3.289086
[1216 rows x 2 columns]

```

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

```

In [86]: top_female_ratings = mean_ratings.sort_values("F", ascending=False)

In [87]: top_female_ratings.head()
Out[87]:
gender                                     F          M
title
Close Shave, A (1995)                    4.644444 4.473795
Wrong Trousers, The (1993)                4.588235 4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950) 4.572650 4.464589
Wallace & Gromit: The Best of Aardman Animation (1996) 4.563107 4.385075
Schindler's List (1993)                   4.562602 4.491415

```

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 [88]: mean_ratings["diff"] = mean_ratings["M"] - mean_ratings["F"]
```

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

```

In [89]: sorted_by_diff = mean_ratings.sort_values("diff")

In [90]: sorted_by_diff.head()
Out[90]:
gender          F          M      diff
title
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

```

Изменив порядок строк на противоположный и снова отобрав первые 10 строк, мы получим фильмы, которым мужчины поставили высокие, а женщины – низкие оценки:

```
In [91]: sorted_by_diff[::-1].head()
Out[91]:
```

	F	M	diff
gender			
title			
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

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. Чтобы найти его, сначала вычислим стандартное отклонение оценок, сгруппированных по названию, а потом оставим только активные названия:

```
In [92]: rating_std_by_title = data.groupby("title")["rating"].std()

In [93]: rating_std_by_title = rating_std_by_title.loc[active_titles]

In [94]: rating_std_by_title.head()
Out[94]:
```

title	
'burbs, The (1989)	1.107760
10 Things I Hate About You (1999)	0.989815
101 Dalmatians (1961)	0.982103
101 Dalmatians (1996)	1.098717
12 Angry Men (1957)	0.812731

Name: rating, dtype: float64

Затем отсортируем в порядке убывания и отберем первые 10 строк, это и будут десять фильмов (в первом приближении) с наиболее сильно различающимися оценками:

```
In [95]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[95]:
```

title	
Dumb & Dumber (1994)	1.321333
Blair Witch Project, The (1999)	1.316368
Natural Born Killers (1994)	1.307198
Tank Girl (1995)	1.277695
Rocky Horror Picture Show, The (1975)	1.260177
Eyes Wide Shut (1999)	1.259624
Evita (1996)	1.253631
Billy Madison (1995)	1.249970
Fear and Loathing in Las Vegas (1998)	1.246408
Bicentennial Man (1999)	1.245533

Name: rating, dtype: float64

Вы, наверное, обратили внимание, что жанры фильма разделяются вертикальной чертой (|), поскольку один фильм может относиться к нескольким жанрам. Чтобы сгруппировать оценки по жанрам, мы можем воспользоваться методом `explode` объекта `DataFrame`. Посмотрим, как он работает. Сначала преобразуем строку жанров в список жанров, воспользовавшись методом `str.split` объекта `Series`.

```

In [96]: movies["genres"].head()
Out[96]:
0      Animation|Children's|Comedy
1      Adventure|Children's|Fantasy
2                      Comedy|Romance
3                      Comedy|Drama
4                      Comedy
Name: genres, dtype: object

In [97]: movies["genres"].head().str.split("|")
Out[97]:
0      [Animation, Children's, Comedy]
1      [Adventure, Children's, Fantasy]
2                      [Comedy, Romance]
3                      [Comedy, Drama]
4                      [Comedy]
Name: genres, dtype: object

In [98]: movies["genre"] = movies.pop("genres").str.split("|")

In [99]: movies.head()
Out[99]:
   movie_id  title \
0         1  Toy Story (1995)
1         2    Jumanji (1995)
2         3  Grumpier Old Men (1995)
3         4  Waiting to Exhale (1995)
4         5  Father of the Bride Part II (1995)
   genre
0  [Animation, Children's, Comedy]
1  [Adventure, Children's, Fantasy]
2          [Comedy, Romance]
3          [Comedy, Drama]
4          [Comedy]

```

Далее вызов `movies.explode("genre")` генерирует новый объект `DataFrame`, содержащий по одной строке для каждого «внутреннего» элемента в каждом списке жанров. Например, если фильм классифицирован как комедия и мелодрама, то в результирующем объекте будет две строки: `"Comedy"` и `"Romance"`:

```

In [100]: movies_exploded = movies.explode("genre")

In [101]: movies_exploded[:10]
Out[101]:
   movie_id  title  genre
0         1  Toy Story (1995)  Animation
0         1  Toy Story (1995)  Children's
0         1  Toy Story (1995)   Comedy
1         2    Jumanji (1995)  Adventure
1         2    Jumanji (1995)  Children's
1         2    Jumanji (1995)  Fantasy
2         3  Grumpier Old Men (1995)   Comedy
2         3  Grumpier Old Men (1995)  Romance
3         4  Waiting to Exhale (1995)   Comedy
3         4  Waiting to Exhale (1995)   Drama

```

Теперь можно объединить все три таблицы и сгруппировать по жанру:

```
In [102]: ratings_with_genre = pd.merge(pd.merge(movies_exploded, ratings), users
)
```

```
In [103]: ratings_with_genre.iloc[0]
```

```
Out[103]:
movie_id      1
title      Toy Story (1995)
genre      Animation
user_id      1
rating      5
timestamp    978824268
gender      F
age      1
occupation    10
zip      48067
Name: 0, dtype: object
```

```
In [104]: genre_ratings = (ratings_with_genre.groupby(["genre", "age"])
.....:                  ["rating"].mean()
.....:                  .unstack("age"))
```

```
In [105]: genre_ratings[:10]
```

```
Out[105]:
age      1      18      25      35      45      50 \
genre
Action    3.506385  3.447097  3.453358  3.538107  3.528543  3.611333
Adventure  3.449975  3.408525  3.443163  3.515291  3.528963  3.628163
Animation  3.476113  3.624014  3.701228  3.740545  3.734856  3.780020
Children's 3.241642  3.294257  3.426873  3.518423  3.527593  3.556555
Comedy     3.497491  3.460417  3.490385  3.561984  3.591789  3.646868
Crime      3.710170  3.668054  3.680321  3.733736  3.750661  3.810688
Documentary 3.730769  3.865865  3.946690  3.953747  3.966521  3.908108
Drama      3.794735  3.721930  3.726428  3.782512  3.784356  3.878415
Fantasy    3.317647  3.353778  3.452484  3.482301  3.532468  3.581570
Film-Noir  4.145455  3.997368  4.058725  4.064910  4.105376  4.175401
age      56
genre
Action    3.610709
Adventure  3.649064
Animation  3.756233
Children's 3.621822
Comedy     3.650949
Crime      3.832549
Documentary 3.961538
Drama      3.933465
Fantasy    3.532700
Film-Noir  4.125932
```

13.3. ИМЕНА, КОТОРЫЕ ДАВАЛИ ДЕТЯМ В США ЗА ПЕРИОД с 1880 по 2010 год

Управление социального обеспечения США выложило в сеть данные о частоте встречаемости детских имен за период с 1880 года по настоящее время. Хэдли Уикхэм (Hadley Wickham), автор нескольких популярных пакетов для R, часто использует этот пример для иллюстрации манипуляций с данными в R.

Чтобы загрузить этот набор, данные придется немного переформатировать, получившийся в результате объект `DataFrame` выглядит так:

```
In [4]: names.head(10)
Out[4]:
```

	name	sex	births	year
0	Mary	F	7065	1880
1	Anna	F	2604	1880
2	Emma	F	2003	1880
3	Elizabeth	F	1939	1880
4	Minnie	F	1746	1880
5	Margaret	F	1578	1880
6	Ida	F	1472	1880
7	Alice	F	1414	1880
8	Bertha	F	1320	1880
9	Sarah	F	1288	1880

С этим набором можно проделать много интересного.

- Наглядно представить долю младенцев, получавших данное имя (совпадающее с вашим или какое-нибудь другое) за весь период времени.
- Определить относительный ранг имени.
- Найти самые популярные в каждом году имена или имена, для которых фиксировалось наибольшее увеличение или уменьшение частоты.
- Проанализировать тенденции выбора имен: количество гласных и согласных, длину, общее разнообразие, изменение в написании, первые и последние буквы.
- Проанализировать внешние источники тенденций: библейские имена, имена знаменитостей, демографические изменения.

С помощью рассмотренных в этой книге инструментов большая часть этих задач решается без особого труда, и я это кратко продемонстрирую.

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 со временем адрес этой страницы поменяется, найти ее, скорее всего, можно будет с помощью поисковой системы. Загрузив и распаковав файл `names.zip`, вы получите каталог, содержащий файлы с именами вида `yob1880.txt`. С помощью команды UNIX `head` я могу вывести первые 10 строк каждого файла (в Windows можно воспользоваться командой `more` или открыть файл в текстовом редакторе):

```
In [106]: !head -n 10 datasets/babynames/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 [107]: names1880 = pd.read_csv("datasets/babynames/yob1880.txt",
.....:                             names=["name", "sex", "births"])

In [108]: names1880
Out[108]:
```

	name	sex	births
0	Mary	F	7065
1	Anna	F	2604
2	Emma	F	2003
3	Elizabeth	F	1939
4	Minnie	F	1746
...	...	..	...
1995	Woodie	M	5
1996	Worthy	M	5
1997	Wright	M	5
1998	York	M	5
1999	Zachariah	M	5

```
[2000 rows x 3 columns]
```

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

```
In [109]: names1880.groupby("sex")["births"].sum()
Out[109]:
sex
F      90993
M     110493
Name: births, dtype: int64
```

Поскольку в каждом файле находятся данные только за один год, то первое, что нужно сделать, – собрать все данные в единый объект `DataFrame` и добавить поле `year`. Это легко сделать методом `pandas.concat`. Выполните следующий код в ячейке Jupyter:

```
pieces = []
for year in range(1880, 2011):
    path = f"datasets/babynames/yob{year}.txt"
    frame = pd.read_csv(path, names=["name", "sex", "births"])

    # Добавить столбец для года
    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 за все годы.

```
In [111]: names
Out[111]:
```

	name	sex	births	year
0	Mary	F	7065	1880
1	Anna	F	2604	1880
2	Emma	F	2003	1880
3	Elizabeth	F	1939	1880
4	Minnie	F	1746	1880
...	...	..	...	...
1690779	Zymaire	M	5	2010
1690780	Zyonne	M	5	2010
1690781	Zyquarius	M	5	2010
1690782	Zyran	M	5	2010
1690783	Zzyzx	M	5	2010

```
[1690784 rows x 4 columns]
```

Имея эти данные, мы уже можем приступить к агрегированию на уровне года и пола, используя метод `groupby` или `pivot_table` (см. рис. 13.4):

```
In [112]: total_births = names.pivot_table("births", index="year",
.....:                                     columns="sex", aggfunc=sum)

In [113]: total_births.tail()
Out[113]:
```

year	sex	
	F	M
2006	1896468	2050234
2007	1916888	2069242
2008	1883645	2032310
2009	1827643	1973359
2010	1759010	1898382

```
In [114]: total_births.plot(title="Total births by sex and year")
```

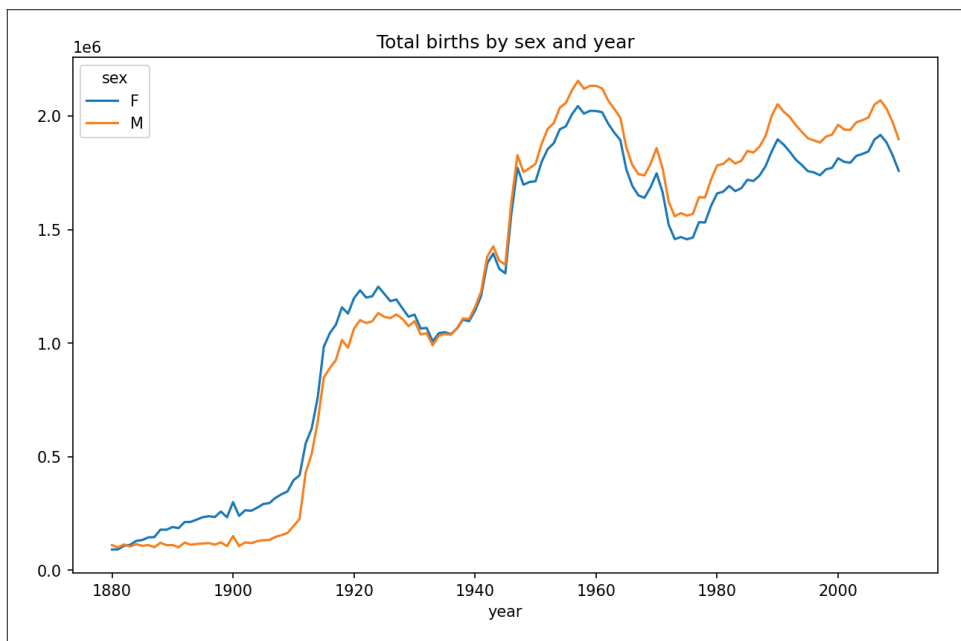


Рис. 13.4. Общее количество родившихся по полу и году

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):
    group["prop"] = group["births"] / group["births"].sum()
    return group
names = names.groupby(["year", "sex"]).apply(add_prop)
```

Получившийся в результате пополненный набор данных состоит из таких столбцов:

```
In [116]: names
Out[116]:
```

	name	sex	births	year	prop
0	Mary	F	7065	1880	0.077643
1	Anna	F	2604	1880	0.028618
2	Emma	F	2003	1880	0.022013
3	Elizabeth	F	1939	1880	0.021309
4	Minnie	F	1746	1880	0.019188
...	...	..	...	...	...
1690779	Zymaire	M	5	2010	0.000003
1690780	Zyonne	M	5	2010	0.000003
1690781	Zyquarius	M	5	2010	0.000003
1690782	Zyran	M	5	2010	0.000003
1690783	Zzyzx	M	5	2010	0.000003

[1690784 rows x 5 columns]

При выполнении такой операции группировки часто бывает полезно произвести проверку разумности результата, например удостовериться, что сумма значений в столбце `prop` по всем группам равна 1.

```
In [117]: names.groupby(["year", "sex"])["prop"].sum()
Out[117]:
year sex
1880  F    1.0
      M    1.0
1881  F    1.0
      M    1.0
1882  F    1.0
      ...
2008  M    1.0
2009  F    1.0
      M    1.0
2010  F    1.0
      M    1.0
Name: prop, Length: 262, dtype: float64
```

Далее я извлеку подмножество данных, чтобы упростить последующий анализ: первые 1000 имен для каждой комбинации пола и года. Это еще одна групповая операция:

```
In [118]: def get_top1000(group):
.....:     return group.sort_values("births", ascending=False)[:1000]

In [119]: grouped = names.groupby(["year", "sex"])

In [120]: top1000 = grouped.apply(get_top1000)

In [121]: top1000.head()
Out[121]:
```

year	sex	name	sex	births	year	prop	
1880	F	0	Mary	F	7065	1880	0.077643
		1	Anna	F	2604	1880	0.028618
		2	Emma	F	2003	1880	0.022013
		3	Elizabeth	F	1939	1880	0.021309
		4	Minnie	F	1746	1880	0.019188

Групповой индекс мы можем удалить, т. к. он больше не понадобится для анализа:

```
In [122]: top1000 = top1000.reset_index(drop=True)
Теперь результирующий набор стал заметно меньше:
In [123]: top1000.head()
Out[123]:
```

	name	sex	births	year	prop
0	Mary	F	7065	1880	0.077643
1	Anna	F	2604	1880	0.028618
2	Emma	F	2003	1880	0.022013
3	Elizabeth	F	1939	1880	0.021309
4	Minnie	F	1746	1880	0.019188

Это набор, содержащий первые 1000 записей, мы и будем использовать его 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 разобьем набор Top 1000 на части, относящиеся к мальчикам и девочкам.

```
In [124]: boys = top1000[top1000["sex"] == "M"]
```

```
In [125]: 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 преформатирование. Сформируем сводную таблицу, в которой представлено общее число родившихся по годам и по именам:

```
In [126]: total_births = top1000.pivot_table("births", index="year",
.....:                                     columns="name",
.....:                                     aggfunc=sum)
```

Теперь можно нанести на график несколько имен, воспользовавшись методом `plot` объекта `DataFrame` (результат показан на рис. 13.5):

```
In [127]: total_births.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB
```

```
In [128]: subset = total_births[["John", "Harry", "Mary", "Marilyn"]]
```

```
In [129]: subset.plot(subplots=True, figsize=(12, 10),
.....:               title="Number of births per year")
```

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

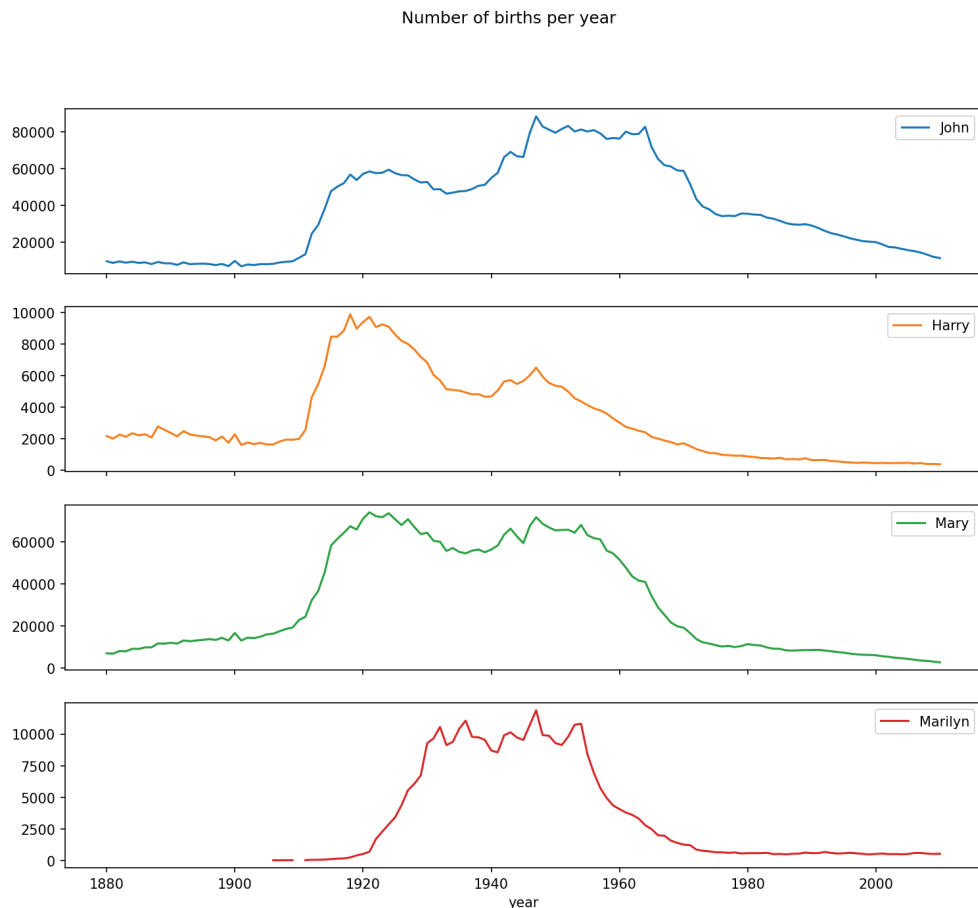


Рис. 13.5. Распределение нескольких имен мальчиков и девочек по годам

Измерение роста разнообразия имен

Убывание кривых на рисунках выше можно объяснить тем, что меньше родителей стали выбирать для своих детей распространенные имена. Эту гипотезу можно проверить и подтвердить имеющимися данными. Один из возможных показателей – доля родившихся в наборе 1000 самых популярных имен, который я агрегирую по году и полу (результат показан на рис. 13.6):

```
In [131]: table = top1000.pivot_table("prop", index="year",
.....:                                columns="sex", aggfunc=sum)

In [132]: table.plot(title="Sum of table1000.prop by year and sex",
.....:               yticks=np.linspace(0, 1.2, 13))
```

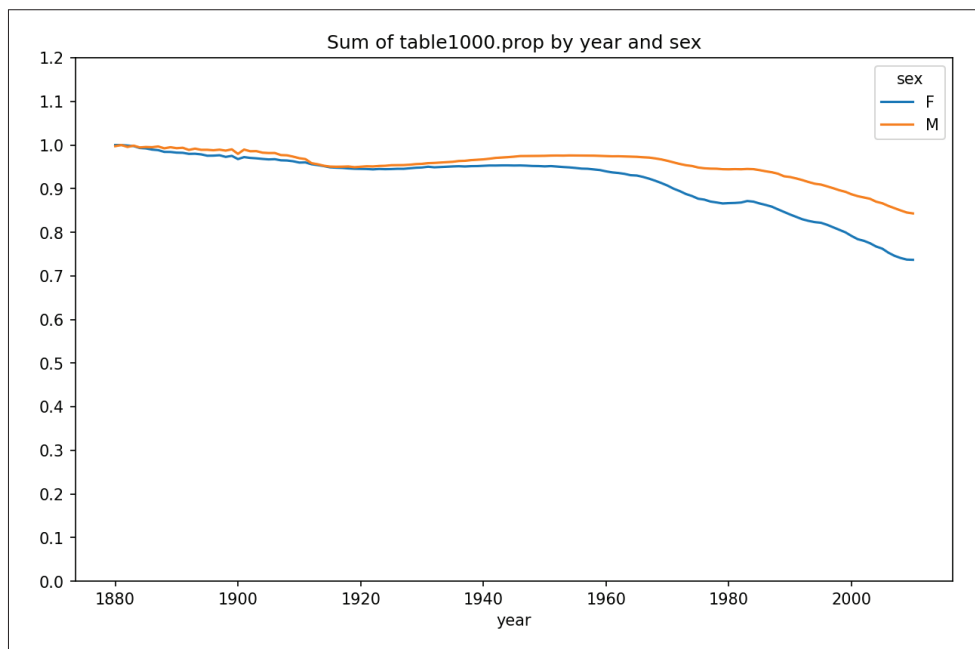


Рис. 13.6. Доля родившихся мальчиков и девочек, представленных в первой тысяче имен

Действительно, похоже, что разнообразие имен растёт (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 [133]: df = boys[boys["year"] == 2010]

In [134]: df
Out[134]:
```

	name	sex	births	year	prop
260877	Jacob	M	21875	2010	0.011523
260878	Ethan	M	17866	2010	0.009411
260879	Michael	M	17133	2010	0.009025
260880	Jayden	M	17030	2010	0.008971
260881	William	M	16870	2010	0.008887
...	...	..	...	...	...
261872	Camilo	M	194	2010	0.000102
261873	Destin	M	194	2010	0.000102
261874	Jaquan	M	194	2010	0.000102
261875	Jaydan	M	194	2010	0.000102
261876	Maxton	M	193	2010	0.000102

[1000 rows x 5 columns]

После сортировки `prop` в порядке убывания мы хотим узнать, сколько популярных имен нужно взять, чтобы достичь 50 %. Можно написать для этого цикл `for`, но NumPy предлагает более хитроумный векторный подход. Если вычислить накопительные суммы `cumsum` массива `prop`, а затем вызвать метод

`searchsorted`, то будет возвращена позиция в массиве накопительных сумм, в которую нужно было бы вставить `0.5`, чтобы не нарушить порядок сортировки:

```
In [135]: prop_cumsum = df["prop"].sort_values(ascending=False).cumsum()
```

```
In [136]: prop_cumsum[:10]
```

```
Out[136]:
```

```
260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621
```

```
Name: prop, dtype: float64
```

```
In [137]: prop_cumsum.searchsorted(0.5)
```

```
Out[137]: 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 [138]: df = boys[boys.year == 1900]
```

```
In [139]: in1900 = df.sort_values("prop", ascending=False).prop.cumsum()
```

```
In [140]: in1900.searchsorted(0.5) + 1
```

```
Out[140]: 25
```

Теперь можно применить эту операцию к каждой комбинации года и пола, произвести группировку по этим полям с помощью метода `groupby`, а затем с помощью метода `apply` применить функцию, возвращающую счетчик для каждой группы:

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

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

```
diversity = diversity.unstack()
```

В получившемся объекте `DataFrame` с именем `diversity` хранятся два временных ряда, по одному для каждого пола, индексированные по году. Его можно исследовать и, как и раньше, нанести на график (рис. 13.7).

```
In [143]: diversity.head()
```

```
Out[143]:
```

```
sex      F      M
year
1880    38    14
1881    38    14
1882    38    15
1883    39    15
```

1884 39 16

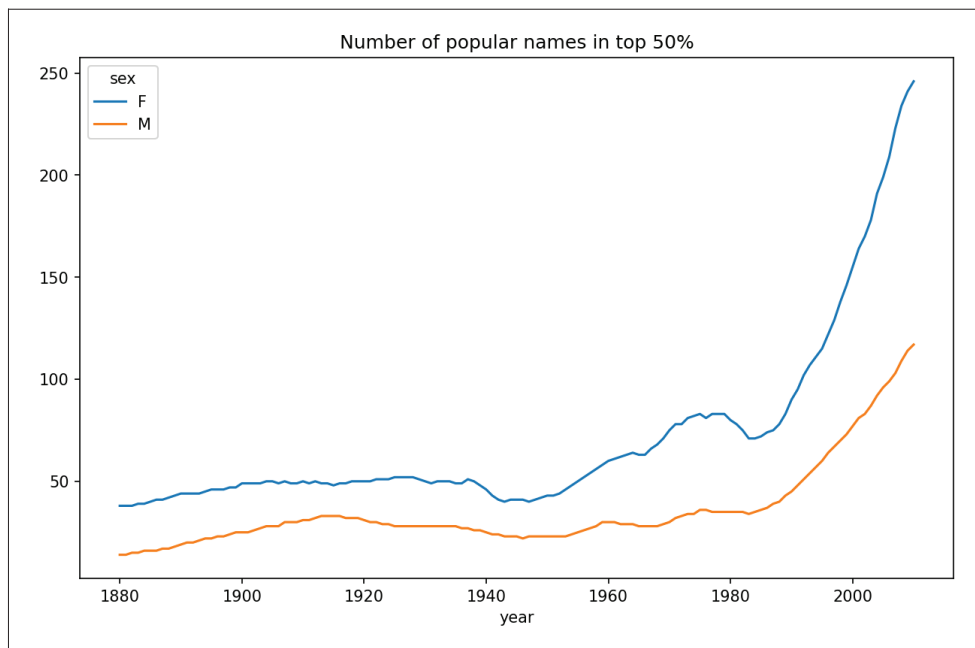
In [144]: `diversity.plot(title="Number of popular names in top 50%")`

Рис. 13.7. График зависимости разнообразия от года

Как видим, девочкам всегда давали более разнообразные имена, чем мальчикам, и со временем эта тенденция проявляется все ярче. Анализ того, что именно является причиной разнообразия, например 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:

```
def get_last_letter(x):
    return x[-1]

last_letters = names["name"].map(get_last_letter)
last_letters.name = "last_letter"

table = names.pivot_table("births", index=last_letters,
                           columns=["sex", "year"], aggfunc=sum)
```

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

```
In [146]: subtable = table.reindex(columns=[1910, 1960, 2010], level="year")
```

```
In [147]: subtable.head()
```

```
Out[147]:
```

sex	F			M		
	1910	1960	2010	1910	1960	2010
last_letter						
a	108376.0	691247.0	670605.0	977.0	5204.0	28438.0
b	NaN	694.0	450.0	411.0	3912.0	38859.0
c	5.0	49.0	946.0	482.0	15476.0	23125.0
d	6750.0	3729.0	2607.0	22111.0	262112.0	44398.0
e	133569.0	435013.0	313833.0	28655.0	178823.0	129012.0

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

```
In [148]: subtable.sum()
```

```
Out[148]:
```

```
sex year
F      1910      396416.0
      1960      2022062.0
      2010      1759010.0
M      1910      194198.0
      1960      2132588.0
      2010      1898382.0
```

```
dtype: float64
```

```
In [149]: letter_prop = subtable / subtable.sum()
```

```
In [150]: letter_prop
```

```
Out[150]:
```

sex	F			M		
	1910	1960	2010	1910	1960	2010
last_letter						
a	0.273390	0.341853	0.381240	0.005031	0.002440	0.014980
b	NaN	0.000343	0.000256	0.002116	0.001834	0.020470
c	0.000013	0.000024	0.000538	0.002482	0.007257	0.012181
d	0.017028	0.001844	0.001482	0.113858	0.122908	0.023387
e	0.336941	0.215133	0.178415	0.147556	0.083853	0.067959
...	...	...	...	...	...	...
v	NaN	0.000060	0.000117	0.000113	0.000037	0.001434
w	0.000020	0.000031	0.001182	0.006329	0.007711	0.016148
x	0.000015	0.000037	0.000727	0.003965	0.001851	0.008614
y	0.110972	0.152569	0.116828	0.077349	0.160987	0.058168
z	0.002439	0.000659	0.000704	0.000170	0.000184	0.001831

```
[26 rows x 6 columns]
```

Зная доли букв, я теперь могу нарисовать столбчатые диаграммы для каждого пола, разбив их по годам (см. рис. 13.8).

```
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)
```

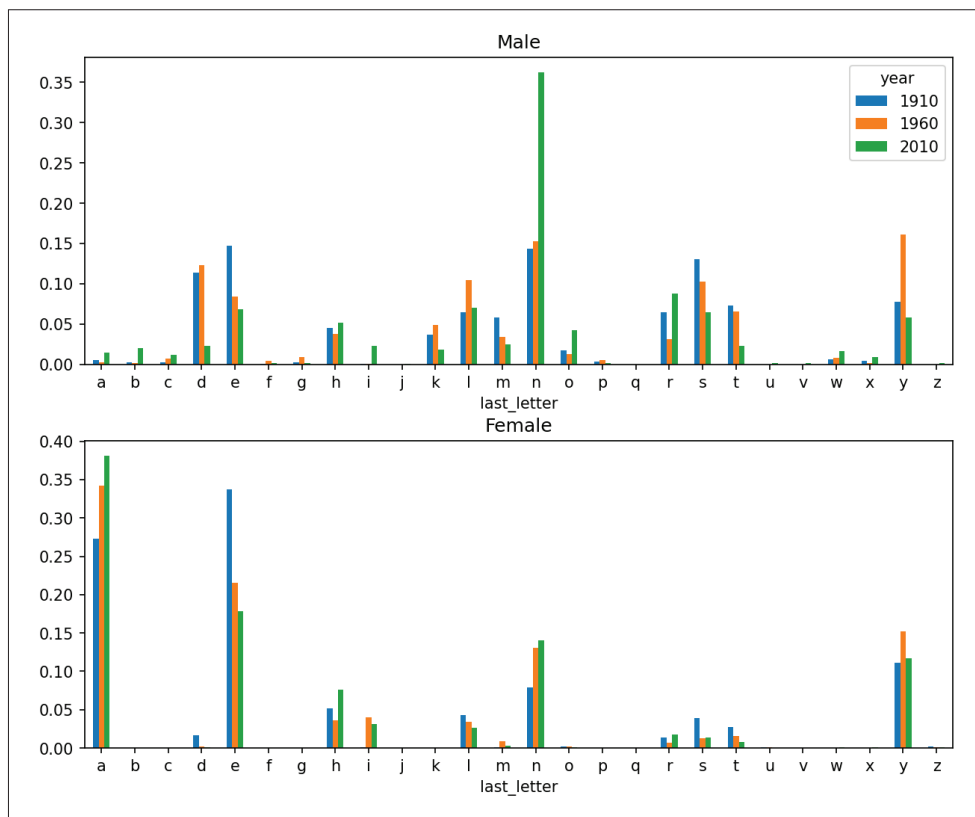


Рис. 13.8. Доли имен мальчиков и девочек, заканчивающихся на каждую букву

Как видим, с 1960-х годов доля имен мальчиков, заканчивающихся буквой *n*, значительно возросла. Снова вернусь к созданной ранее полной таблице, нормирую ее по году и полу, выберу некое подмножество букв для имен мальчиков и транспонирую, чтобы превратить каждый столбец во временной ряд:

```
In [153]: letter_prop = table / table.sum()

In [154]: dny_ts = letter_prop.loc[["d", "n", "y"], "M"].T

In [155]: dny_ts.head()
Out[155]:
last_letter      d      n      y
year
1880      0.083055  0.153213  0.075760
1881      0.083247  0.153214  0.077451
1882      0.085340  0.149560  0.077537
1883      0.084066  0.151646  0.079144
1884      0.086120  0.149915  0.080405
```

Имея этот объект `DataFrame`, содержащий временные ряды, я могу все тем же методом `plot` построить график изменения тенденций в зависимости от времени (рис. 13.9):


```
In [158]: dny_ts.plot()
```

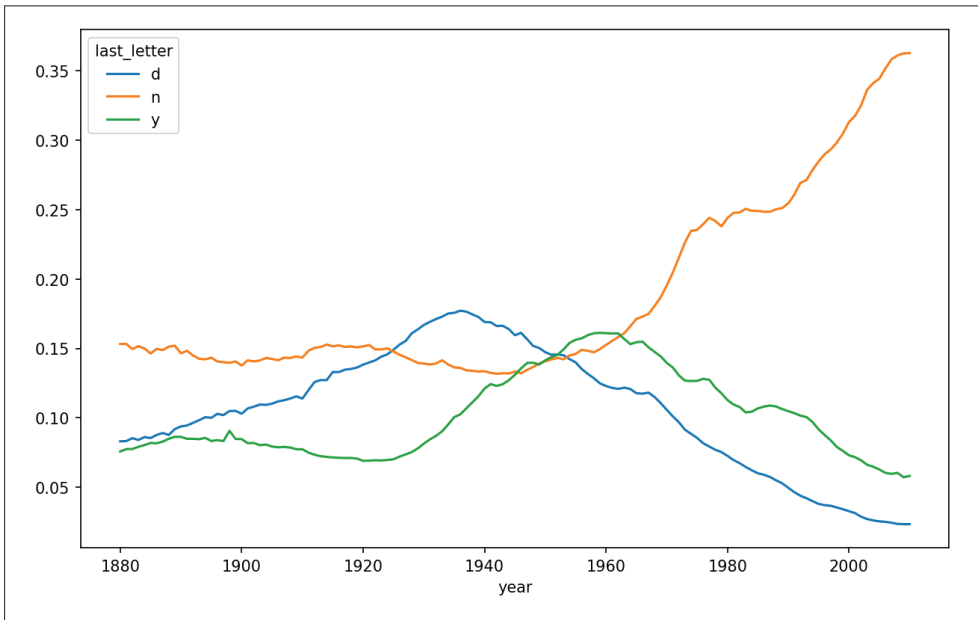


Рис. 13.9. Зависимость доли мальчиков с именами, заканчивающимися на буквы d, n, y, от времени

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 [159]: all_names = pd.Series(top1000["name"].unique())

In [160]: lesley_like = all_names[all_names.str.contains("Lesl")]

In [161]: lesley_like
Out[161]:
632    Leslie
2294   Lesley
4262   Leslee
4728   Lesli
6103   Lesly
dtype: object
```

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

```
In [162]: filtered = top1000[top1000["name"].isin(lesley_like)]

In [163]: filtered.groupby("name")["births"].sum()
Out[163]:
name
Leslee    1082
```

```

Lesley    35022
Lesli     929
Leslie    370429
Lesly     10067
Name: births, dtype: int64

```

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

```

In [164]: table = filtered.pivot_table("births", index="year",
.....:                                columns="sex", aggfunc="sum")

```

```

In [165]: table = table.div(table.sum(axis="columns"), axis="index")

```

```

In [166]: table.tail()

```

```

Out[166]:
sex      F  M
year
2006    1.0 NaN
2007    1.0 NaN
2008    1.0 NaN
2009    1.0 NaN
2010    1.0 NaN

```

Наконец, нетрудно построить график распределения по полу в зависимости от времени (рис. 13.10).

```

In [168]: table.plot(style={"M": "k-", "F": "k--"})

```

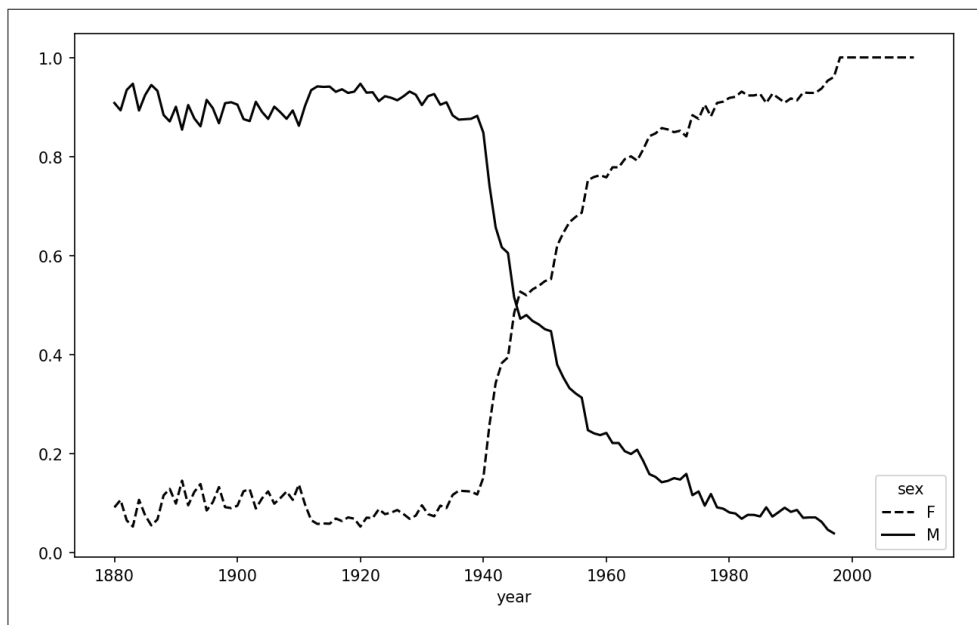


Рис. 13.10. Изменение во времени доли мальчиков и девочек с именами, похожими на Lesley

13.4. БАЗА ДАННЫХ О ПРОДУКТАХ ПИТАНИЯ МИНИСТЕРСТВА СЕЛЬСКОГО ХОЗЯЙСТВА США

Министерство сельского хозяйства США публикует данные о пищевой ценности продуктов питания. Программист Эшли Уильямс (Ashley Williams) преобразовал эту базу данных в формат JSON. Записи выглядят следующим образом:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },
    ...
  ],
  "nutrients": [
    {
      "value": 20.8,
      "units": "g",
      "description": "Protein",
      "group": "Composition"
    },
    ...
  ]
}
```

У каждого продукта питания есть ряд идентифицирующих атрибутов и два списка: питательные элементы и размеры порций. Для анализа данные в такой форме подходят плохо, поэтому необходимо их переформатировать.

Загрузить этот файл в Python-программу можно с помощью любой библиотеки для работы с JSON. Я воспользуюсь стандартным модулем Python `json`:

```
In [169]: import json

In [170]: db = json.load(open("datasets/usda_food/database.json"))

In [171]: len(db)
Out[171]: 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 [172]: db[0].keys()
Out[172]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'group', 'portions',
'nutrients'])

In [173]: db[0]["nutrients"][0]
```

```
Out[173]:
{'value': 25.18,
 'units': 'g',
 'description': 'Protein',
 'group': 'Composition'}
```

```
In [174]: nutrients = pd.DataFrame(db[0]["nutrients"])
```

```
In [175]: nutrients.head(7)
```

```
Out[175]:
```

	value	units	description	group
0	25.18	g	Protein	Composition
1	29.20	g	Total lipid (fat)	Composition
2	3.06	g	Carbohydrate, by difference	Composition
3	3.28	g	Ash	Other
4	376.00	kcal	Energy	Energy
5	39.28	g	Water	Composition
6	1573.00	kJ	Energy	Energy

Преобразуя список словарей в DataFrame, можно задать список полей, подлежащих извлечению. Мы ограничимся названием продукта, группой, идентификатором и производителем:

```
In [176]: info_keys = ["description", "group", "id", "manufacturer"]
```

```
In [177]: info = pd.DataFrame(db, columns=info_keys)
```

```
In [178]: info.head()
```

```
Out[178]:
```

	description	group	id \
0	Cheese, caraway	Dairy and Egg Products	1008
1	Cheese, cheddar	Dairy and Egg Products	1009
2	Cheese, edam	Dairy and Egg Products	1018
3	Cheese, feta	Dairy and Egg Products	1019
4	Cheese, mozzarella, part skim milk	Dairy and Egg Products	1028

```
manufacturer
0
1
2
3
4
```

```
In [179]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
#   Column          Non-Null Count  Dtype
---  ---
0   description      6636 non-null   object
1   group            6636 non-null   object
2   id               6636 non-null   int64
3   manufacturer     5195 non-null   object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

Из результата `info.info()` видно, что в столбце `manufacturer` есть отсутствующие данные.

Метод `value_counts` покажет распределение продуктов питания по группам:

```
In [180]: pd.value_counts(info["group"])[0:10]
Out[180]:
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
Fruits and Fruit Juices               328
Pork Products                        328
Name: group, dtype: int64
```

Чтобы теперь произвести анализ данных о питательных элементах, проще всего собрать все питательные элементы для всех продуктов в одну большую таблицу. Для этого понадобится несколько шагов. Сначала я преобразую каждый список питательных элементов в объект `DataFrame`, добавлю столбец `id`, содержащий идентификатор продукта, и помещу этот `DataFrame` в список. После этого все объекты можно будет конкатенировать методом `concat`. Выполните следующий код в ячейке Jupyter:

```
nutrients = []
for rec in db:
    fnuts = pd.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 [182]: nutrients
Out[182]:
```

	value	units		description	group	id
0	25.180	g		Protein	Composition	1008
1	29.200	g		Total lipid (fat)	Composition	1008
2	3.060	g		Carbohydrate, by difference	Composition	1008
3	3.280	g		Ash	Other	1008
4	376.000	kcal		Energy	Energy	1008
...	...	...		...	...	...
389350	0.000	mcg		Vitamin B-12, added	Vitamins	43546
389351	0.000	mg		Cholesterol	Other	43546
389352	0.072	g		Fatty acids, total saturated	Other	43546
389353	0.028	g		Fatty acids, total monounsaturated	Other	43546
389354	0.041	g		Fatty acids, total polyunsaturated	Other	43546

```
[389355 rows x 5 columns]
```

Я заметил, что в этом `DataFrame` есть дубликаты, поэтому лучше их удалить:

```
In [183]: nutrients.duplicated().sum() # количество дубликатов
Out[183]: 14179

In [184]: nutrients = nutrients.drop_duplicates()
```

Поскольку в обоих объектах DataFrame имеются столбцы 'group' и 'description', переименуем их, чтобы было понятно, что есть что:

```
In [185]: col_mapping = {"description" : "food",
.....:                  "group" : "fgroup"}
```

```
In [186]: info = info.rename(columns=col_mapping, copy=False)
```

```
In [187]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
#   Column                Non-Null Count  Dtype
---  ---
0   food                  6636 non-null   object
1   fgroup                6636 non-null   object
2   id                    6636 non-null   int64
3   manufacturer          5195 non-null   object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

```
In [188]: col_mapping = {"description" : "nutrient",
.....:                  "group" : "nutgroup"}
```

```
In [189]: nutrients = nutrients.rename(columns=col_mapping, copy=False)
```

```
In [190]: nutrients
```

```
Out[190]:
```

	value	units		nutrient	nutgroup	id
0	25.180	g		Protein	Composition	1008
1	29.200	g		Total lipid (fat)	Composition	1008
2	3.060	g		Carbohydrate, by difference	Composition	1008
3	3.280	g		Ash	Other	1008
4	376.000	kcal		Energy	Energy	1008
...	...	...		...	...	...
389350	0.000	mcg		Vitamin B-12, added	Vitamins	43546
389351	0.000	mg		Cholesterol	Other	43546
389352	0.072	g		Fatty acids, total saturated	Other	43546
389353	0.028	g		Fatty acids, total monounsaturated	Other	43546
389354	0.041	g		Fatty acids, total polyunsaturated	Other	43546

[375176 rows x 5 columns]

Сделав все это, мы можем объединить info с nutrients:

```
In [191]: ndata = pd.merge(nutrients, info, on="id")
```

```
In [192]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
#   Column                Non-Null Count  Dtype
---  ---
0   value                 375176 non-null  float64
1   units                 375176 non-null  object
2   nutrient              375176 non-null  object
3   nutgroup              375176 non-null  object
4   id                    375176 non-null  int64
5   food                  375176 non-null  object
```

```
6   fgroup      375176 non-null object
7   manufacturer 293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB
```

```
In [193]: ndata.iloc[30000]
Out[193]:
value      0.04
units      g
nutrient    Glycine
nutgroup    Amino Acids
id          6158
food        Soup, tomato bisque, canned, condensed
fgroup      Soups, Sauces, and Gravies
manufacturer
Name: 30000, dtype: object
```

Теперь можно построить график медианных значений по группе и типу питательного элемента (рис. 13.11):

```
In [195]: result = ndata.groupby(["nutrient", "fgroup"])["value"].quantile(0.5)

In [196]: result["Zinc, Zn"].sort_values().plot(kind="barh")
```

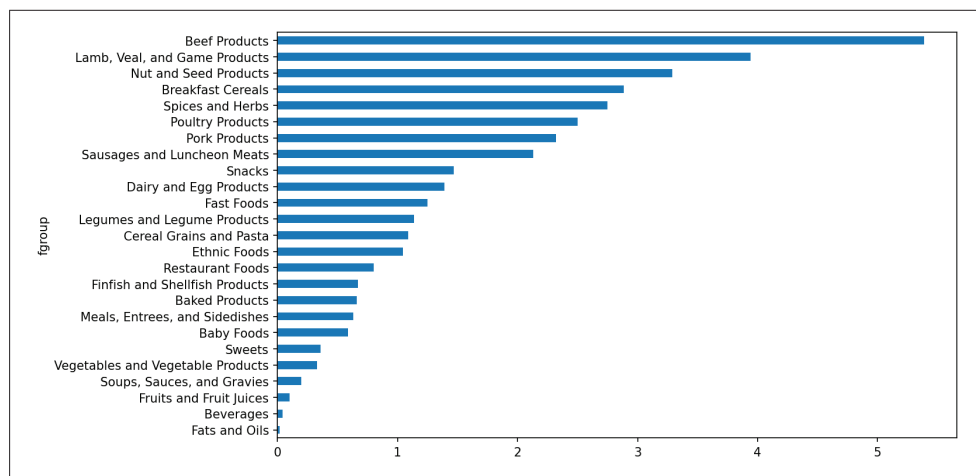


Рис. 13.11. Медианные значения цинка по группе питательных элементов

Воспользовавшись методом `idxmax` или `argmax` объекта `Series`, мы сможем найти, какой продукт питания наиболее богат каждым питательным элементом. Выполните следующий код в ячейке Jupyter:

```
by_nutrient = ndata.groupby(["nutgroup", "nutrient"])

def get_maximum(x):
    return x.loc[x.value.idxmax()]

max_foods = by_nutrient.apply(get_maximum)[["value", "food"]]

# Немного уменьшить длину названия продукта питания
max_foods["food"] = max_foods["food"].str[:50]
```

Получившийся объект DataFrame слишком велик, для того чтобы приводить его полностью. Ниже приведена только группа питательных элементов 'Amino Acids' (аминокислоты):

```
In [198]: max_foods.loc["Amino Acids"]["food"]
Out[198]:
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 RE
Isoleucine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Leucine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Lysine Seal, bearded (Oogruk), meat, dried (Alaska Native)
Methionine Fish, cod, Atlantic, dried and salted
Phenylalanine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Proline Gelatins, dry powder, unsweetened
Serine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Threonine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Tryptophan Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Valine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Name: food, dtype: object
```

13.5. БАЗА ДАННЫХ ФЕДЕРАЛЬНОЙ ИЗБИРАТЕЛЬНОЙ КОМИССИИ

Федеральная избирательная комиссия США публикует данные о пожертвованиях участникам политических кампаний. Указывается имя жертвователя, род занятий, место работы и сумма пожертвования. Интерес представляет набор данных, относящийся к президентским выборам 2012 года. Версия этого набора, загруженная мной в июне 2012 года, представляет собой CSV-файл *P00000001-ALL.csv* размером 150 МБ, который можно скачать с помощью функции `pandas.read_csv`:

```
In [199]: fec = pd.read_csv("datasets/fec/P00000001-ALL.csv", low_memory=False)
```

```
In [200]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
#   Column                Non-Null Count  Dtype
---  -
0   cmte_id                1001731 non-null object
1   cand_id                1001731 non-null object
2   cand_nm                1001731 non-null object
3   contbr_nm              1001731 non-null object
4   contbr_city            1001712 non-null object
5   contbr_st              1001727 non-null object
6   contbr_zip             1001620 non-null object
7   contbr_employer        988002 non-null object
8   contbr_occupation      993301 non-null object
```



```

9  contb_receipt_amt  1001731 non-null float64
10 contb_receipt_dt   1001731 non-null object
11 receipt_desc       14166 non-null object
12 memo_cd            92482 non-null object
13 memo_text          97770 non-null object
14 form_tp            1001731 non-null object
15 file_num           1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB

```



Несколько человек просили меня заменить набор данных о выборах 2012 года набором, относящимся к 2016 или 2020 году. К сожалению, более поздние наборы данных, предоставляемые Федеральной избирательной комиссией, стали больше и сложнее, и я решил, что детали работы с ними будут только отвлекать от техники анализа, которую я хотел проиллюстрировать.

Ниже приведен пример записи в объекте DataFrame:

```

In [201]: fec.iloc[123456]
Out[201]:
cmtc_id C00431445
cand_id P80003338
cand_nm Obama, Barack
contbr_nm ELLMAN, IRA
contbr_city TEMPE
contbr_st AZ
contbr_zip 852816719
contbr_employer ARIZONA STATE UNIVERSITY
contbr_occupation PROFESSOR
contb_receipt_amt 50.0
contb_receipt_dt 01-DEC-11
receipt_desc NaN
memo_cd NaN
memo_text NaN
form_tp SA17A
file_num 772372
Name: 123456, dtype: object

```

Наверное, вы сходу сможете придумать множество способов манипуляции этими данными для извлечения полезной статистики о спонсорах и закономерностях жертвования. Далее я покажу различные виды анализа, чтобы проиллюстрировать рассмотренные в книге технические приемы.

Как видите, данные не содержат сведений о принадлежности кандидата к политической партии, а эту информацию было бы полезно добавить. Получить список различных кандидатов можно с помощью функции `unique`:

```

In [202]: unique_cands = fec["cand_nm"].unique()

In [203]: unique_cands
Out[203]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
      'Roemer, Charles E. 'Buddy' III', 'Pawlenty, Timothy',
      'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick',
      'Cain, Herman', 'Gingrich, Newt', 'McCotter, Thaddeus G',
      'Huntsman, Jon', 'Perry, Rick'], dtype=object)

```

```
In [204]: unique_cands[2]
Out[204]: 'Obama, Barack'
Указать партийную принадлежность проще всего с помощью словаря11:
parties = {"Bachmann, Michelle": "Republican",
          "Cain, Herman": "Republican",
          "Gingrich, Newt": "Republican",
          "Huntsman, Jon": "Republican",
          "Johnson, Gary Earl": "Republican",
          "McCotter, Thaddeus G": "Republican",
          "Obama, Barack": "Democrat",
          "Paul, Ron": "Republican",
          "Pawlenty, Timothy": "Republican",
          "Perry, Rick": "Republican",
          "Roemer, Charles E. 'Buddy' III": "Republican",
          "Romney, Mitt": "Republican",
          "Santorum, Rick": "Republican"}
```

Далее, применяя этот словарь и метод `map` объектов `Series`, мы можем построить массив политических партий по именам кандидатов:

```
In [206]: fec["cand_nm"][123456:123461]
Out[206]:
123456 Obama, Barack
123457 Obama, Barack
123458 Obama, Barack
123459 Obama, Barack
123460 Obama, Barack
Name: cand_nm, dtype: object

In [207]: fec["cand_nm"][123456:123461].map(parties)
Out[207]:
123456 Democrat
123457 Democrat
123458 Democrat
123459 Democrat
123460 Democrat
Name: cand_nm, dtype: object

# Добавить в виде столбца
In [208]: fec["party"] = fec["cand_nm"].map(parties)

In [209]: fec["party"].value_counts()
Out[209]:
Democrat 593746
Republican 407985
Name: party, dtype: int64
```

Теперь два замечания касательно подготовки данных. Во-первых, данные включают как пожертвования, так и возвраты (пожертвования со знаком минус):

```
In [210]: (fec["contb_receipt_amt"] > 0).value_counts()
Out[210]:
True      991475
False     10256
Name: contb_receipt_amt, dtype: int64
```

¹¹ Здесь сделано упрощающее предположение о том, что Гэри Джонсон – республиканец, хотя впоследствии он стал кандидатом от Либертарианской партии.

Чтобы упростить анализ, я ограничусь только положительными суммами пожертвований:

```
In [211]: fec = fec[fec["contb_receipt_amt"] > 0]
```

Поскольку главными кандидатами были Барак Обама и Митт Ромни, я подготовлю также подмножество, содержащее данные о пожертвованиях на их кампании:

```
In [212]: fec_mrbo = fec[fec["cand_nm"].isin(["Obama, Barack", "Romney, Mitt"])]
```

Статистика пожертвований по роду занятий и месту работы

Распределение пожертвований по роду занятий – тема, которой посвящено много исследований. Например, юристы обычно жертвуют в пользу демократов, а руководители предприятий – в пользу республиканцев. Вы вовсе не обязаны верить мне на слово, можете сами проанализировать данные. Для начала решим простую задачу – получим общую статистику пожертвований по роду занятий:

```
In [213]: fec["contbr_occupation"].value_counts()[:10]
Out[213]:
RETIRED 233990
INFORMATION REQUESTED                35107
ATTORNEY 34286
HOMEMAKER 29931
PHYSICIAN 23432
INFORMATION REQUESTED PER BEST EFFORTS 21138
ENGINEER 14334
TEACHER 13990
CONSULTANT 13273
PROFESSOR 12555
Name: contbr_occupation, dtype: int64
```

Видно, что часто различные занятия на самом деле относятся к одной и той же основной профессии, с небольшими вариациями. Ниже показан код, который позволяет произвести очистку, отобразив один род занятий на другой. Обратите внимание на «трюк» с методом `dict.get`, который позволяет «передавать насквозь» занятия, которым ничего не сопоставлено:

```
occ_mapping = {
    "INFORMATION REQUESTED PER BEST EFFORTS" : "NOT PROVIDED",
    "INFORMATION REQUESTED" : "NOT PROVIDED",
    "INFORMATION REQUESTED (BEST EFFORTS)" : "NOT PROVIDED",
    "C.E.O." : "CEO"
}

def get_occ(x):
    # Если ничего не сопоставлено, вернуть x
    return occ_mapping.get(x, x)

fec["contbr_occupation"] = fec["contbr_occupation"].map(get_occ)
```

То же самое я сделаю для места работы:

```

emp_mapping = {
    "INFORMATION REQUESTED PER BEST EFFORTS" : "NOT PROVIDED",
    "INFORMATION REQUESTED" : "NOT PROVIDED",
    "SELF" : "SELF-EMPLOYED",
    "SELF EMPLOYED" : "SELF-EMPLOYED",
}

def get_emp(x):
    # Если ничего не сопоставлено, вернуть x
    return emp_mapping.get(x, x)
fec["contbr_employer"] = fec["contbr_employer"].map(f)

```

Теперь можно воспользоваться функцией `pivot_table` для агрегирования данных по партиям и роду занятий, а затем отфильтровать роды занятий, на которые пришлось пожертвований на общую сумму не менее 2 млн долларов:

```

In [216]: by_occupation = fec.pivot_table("contb_receipt_amt",
.....:                                   index="contbr_occupation",
.....:                                   columns="party", aggfunc="sum")

In [217]: over_2mm = by_occupation[by_occupation.sum(axis="columns") > 2000000]

In [218]: over_2mm
Out[218]:
party          Democrat Republican
contbr_occupation
ATTORNEY      11141982.97  7477194.43
CEO            2074974.79  4211040.52
CONSULTANT    2459912.71  2544725.45
ENGINEER       951525.55  1818373.70
EXECUTIVE     1355161.05  4138850.09
HOMEMAKER     4248875.80  13634275.78
INVESTOR       884133.00  2431768.92
LAWYER        3160478.87  391224.32
MANAGER        762883.22  1444532.37
NOT PROVIDED   4866973.96  20565473.01
OWNER         1001567.36  2408286.92
PHYSICIAN     3735124.94  3594320.24
PRESIDENT     1878509.95  4720923.76
PROFESSOR     2165071.08  296702.73
REAL ESTATE    528902.09  1625902.25
RETIRED       25305116.38  23561244.49
SELF-EMPLOYED  672393.40  1640252.54

```

Эти данные проще воспринять в виде графика (параметр `'barh'` означает горизонтальную столбчатую диаграмму, см. рис. 13.12):

```

In [220]: over_2mm.plot(kind="barh")

```

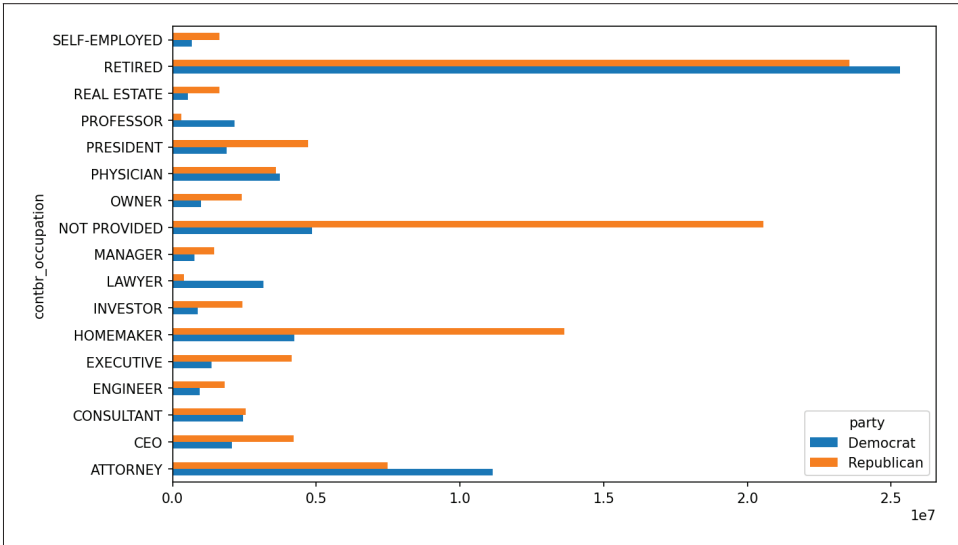


Рис. 13.12. Общая сумма пожертвований по партиям для родов занятий с максимальной суммой пожертвований

Возможно, вам интересны профессии самых щедрых жертвователей или названия компаний, которые больше всех пожертвовали Обаме или Ромни. Для этого можно сгруппировать данные по имени кандидата, а потом воспользоваться вариантом метода `top`, рассмотренного выше в этой главе:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)["contb_receipt_amt"].sum()
    return totals.nlargest(n)
```

Затем агрегируем по роду занятий и месту работы:

```
In [222]: grouped = fec_mrbo.groupby("cand_nm")
```

```
In [223]: grouped.apply(get_top_amounts, "contbr_occupation", n=7)
```

```
Out[223]:
```

cand_nm	contbr_occupation	
Obama, Barack	RETIRED	25305116.38
	ATTORNEY	11141982.97
	INFORMATION REQUESTED	4866973.96
	HOMEMAKER	4248875.80
	PHYSICIAN	3735124.94
	LAWYER	3160478.87
	CONSULTANT	2459912.71
Romney, Mitt	RETIRED	11508473.59
	INFORMATION REQUESTED PER BEST EFFORTS	11396894.84
	HOMEMAKER	8147446.22
	ATTORNEY	5364718.82
	PRESIDENT	2491244.89
	EXECUTIVE	2300947.03
	C.E.O.	1968386.11

```
Name: contb_receipt_amt, dtype: float64
```

```

In [224]: grouped.apply(get_top_amounts, "contbr_employer", n=10)
Out[224]:
cand_nm contbr_employer
Obama, Barack RETIRED 22694358.85
              SELF-EMPLOYED 17080985.96
              NOT EMPLOYED 8586308.70
              INFORMATION REQUESTED 5053480.37
              HOMEMAKER 2605408.54
              SELF 1076531.20
              SELF EMPLOYED 469290.00
              STUDENT 318831.45
              VOLUNTEER 257104.00
              MICROSOFT 215585.36
Romney, Mitt INFORMATION REQUESTED PER BEST EFFORTS
12059527.24
              RETIRED 11506225.71
              HOMEMAKER 8147196.22
              SELF-EMPLOYED 7409860.98
              STUDENT 496490.94
              CREDIT SUISSE 281150.00
              MORGAN STANLEY 267266.00
              GOLDMAN SACH & CO. 238250.00
              BARCLAYS CAPITAL 162750.00
              H.I.G. CAPITAL 139500.00
Name: contb_receipt_amt, dtype: float64

```

Распределение суммы пожертвований по интервалам

Полезный вид анализа данных – дискретизация сумм пожертвований с помощью функции `cut`:

```

In [225]: bins = np.array([0, 1, 10, 100, 1000, 10000,
.....:                    100_000, 1_000_000, 10_000_000])

In [226]: labels = pd.cut(fec_mrbo["contb_receipt_amt"], bins)

In [227]: labels
Out[227]:
411 (10, 100]
412 (100, 1000]
413 (100, 1000]
414 (10, 100]
415 (10, 100]
...
701381 (10, 100]
701382 (100, 1000]
701383 (1, 10]
701384 (10, 100]
701385 (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64, right]): [(0, 1] < (1, 10] < (10, 100] < (100, 1000] <
(1000, 10000] < (10000, 100000] <
(100000, 1000000] < (1000000, 10000000]]

```

Затем можно сгруппировать данные для Обамы и Ромни по имени и метке интервала и построить гистограмму сумм пожертвований:

```
In [228]: grouped = fec_mrbo.groupby(["cand_nm", labels])
```

```
In [229]: grouped.size().unstack(level=0)
```

```
Out[229]:
cand_nm      Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                493                77
(1, 10]             40070             3681
(10, 100]           372280           31853
(100, 1000]         153991           43357
(1000, 10000]       22284            26186
(10000, 100000]        2              1
(100000, 1000000]      3              0
(1000000, 10000000]    4              0
```

Отсюда видно, что Обама получил гораздо больше мелких пожертвований, чем Ромни. Можно также вычислить сумму размеров пожертвований и нормировать распределение по интервалам, чтобы наглядно представить процентную долю пожертвований каждого размера от общего их числа по отдельным кандидатам (рис. 13.13):

```
In [231]: bucket_sums = grouped["contb_receipt_amt"].sum().unstack(level=0)
```

```
In [232]: normed_sums = bucket_sums.div(bucket_sums.sum(axis="columns"),
.....:                                axis="index")
```

```
In [233]: normed_sums
```

```
Out[233]:
cand_nm      Obama, Barack      Romney, Mitt
contb_receipt_amt
(0, 1]    0.805182 0.194818
(1, 10]   0.918767 0.081233
(10, 100] 0.910769 0.089231
(100, 1000] 0.710176      0.289824
(1000, 10000] 0.447326      0.552674
(10000, 100000] 0.823120      0.176880
(100000, 1000000] 1.000000      0.000000
(1000000, 10000000] 1.000000      0.000000
0.000000
```

```
In [234]: normed_sums[:-2].plot(kind="barh")
```

Я исключил два самых больших интервала, потому что они соответствуют пожертвованиям юридических лиц.

Этот анализ можно уточнить и улучшить во многих направлениях. Например, можно было бы агрегировать пожертвования по имени и почтовому индексу спонсора, чтобы отделить спонсоров, внесших много мелких пожертвований, от тех, кто внес одно или несколько крупных. Призываю вас скачать этот набор данных и исследовать его самостоятельно.

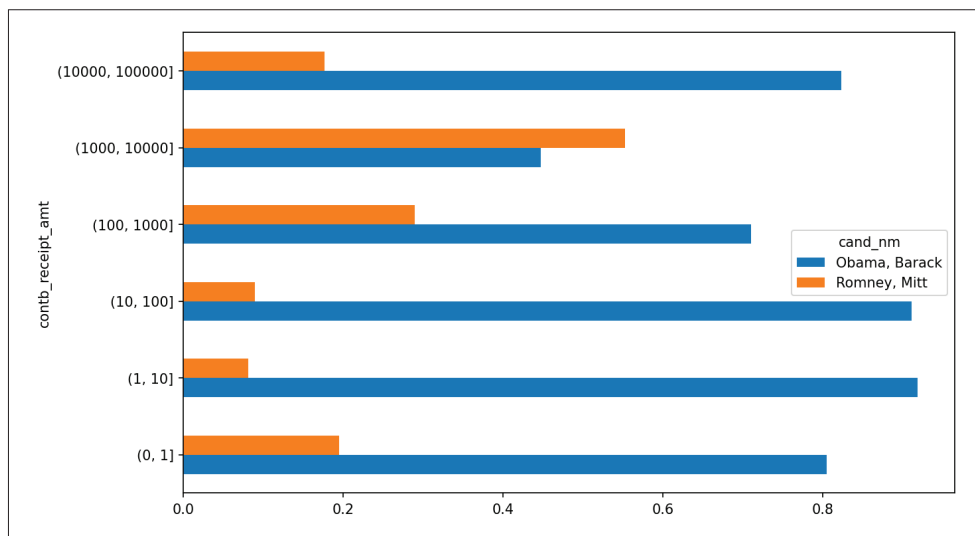


Рис. 13.13. Процентная доля пожертвований каждого размера от общего их числа для обоих кандидатов

Статистика пожертвований по штатам

Начать можно с агрегирования данных по кандидатам и штатам:

```
In [235]: grouped = fec_mrbo.groupby(["cand_nm", "contbr_st"])

In [236]: totals = grouped["contb_receipt_amt"].sum().unstack(level=0).fillna(0)

In [237]: totals = totals[totals.sum(axis="columns") > 100000]

In [238]: totals.head(10)
Out[238]:
cand_nm Obama, Barack      Romney, Mitt
contbr_st
AK 281840.15  86204.24
AL 543123.48  527303.51
AR 359247.28  105556.00
AZ 1506476.98  1888436.23
CA 23824984.24  11237636.60
CO 2132429.49  1506714.12
CT 2068291.26  3499475.45
DC 4373538.80  1025137.50
DE 336669.14  82712.00
FL 7318178.58  8338458.81
```

Поделив каждую строку на общую сумму пожертвований, мы получим для каждого кандидата процентную долю от общей суммы, приходящуюся на каждый штат:


```
In [239]: percent = totals.div(totals.sum(axis="columns"), axis="index")
In [240]: percent.head(10)
Out[240]:
cand_nm Obama, BarackRomney, Mitt
contbr_st
AK 0.765778 0.234222
AL 0.507390 0.492610
AR 0.772902 0.227098
AZ 0.443745 0.556255
CA 0.679498 0.320502
CO 0.585970 0.414030
CT 0.371476 0.628524
DC 0.810113 0.189887
DE 0.802776 0.197224
FL 0.467417 0.532583
```

13.6. ЗАКЛЮЧЕНИЕ

На этом основной текст книги заканчивается. Дополнительные материалы, которые могут быть вам полезны, я включил в приложения.

За десять лет, прошедших с момента публикации первого издания этой книги, Python стал популярным и широко распространенным языком для анализа данных. Полученные в процессе чтения навыки программирования останутся актуальными еще долго. Надеюсь, что рассмотренные нами инструменты и библиотеки сослужат вам хорошую службу.

Приложение А

Дополнительные сведения о библиотеке NumPy

В этом приложении мы глубже рассмотрим библиотеку NumPy, предназначенную для вычислений с массивами. Мы разберемся во внутренних деталях типа `ndarray` и поговорим о дополнительных операциях над массивами и алгоритмах.

Приложение содержит разнородные темы, поэтому читать его последовательно не обязательно. В основных главах я генерирую случайные данные в различных примерах с помощью подразумеваемого по умолчанию генератора случайных чисел из модуля `numpy.random`:

```
In [11]: rng = np.random.default_rng(seed=12345)
```

А.1. ВНУТРЕННЕЕ УСТРОЙСТВО ОБЪЕКТА NDARRAY

Объект `ndarray` из библиотеки NumPy позволяет интерпретировать блок однородных данных (непрерывный или с шагом, подробнее об этом ниже) как многомерный массив. Мы уже видели, что тип данных, или *dtype*, определяет, как именно интерпретируются данные: как числа с плавающей точкой, целые, булевы или еще как-то.

Своей эффективностью `ndarray` отчасти обязан тому, что любой объект массива является *шаговым* (strided) представлением блока данных. Может возникнуть вопрос, как удастся построить представление массива `arr[:,2, :-1]` без копирования данных. Дело в том, что объект `ndarray` – не просто блок памяти, дополненный знанием о типе в виде *dtype*; в нем еще хранится информация, позволяющая перемещаться по массиву шагами разного размера. Точнее, в реализации `ndarray` имеется:

- *указатель на данные*, т. е. на блок полученной от системы памяти;
- *тип данных*, или *dtype*, описывающий значения элементов массива фиксированного размера;
- кортеж, описывающий *форму* массива;
- кортеж *шагов*, т. е. целых чисел, показывающих, на сколько байтов нужно сместиться, чтобы перейти к следующему элементу по некоторому измерению.

На рис. А.1 схематически показано внутреннее устройство `ndarray`.



Рис. А.1. Объект ndarray из библиотеки NumPy

Например, у массива 10×5 будет форма (10, 5):

```
In [12]: np.ones((10, 5)).shape
Out[12]: (10, 5)
```

Для типичного массива (организованного в соответствии с принятым в языке С соглашением) $3 \times 4 \times 5$ чисел типа `float64` (8-байтовых) кортеж шагов имеет вид (160, 40, 8) (знать о шагах полезно, потому что в общем случае чем больше шаг по конкретной оси, тем дороже обходятся вычисления по этой оси):

```
In [13]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[13]: (160, 40, 8)
```

Хотя типичному пользователю NumPy редко приходится интересоваться шагами массива, они играют важнейшую роль в построении представлений массива без копирования. Шаги могут быть даже отрицательными, что позволяет проходить массив в «обратном направлении», как в случае среза вида `obj[::-1]` или `obj[:, ::-1]`.

Иерархия типов данных в NumPy

Иногда в программе необходимо проверить, что хранится в массиве: целые числа, числа с плавающей точкой, строки или объекты Python. Поскольку существует много типов с плавающей точкой (от `float16` до `float128`), для проверки того, что `dtype` присутствует в списке типов, приходится писать длинный код. По счастью, определены такие суперклассы, как `np.integer` или `np.floating`, которые можно использовать в сочетании с функцией `np.issubdtype`:

```
In [14]: ints = np.ones(10, dtype=np.uint16)

In [15]: floats = np.ones(10, dtype=np.float32)

In [16]: np.issubdtype(ints.dtype, np.integer)
Out[16]: True

In [17]: np.issubdtype(floats.dtype, np.floating)
Out[17]: True
```

Вывести все родительские классы данного типа `dtype` позволяет метод `npco`:

```
In [18]: np.float64.mro()
Out[18]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Поэтому мы также имеем:

```
In [19]: np.issubdtype(ints.dtype, np.number)
Out[19]: True
```

Большинству пользователей NumPy об этом знать необязательно, но иногда оказывается удобно. На рис. А.2 показан граф наследования dtype¹².

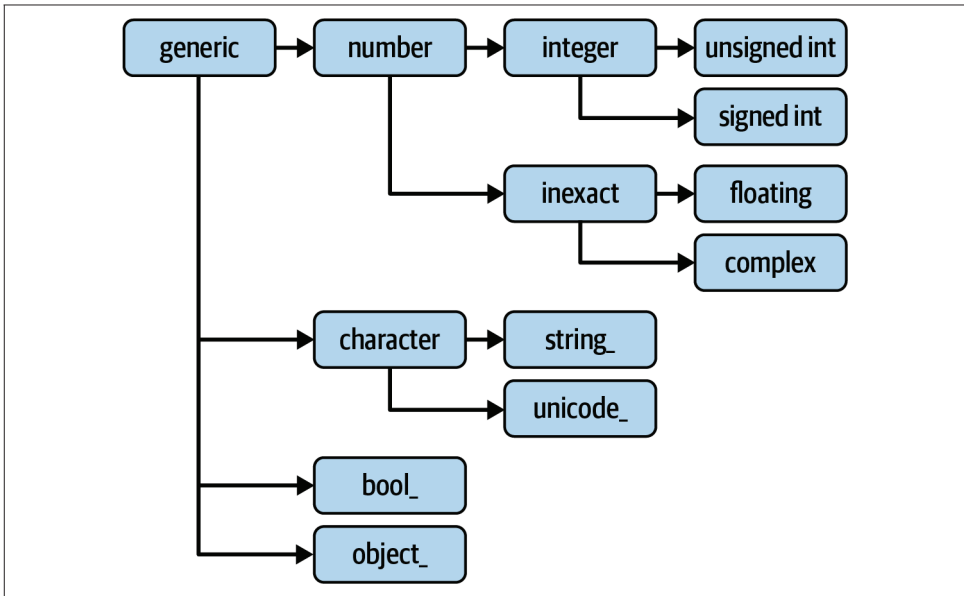


Рис. А.2. Иерархия классов типов данных в NumPy

А.2. ДОПОЛНИТЕЛЬНЫЕ МАНИПУЛЯЦИИ С МАССИВАМИ

Помимо прихотливого индексирования, вырезания и формирования булевых подмножеств, существует много других способов работы с массивами. И хотя большую часть сложных задач, решаемых в ходе анализа данных, берут на себя высокоуровневые функции из библиотеки pandas, иногда возникает необходимость написать алгоритм обработки данных, которого нет в имеющихся библиотеках.

¹² В именах некоторых типов dtype присутствуют знаки подчеркивания. Они нужны, чтобы избежать конфликтов между именами типов NumPy и встроенных типов Python.

Изменение формы массива

Во многих случаях изменить форму массива можно без копирования данных. Для этого следует передать кортеж с описанием новой формы методу экземпляра массива `reshape`. Например, предположим, что имеется одномерный массив, который мы хотели бы преобразовать в матрицу (результат показан на рис. А.3):

```
In [20]: arr = np.arange(8)

In [21]: arr
Out[21]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [22]: arr.reshape((4, 2))
Out[22]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

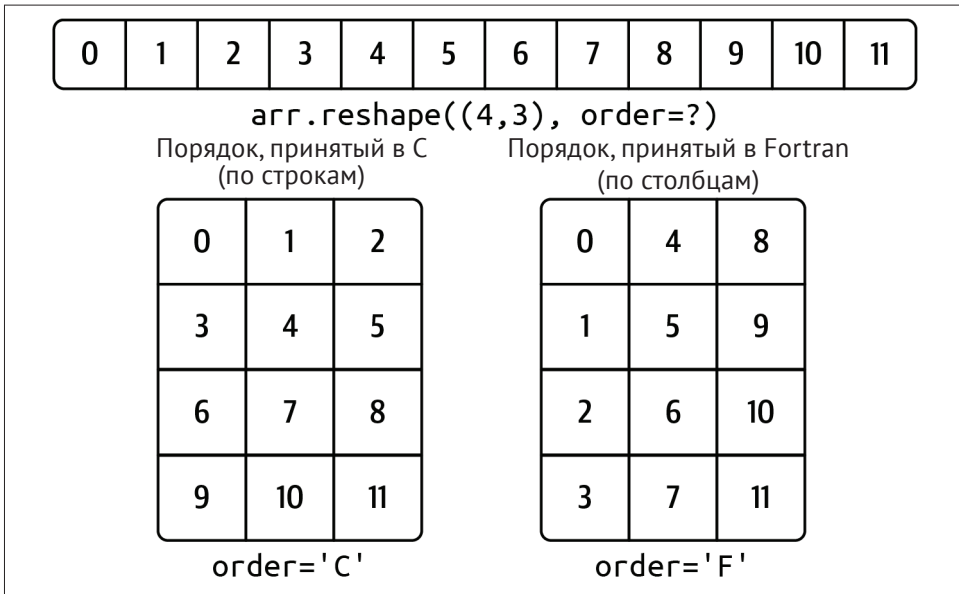


Рис. А.3. Изменение формы с преобразованием в двумерный массив, организованный как в C (по строкам) и как в Fortran (по столбцам)

Форму многомерного массива также можно изменить:

```
In [23]: arr.reshape((4, 2)).reshape((2, 4))
Out[23]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Одно из измерений, переданных в описателе формы, может быть равно `-1`, его значение будет выведено из данных:

```
In [24]: arr = np.arange(15)
```

```
In [25]: arr.reshape((5, -1))
```

```
Out[25]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Поскольку атрибут `shape` массива является кортежем, его также можно передать методу `reshape`:

```
In [26]: other_arr = np.ones((3, 5))
```

```
In [27]: other_arr.shape
```

```
Out[27]: (3, 5)
```

```
In [28]: arr.reshape(other_arr.shape)
```

```
Out[28]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

Обратная операция – переход от многомерного к одномерному массиву – называется **линеаризацией**:

```
In [29]: arr = np.arange(15).reshape((5, 3))
```

```
In [30]: arr
```

```
Out[30]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
In [31]: arr.ravel()
```

```
Out[31]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Метод `ravel` не создает копию данных, если значения, оказавшиеся в результирующем массиве, были соседними в исходном.

Метод `flatten` ведет себя как `ravel`, но всегда возвращает копию данных:

```
In [32]: arr.flatten()
```

```
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Данные можно линеаризовать в разном порядке. Начинаящим пользователям NumPy эта тема может показаться довольно сложной, поэтому ей посвящен целиком следующий подраздел.

Упорядочение элементов массива в C и в Fortran

Библиотека NumPy предлагает большую гибкость в определении порядка размещения данных в памяти. По умолчанию массивы NumPy размещаются *по строкам*. Это означает, что при размещении двумерного массива в памяти соседние элементы строки находятся в соседних ячейках памяти. Альтернативой

является размещение *по столбцам*, тогда в соседних ячейках находятся соседние элементы столбца.

По историческим причинам, порядок размещения по строкам называется порядком C, а по столбцам – порядком Fortran. В языке FORTRAN 77 матрицы размещаются по столбцам.

Функции типа `reshape` и `ravel` принимают аргумент `order`, показывающий, в каком порядке размещать данные в массиве. Обычно задают значение 'C' или 'F' (о менее употребительных значениях 'A' и 'K' можно прочесть в документации по NumPy, а их действие показано на рис. А.3).

```
In [33]: arr = np.arange(12).reshape((3, 4))

In [34]: arr
Out[34]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [35]: arr.ravel()
Out[35]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [36]: arr.ravel('F')
Out[36]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Изменение формы массива, имеющего больше двух измерений, – головомное упражнение (см. рис. А.3). Основное различие между порядком C и Fortran состоит в том, в каком порядке перебираются измерения:

Порядок по строкам (C)

Старшие измерения обходятся *раньше* (т. е. сначала обойти ось 1, а потом переходить к оси 0).

Порядок по столбцам (Fortran)

Старшие измерения обходятся *позже* (т. е. сначала обойти ось 0, а потом переходить к оси 1).

Конкатенация и разбиение массива

Метод `numpy.concatenate` принимает произвольную последовательность (кортеж, список и т. п.) массивов и соединяет их вместе в порядке, определяемом указанной осью.

```
In [37]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [38]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [39]: np.concatenate([arr1, arr2], axis=0)
Out[39]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [40]: np.concatenate([arr1, arr2], axis=1)
Out[40]:
array([[ 1, 2, 3, 7, 8, 9],
       [ 4, 5, 6, 10, 11, 12]])
```

Есть несколько вспомогательных функций, например `vstack` и `hstack`, для выполнения типичных операций конкатенации. Приведенные выше операции можно было бы записать и так:

```
In [41]: np.vstack((arr1, arr2))
Out[41]:
array([[ 1, 2, 3],
       [ 4, 5, 6],
       [ 7, 8, 9],
       [10, 11, 12]])
```

```
In [42]: np.hstack((arr1, arr2))
Out[42]:
array([[ 1, 2, 3, 7, 8, 9],
       [ 4, 5, 6, 10, 11, 12]])
```

С другой стороны, функция `split` разбивает массив на несколько частей вдоль указанной оси:

```
In [43]: arr = rng.standard_normal((5, 2))

In [44]: arr
Out[44]:
array([[ -1.4238,  1.2637],
       [ -0.8707, -0.2592],
       [ -0.0753, -0.7409],
       [ -1.3678,  0.6489],
       [  0.3611, -1.9529]])

In [45]: first, second, third = np.split(arr, [1, 3])

In [46]: first
Out[46]: array([[ -1.4238,  1.2637]])

In [47]: second
Out[47]:
array([[ -0.8707, -0.2592],
       [ -0.0753, -0.7409]])

In [48]: third
Out[48]:
array([[ -1.3678,  0.6489],
       [  0.3611, -1.9529]])
```

Значение `[1, 3]`, переданное `np.split`, содержит индексы, по которым массив нужно разбить на части.

Перечень всех функций, относящихся к конкатенации и разбиению, приведен в табл. А.1, хотя некоторые из них – лишь надстройки над очень общей функцией `concatenate`.

Таблица А.1. Функции конкатенации массива

Функция	Описание
<code>concatenate</code>	Самая общая функция – конкатенирует коллекцию массивов вдоль указанной оси
<code>vstack</code> , <code>row_stack</code>	Составляет массивы по строкам (вдоль оси 0)
<code>hstack</code>	Составляет массивы по столбцам (вдоль оси 1)
<code>column_stack</code>	Аналогична <code>hstack</code> , но сначала преобразует одномерные массивы в двумерные векторы по столбцам
<code>dstack</code>	Составляет массивы в глубину (вдоль оси 2)
<code>split</code>	Разбивает массив в указанных позициях вдоль указанной оси
<code>hsplit</code> / <code>vsplit</code>	Вспомогательные функции для разбиения по оси 0 и 1 соответственно

Вспомогательные объекты: `r_` и `c_`

В пространстве имен NumPy есть два специальных объекта: `r_` и `c_`, благодаря которым составление массивов можно записать более кратко:

```
In [49]: arr = np.arange(6)
```

```
In [50]: arr1 = arr.reshape((3, 2))
```

```
In [51]: arr2 = rng.standard_normal((3, 2))
```

```
In [52]: np.r_[arr1, arr2]
```

```
Out[52]:
```

```
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
       [ 4.    ,  5.    ],
       [ 2.3474,  0.9685],
       [-0.7594,  0.9022],
       [-0.467 , -0.0607]])
```

```
In [53]: np.c_[np.r_[arr1, arr2], arr]
```

```
Out[53]:
```

```
array([[ 0.    ,  1.    ,  0.    ],
       [ 2.    ,  3.    ,  1.    ],
       [ 4.    ,  5.    ,  2.    ],
       [ 2.3474,  0.9685,  3.    ],
       [-0.7594,  0.9022,  4.    ],
       [-0.467 , -0.0607,  5.    ]])
```

С их помощью можно также преобразовывать срезы в массивы:

```
In [55]: arr = np.arange(3)
```

```
In [56]: arr
```

```
Out[56]: array([0, 1, 2])
```

```
In [57]: arr.repeat(3)
```

```
Out[57]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```



Необходимость повторять массивы при работе с NumPy возникает реже, чем в других популярных средах программирования, например MATLAB. Основная причина заключается в том, что *укладывание* (тема следующего раздела) решает эту задачу лучше.

По умолчанию, если передать целое число, то каждый элемент повторяется столько раз. Если же передать массив целых чисел, то разные элементы могут быть повторены разное число раз:

```
In [58]: arr.repeat([2, 3, 4])
Out[58]: array([0, 0, 1, 1, 1, 2, 2, 2, 2, 2])
```

Элементы многомерных массивов повторяются вдоль указанной оси:

```
In [59]: arr = rng.standard_normal((2, 2))
```

```
In [60]: arr
Out[60]:
array([[ 0.7888, -1.2567],
       [ 0.5759,  1.399 ]])
```

```
In [61]: arr.repeat(2, axis=0)
Out[61]:
array([[ 0.7888, -1.2567],
       [ 0.7888, -1.2567],
       [ 0.5759,  1.399 ],
       [ 0.5759,  1.399 ]])
```

Отметим, что если ось не указана, то массив сначала линейризуется, а это, скорее всего, не то, что вы хотели. Чтобы повторить разные срезы многомерного массива различное число раз, можно передать массив целых чисел:

```
In [62]: arr.repeat([2, 3], axis=0)
Out[62]:
array([[ 0.7888, -1.2567],
       [ 0.7888, -1.2567],
       [ 0.5759,  1.399 ],
       [ 0.5759,  1.399 ],
       [ 0.5759,  1.399 ]])

In [63]: arr.repeat([2, 3], axis=1)
Out[63]:
array([[ 0.7888,  0.7888, -1.2567, -1.2567, -1.2567],
       [ 0.5759,  0.5759,  1.399 ,  1.399 ,  1.399 ]])
```

Функция `tile` (замостить), с другой стороны, – просто сокращенный способ составления копий массива вдоль оси. Это можно наглядно представлять себе как «укладывание плиток»:

```
In [64]: arr
Out[64]:
array([[ 0.7888, -1.2567],
       [ 0.5759,  1.399 ]])

In [65]: np.tile(arr, 2)
```

```
Out[65]:
array([[ 0.7888, -1.2567, 0.7888, -1.2567],
       [ 0.5759,  1.399 , 0.5759,  1.399 ]])
```

Второй аргумент – количество плиток; если это скаляр, то мощение производится по строкам, а не по столбцам. Но второй аргумент `tile` может быть кортежем, описывающим порядок мощения:

```
In [66]: arr
Out[66]:
array([[ 0.7888, -1.2567],
       [ 0.5759,  1.399 ]])

In [67]: np.tile(arr, (2, 1))
Out[67]:
array([[ 0.7888, -1.2567],
       [ 0.5759,  1.399 ],
       [ 0.7888, -1.2567],
       [ 0.5759,  1.399 ]])

In [68]: np.tile(arr, (3, 2))
Out[68]:
array([[ 0.7888, -1.2567, 0.7888, -1.2567],
       [ 0.5759,  1.399 , 0.5759,  1.399 ],
       [ 0.7888, -1.2567, 0.7888, -1.2567],
       [ 0.5759,  1.399 , 0.5759,  1.399 ],
       [ 0.7888, -1.2567, 0.7888, -1.2567],
       [ 0.5759,  1.399 , 0.5759,  1.399 ]])
```

Эквиваленты прихотливого индексирования: функции `take` и `put`

В главе 4 описывался способ получить и установить подмножество массива с помощью *прихотливого* индексирования массивами целых чисел:

```
In [69]: arr = np.arange(10) * 100

In [70]: inds = [7, 1, 2, 6]

In [71]: arr[inds]
Out[71]: array([700, 100, 200, 600])
```

Существуют и другие методы `ndarray`, полезные в частном случае, когда выборка производится только по одной оси:

```
In [72]: arr.take(inds)
Out[72]: array([700, 100, 200, 600])

In [73]: arr.put(inds, 42)

In [74]: arr
Out[74]: array([ 0, 42, 42, 300, 400, 500, 42, 42, 800, 900])

In [75]: arr.put(inds, [40, 41, 42, 43])

In [76]: arr
Out[76]: array([ 0, 41, 42, 300, 400, 500, 43, 40, 800, 900])
```

Чтобы использовать функцию `take` для других осей, нужно передать именованный параметр `axis`:

```
In [77]: inds = [2, 0, 2, 1]

In [78]: arr = rng.standard_normal((2, 4))

In [79]: arr
Out[79]:
array([[ 1.3223, -0.2997,  0.9029, -1.6216],
       [-0.1582,  0.4495, -1.3436, -0.0817]])

In [80]: arr.take(inds, axis=1)
Out[80]:
array([[ 0.9029,  1.3223,  0.9029, -0.2997],
       [-1.3436, -0.1582, -1.3436,  0.4495]])
```

Функция `put` не принимает аргумент `axis`, а обращается по индексу к линейризованной версии массива (одномерному массиву в порядке C). Следовательно, если с помощью массива индексов требуется установить элементы на других осях, то придется воспользоваться прихотливым индексированием.

А.3. Укладывание

Словом «укладывание» (broadcasting) описывается способ выполнения арифметических операций над массивами разной формы. Это очень мощный механизм, но даже опытные пользователи иногда испытывают затруднения с его пониманием. Простейший пример укладывания – комбинирование скалярного значения с массивом:

```
In [81]: arr = np.arange(5)

In [82]: arr
Out[82]: array([0, 1, 2, 3, 4])

In [83]: arr * 4
Out[83]: array([ 0,  4,  8, 12, 16])
```

Здесь мы говорим, что скалярное значение 4 *уложено* на все остальные элементы путем умножения.

Другой пример: мы можем сделать среднее по столбцам массива равным нулю, вычтя из каждого столбца столбец, содержащий средние значения. И сделать это очень просто:

```
In [84]: arr = rng.standard_normal((4, 3))

In [85]: arr.mean(0)
Out[85]: array([0.1206,  0.243 ,  0.1444])

In [86]: demeaned = arr - arr.mean(0)

In [87]: demeaned
Out[87]:
```

```
array([[ 1.6042,  2.3751,  0.633 ],
       [ 0.7081, -1.202 , -1.3538],
       [-1.5329,  0.2985,  0.6076],
       [-0.7793, -1.4717,  0.1132]])
```

```
In [88]: demeaned.mean(0)
Out[88]: array([ 0., -0.,  0.])
```

Эта операция показана на рис. А.4. Для приведения к нулю средних по строкам с помощью укладывания требуется проявить осторожность. К счастью, укладывание значений меньшей размерности вдоль любого измерения массива (например, вычитание средних по строкам из каждого столбца двумерного массива) возможно при соблюдении следующего правила:

Правило укладывания

Два массива совместимы по укладыванию, если для обоих *последних измерений* (т. е. отсчитываемых с конца) длины осей совпадают или хотя бы одна длина равна 1. Тогда укладывание производится по отсутствующим измерениям или по измерениям длины 1.

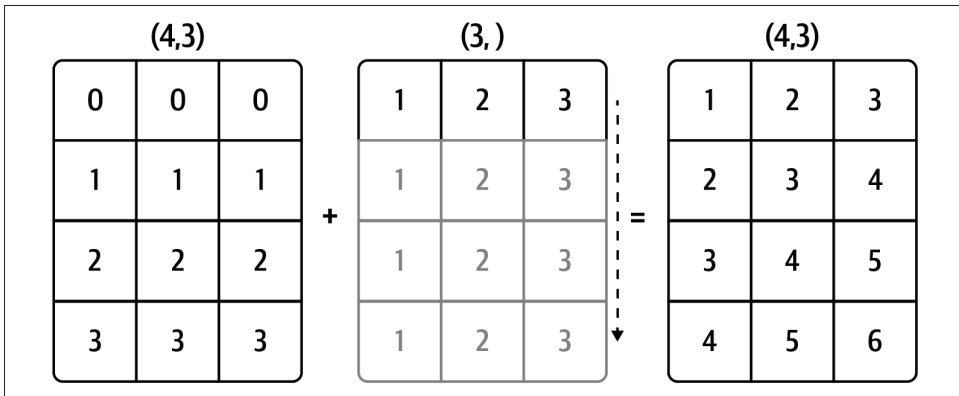


Рис. А.4. Укладывание одномерного массива по оси 0

Даже я, опытный пользователь NumPy, иногда вынужден рисовать картинки, чтобы понять, как будет применяться правило укладывания. Вернемся к последнему примеру и предположим, что мы хотим вычесть среднее значение из каждой строки, а не из каждого столбца. Поскольку длина массива `arr.mean(0)` равна 3, он совместим по укладыванию вдоль оси 0, т. к. по последнему измерению длины осей (три) совпадают. Согласно правилу, чтобы произвести вычитание по оси 1 (т. е. вычесть среднее по строкам из каждой строки), меньший массив должен иметь форму (4, 1):

```

In [89]: arr
Out[89]:
array([[ 1.7247,  2.6182,  0.7774],
       [ 0.8286, -0.959 , -1.2094],
       [-1.4123,  0.5415,  0.7519],
       [-0.6588, -1.2287,  0.2576]])

In [90]: row_means = arr.mean(1)

In [91]: row_means.shape
Out[91]: (4,)

In [92]: row_means.reshape((4, 1))
Out[92]:
array([[ 1.7068],
       [-0.4466],
       [-0.0396],
       [-0.5433]])

In [93]: demeaned = arr - row_means.reshape((4, 1))

In [94]: demeaned.mean(1)
Out[94]: array([-0.,  0.,  0.,  0.])
    
```

Эта операция проиллюстрирована на рис. А.5.

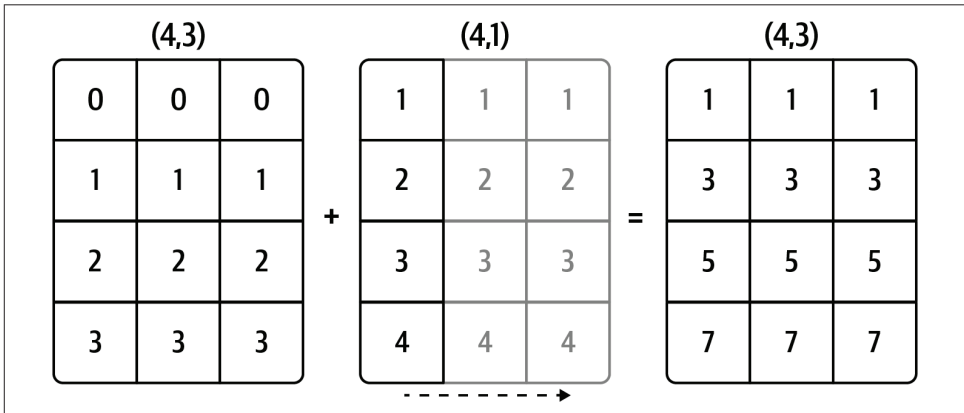


Рис. А.5. Укладывание по оси 1 двумерного массива

На рис. А.6 приведена еще одна иллюстрация, где мы вычитаем двумерный массив из трехмерного по оси 0.

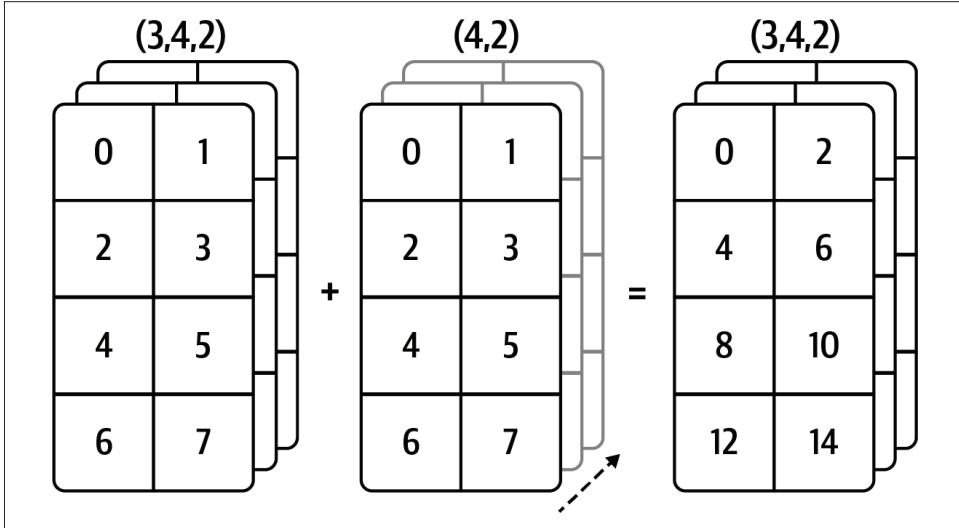


Рис. А.6. Укладывание по оси 0 трехмерного массива

Укладывание по другим осям

Укладывание многомерных массивов может показаться еще более головоломной задачей, но на самом деле нужно только соблюдать правило. В противном случае будет выдана ошибка вида:

```
In [95]: arr - arr.mean(1)
-----
ValueError                                Traceback (most recent call last)
<ipython-input-95-8b8ada26fac0> in <module>
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

Очень часто возникает необходимость выполнить арифметическую операцию с массивом меньшей размерности по оси, отличной от 0. Согласно правилу укладывания, длина «размерности укладывания» в меньшем массиве должна быть равна 1. В примере вычитания среднего это означало, что массив средних по строкам должен иметь форму (4, 1), а не (4,):

```
In [96]: arr - arr.mean(1).reshape((4, 1))
Out[96]:
array([[ 0.018 ,  0.9114, -0.9294],
       [ 1.2752, -0.5124, -0.7628],
       [-1.3727,  0.5811,  0.7915],
       [-0.1155, -0.6854,  0.8009]])
```

В трехмерном случае укладывание по любому из трех измерений сводится к изменению формы данных для обеспечения совместимости массивов. На рис. А.7 наглядно показано, каковы должны быть формы для укладывания по любой оси трехмерного массива.

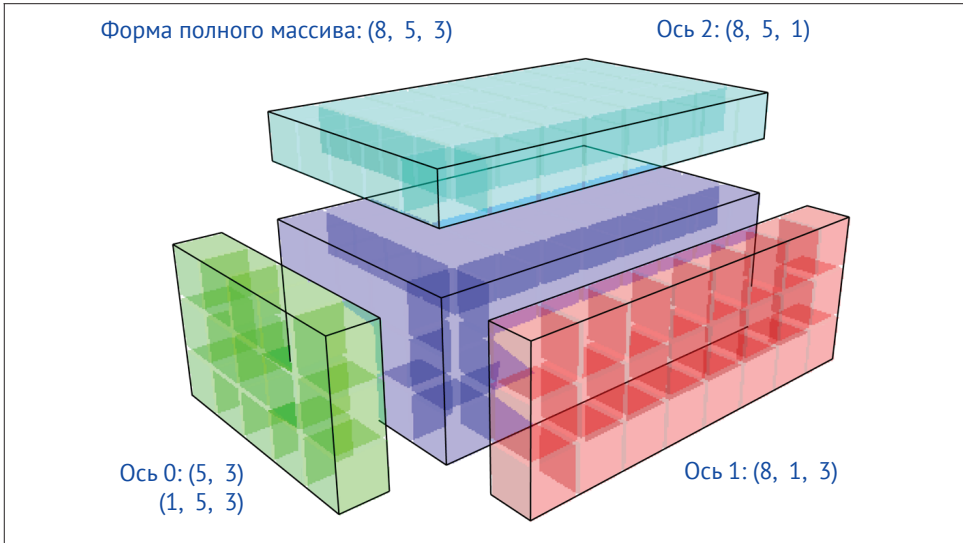


Рис. А.7. Совместимые формы двумерного массива для укладывания в трехмерный массив

Поэтому часто приходится добавлять новую ось длины 1 специально для укладывания. Один из вариантов – использование `reshape`, но для вставки оси нужно построить кортеж, описывающий новую форму. Это утомительное занятие. Поэтому в NumPy имеется специальный синтаксис для вставки новых осей путем доступа по индексу. Чтобы вставить новую ось, мы воспользуемся специальным атрибутом `np.newaxis` и «полными» срезами:

```
In [97]: arr = np.zeros((4, 4))

In [98]: arr_3d = arr[:, np.newaxis, :]

In [99]: arr_3d.shape
Out[99]: (4, 1, 4)

In [100]: arr_1d = rng.standard_normal(3)

In [101]: arr_1d[:, np.newaxis]
Out[101]:
array([[ 0.3129],
       [-0.1308],
       [ 1.27   ]])

In [102]: arr_1d[np.newaxis, :]
Out[102]: array([[ 0.3129, -0.1308,  1.27   ]])
```

Таким образом, если имеется трехмерный массив и требуется привести его к нулевому среднему по оси 2, то нужно написать:

```
In [103]: arr = rng.standard_normal((3, 4, 5))

In [104]: depth_means = arr.mean(2)
```



```

In [105]: depth_means
Out[105]:
array([[ 0.0431,  0.2747, -0.1885, -0.2014],
       [-0.5732, -0.5467,  0.1183, -0.6301],
       [ 0.0972,  0.5954,  0.0331, -0.6002]])

In [106]: depth_means.shape
Out[106]: (3, 4)

In [107]: demeaned = arr - depth_means[:, :, np.newaxis]

In [108]: demeaned.mean(2)
Out[108]:
array([[ 0., -0.,  0., -0.],
       [ 0., -0., -0., -0.],
       [ 0.,  0.,  0.,  0.]])

```

Возможно, вас интересует, нет ли способа обобщить вычитание среднего вдоль оси, не жертвуя производительностью. Есть, но придется попотеть с индексированием:

```

def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # Это обобщает операции вида [:, :, np.newaxis] на N измерений
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]

```

Установка элементов массива с помощью укладывания

То же правило укладывания, что управляет арифметическими операциями, применимо и к установке значений элементов с помощью доступа по индексу. В простейшем случае это выглядит так:

```

In [109]: arr = np.zeros((4, 3))

In [110]: arr[:] = 5

In [111]: arr
Out[111]:
array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]])

```

Если имеется одномерный массив значений, который требуется записать в столбцы массива, то можно сделать и это – при условии совместимости формы:

```

In [112]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [113]: arr[:, :] = col[:, np.newaxis]

In [114]: arr
Out[114]:

```

```
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [115]: arr[:2] = [[-1.37], [0.509]]

In [116]: arr
Out[116]:
array([[ -1.37 , -1.37 , -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6 ,  1.6 ,  1.6 ]])
```

А.4. ДОПОЛНИТЕЛЬНЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ УНИВЕРСАЛЬНЫХ ФУНКЦИЙ

Многие пользователи NumPy используют универсальные функции только ради быстрого выполнения поэлементных операций, однако у них есть и другие возможности, которые иногда позволят кратко записать код без циклов.

Методы экземпляра u-функций

Любая бинарная u-функция в NumPy имеет специальные методы для выполнения некоторых видов векторных операций. Все они перечислены в табл. А.2, но я приведу и несколько конкретных примеров для иллюстрации.

Метод `reduce` принимает массив и агрегирует его, возможно вдоль указанной оси, выполняя последовательность бинарных операций. Вот, например, как можно с помощью `np.add.reduce` просуммировать элементы массива:

```
In [117]: arr = np.arange(10)

In [118]: np.add.reduce(arr)
Out[118]: 45

In [119]: arr.sum()
Out[119]: 45
```

Начальное значение (для `add` оно равно 0) зависит от u-функции. Если задана ось, то редукция производится вдоль этой оси. Это позволяет давать краткие ответы на некоторые вопросы. В качестве не столь скучного примера воспользуемся методом `np.logical_and`, чтобы проверить, отсортированы ли значения в каждой строке массива:

```
In [120]: my_rng = np.random.default_rng(12346) # для воспроизводимости

In [121]: arr = my_rng.standard_normal((5, 5))

In [122]: arr
Out[122]:
array([[ -0.9039,  0.1571,  0.8976, -0.7622, -0.1763],
       [ 0.053 , -1.6284, -0.1775,  1.9636,  1.7813],
       [-0.8797, -1.6985, -1.8189,  0.119 , -0.4441],
       [ 0.7691, -0.0343,  0.3925,  0.7589, -0.0705],
       [ 1.0498,  1.0297, -0.4201,  0.7863,  0.9612]])
```

```
In [123]: arr[:, :2].sort(1) # отсортировать несколько строк
```

```
In [124]: arr[:, :-1] < arr[:, 1:]
```

```
Out[124]:
```

```
array([[ True,  True,  True,  True],
       [False,  True,  True,  False],
       [ True,  True,  True,  True],
       [False,  True,  True,  False],
       [ True,  True,  True,  True]])
```

Отметим, что `logical_and.reduce` эквивалентно методу `all`.

Функция `accumulate` соотносится с `reduce`, как `cumsum` с `sum`. Она порождает массив того же размера, содержащий промежуточные «аккумулированные» значения:

```
In [126]: arr = np.arange(15).reshape((3, 5))
```

```
In [127]: np.add.accumulate(arr, axis=1)
```

```
Out[127]:
```

```
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

Функция `outer` вычисляет прямое произведение двух массивов:

```
In [128]: arr = np.arange(3).repeat([1, 2, 2])
```

```
In [129]: arr
```

```
Out[129]: array([0, 1, 1, 2, 2])
```

```
In [130]: np.multiply.outer(arr, np.arange(5))
```

```
Out[130]:
```

```
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

Размерность массива, возвращенного `outer`, является результатом конкатенации размерностей его параметров:

```
In [131]: x, y = rng.standard_normal((3, 4)), rng.standard_normal(5)
```

```
In [132]: result = np.subtract.outer(x, y)
```

```
In [133]: result.shape
```

```
Out[133]: (3, 4, 5)
```

Последний метод, `reduceat`, выполняет «локальную редукцию», т. е. по существу операцию `groupby`, в которой агрегируется сразу несколько срезов массива. Он принимает последовательность «границ интервалов», описывающую, как разбивать и агрегировать значения:

```
In [134]: arr = np.arange(10)
```

```
In [135]: np.add.reduceat(arr, [0, 5, 8])
```

```
Out[135]: array([10, 18, 17])
```

На выходе получаются результаты редукции (в данном случае суммирования) по срезам `arr[0:5]`, `arr[5:8]` и `arr[8:]`. Как и другие методы, `reduceat` принимает необязательный аргумент `axis`:

```
In [136]: arr = np.multiply.outer(np.arange(4), np.arange(5))
```

```
In [137]: arr
Out[137]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])
```

```
In [138]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[138]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

Неполный перечень и-функций приведен в табл. A.2.

Таблица A.2. Методы и-функций

Метод	Описание
<code>accumulate(x)</code>	Агрегирует значения, сохраняя все промежуточные агрегаты
<code>at(x, indices, b=None)</code>	Выполняет операцию над <code>x</code> на месте в точках, определяемых заданными индексами. Аргумент <code>b</code> передается вторым аргументом и-функциям, принимающим два массива
<code>reduce(x)</code>	Агрегирует значения путем последовательного применения операции
<code>reduceat(x, bins)</code>	«Локальная» редукция, или «group by». Редуцирует соседние срезы данных и порождает массив агрегатов
<code>outer(x, y)</code>	Применяет операцию ко всем парам элементов <code>x</code> и <code>y</code> . Результирующий массив имеет форму <code>x.shape + y.shape</code>

Написание новых и-функций на Python

Для создания собственных и-функций для NumPy существует несколько механизмов. Самый общий – использовать C API NumPy, но он выходит за рамки этой книги. В этом разделе мы будем рассматривать и-функции на чистом Python.

Метод `numpy.frompyfunc` принимает функцию Python и спецификацию количества входов и выходов. Например, простую функцию, выполняющую поэлементное сложение, можно было бы задать так:

```
In [139]: def add_elements(x, y):
.....:     return x + y

In [140]: add_them = np.frompyfunc(add_elements, 2, 1)

In [141]: add_them(np.arange(8), np.arange(8))
Out[141]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Функции, созданные методом `frompyfunc`, всегда возвращают массивы объектов Python, что не очень удобно. По счастью, есть альтернативный, хотя и не столь функционально богатый метод `numpy.vectorize`, который позволяет задать выходной тип:

```
In [142]: add_them = np.vectorize(add_elements, otypes=[np.float64])
```

```
In [143]: add_them(np.arange(8), np.arange(8))
Out[143]: array([ 0., 2., 4., 6., 8., 10., 12., 14.])
```

Оба метода позволяют создавать аналоги *u-функций*, которые, правда, работают очень медленно, потому что должны вызывать функцию Python для вычисления каждого элемента, а это далеко не так эффективно, как циклы в написанных на C универсальных функциях NumPy:

```
In [144]: arr = rng.standard_normal(10000)
```

```
In [145]: %timeit add_them(arr, arr)
2.43 ms +- 30.5 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [146]: %timeit np.add(arr, arr)
2.88 us +- 47.9 ns per loop (mean +- std. dev. of 7 runs, 100000 loops each)
```

Ниже в этом приложении мы покажем, как создавать быстрые *u-функции* на Python с помощью библиотеки Numba (<http://numba.pydata.org/>).

A.5. СТРУКТУРНЫЕ МАССИВЫ И МАССИВЫ ЗАПИСЕЙ

Вы, наверное, обратили внимание, что все рассмотренные до сих пор примеры `ndarray` были контейнерами *однородных* данных, т. е. блоками памяти, в которых каждый элемент занимает одно и то же количество байтов, определяемое типом данных `dtype`. Создается впечатление, что представить в виде массива неоднородные данные, как в таблице, невозможно. *Структурный массив* – это объект `ndarray`, в котором каждый элемент можно рассматривать как аналог *структуры* (`struct`) в языке C (отсюда и название «структурный») или строки в таблице SQL, содержащие несколько именованных полей:

```
In [147]: dtype = [('x', np.float64), ('y', np.int32)]
```

```
In [148]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)
```

```
In [149]: sarr
Out[149]: array([(1.5, 6), (3.1416, -2)], dtype=[('x', '<f8'), ('y', '<i4')])
```

Существует несколько способов задать структурный `dtype` (см. документацию по NumPy в сети). Наиболее распространенный – с помощью списка кортежей вида `(field_name, field_data_type)`. Теперь элементами массива являются кортежоподобные объекты, к элементам которых можно обращаться как к словарю:

```
In [150]: sarr[0]
Out[150]: (1.5, 6)
```

```
In [151]: sarr[0]['y']
Out[151]: 6
```

Имена полей хранятся в атрибуте `dtype.names`. При доступе к полю структурного массива возвращается шаговое представление данных, т. е. копирования не происходит:

```
In [152]: sarr['x']
Out[152]: array([1.5 , 3.1416])
```

Вложенные типы данных и многомерные поля

При описании структурного `dtype` можно факультативно передать форму (в виде целого числа или кортежа):

```
In [153]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [154]: arr = np.zeros(4, dtype=dtype)

In [155]: arr
Out[155]:
array([(0, 0, 0), 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

В данном случае поле `x` в каждой записи ссылается на массив длиной 3:

```
In [156]: arr[0]['x']
Out[156]: array([0, 0, 0])
```

При этом результатом операции `arr['x']` является двумерный массив, а не одномерный, как в предыдущих примерах:

```
In [157]: arr['x']
Out[157]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Это позволяет представлять более сложные вложенные структуры в виде одного блока памяти в массиве. Допускаются также вложенные типы данных, что позволяет создавать еще более сложные структуры. Например:

```
In [158]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]

In [159]: data = np.array([(1, 2), 5], [(3, 4), 6], dtype=dtype)

In [160]: data['x']
Out[160]: array([(1., 2.), (3., 4.)], dtype=[('a', '<f8'), ('b', '<f4')])

In [161]: data['y']
Out[161]: array([5, 6], dtype=int32)

In [162]: data['x']['a']
Out[162]: array([1., 3.])
```

Объект `DataFrame` из библиотеки `pandas` не поддерживает этот механизм напрямую, хотя иерархическое индексирование в чем-то похоже.

Зачем нужны структурные массивы?

По сравнению с объектом `DataFrame` из `pandas`, структурные массивы `NumPy` – средство более низкого уровня. Они позволяют интерпретировать блок памяти как табличную структуру с вложенными столбцами. Поскольку каждый элемент представлен в памяти фиксированным количеством байтов, структурный массив дает очень эффективный способ записи данных на диск и чтения с диска (в том числе в файлы, отображенные на память, о чем речь пойдет ниже), передачи по сети и прочих операций такого рода. Расположение значений в памяти структурированного массива определяется двоичным представлением структурных типов данных в языке программирования C.

Еще одно распространенное применение структурных массивов связано со стандартным способом сериализации данных в C и C++, часто встречающимся в унаследованных системах; данные выводятся в файл в виде потока байтов с фиксированной длиной записи. Коль скоро известен формат файла (размер каждой записи, порядок байтов и тип данных каждого элемента), данные можно прочитать в память методом `np.fromfile`. Подобные специализированные применения выходят за рамки этой книги, но знать об их существовании полезно.

А.6. ЕЩЕ О СОРТИРОВКЕ

Как и у встроенных списков Python, метод `sort` объекта производит сортировку *на месте*, т. е. массив переупорядочивается без порождения нового массива:

```
In [163]: arr = rng.standard_normal(6)

In [164]: arr.sort()

In [165]: arr
Out[165]: array([-1.1553, -0.9319, -0.5218, -0.4745, -0.1649, 0.03 ])
```

Сортируя на месте, не забывайте, что если сортируемый массив – представление другого массива `ndarray`, то модифицируется исходный массив:

```
In [166]: arr = rng.standard_normal((3, 5))

In [167]: arr
Out[167]:
array([[ -1.1956,  0.4691, -0.3598,  1.0359,  0.2267],
       [-0.7448, -0.5931, -1.055 , -0.0683,  0.458 ],
       [-0.07  ,  0.1462, -0.9944,  1.1436,  0.5026]])

In [168]: arr[:, 0].sort() # Sort first column values in place

In [169]: arr
Out[169]:
array([[ -1.1956,  0.4691, -0.3598,  1.0359,  0.2267],
       [-0.7448, -0.5931, -1.055 , -0.0683,  0.458 ],
       [-0.07  ,  0.1462, -0.9944,  1.1436,  0.5026]])
```

С другой стороны, функция `numpy.sort` создает отсортированную копию массива, принимая те же самые аргументы (в частности, `kind`), что и метод `ndarray.sort`:

```
In [170]: arr = rng.standard_normal(5)

In [171]: arr
Out[171]: array([ 0.8981, -1.1704, -0.2686, -0.796 , 1.4522])

In [172]: np.sort(arr)
Out[172]: array([-1.1704, -0.796 , -0.2686, 0.8981, 1.4522])

In [173]: arr
Out[173]: array([ 0.8981, -1.1704, -0.2686, -0.796 , 1.4522])
```

Все методы сортировки принимают аргумент `axis`, что позволяет независимо сортировать участки массива вдоль указанной оси:

```
In [174]: arr = rng.standard_normal((3, 5))

In [175]: arr
Out[175]:
array([[ -0.2535,  2.1183,  0.3634, -0.6245,  1.1279],
       [ 1.6164, -0.2287, -0.6201, -0.1143, -1.2067],
       [-1.0872, -2.1518, -0.6287, -1.3199,  0.083 ]])

In [176]: arr.sort(axis=1)

In [177]: arr
Out[177]:
array([[ -0.6245, -0.2535,  0.3634,  1.1279,  2.1183],
       [-1.2067, -0.6201, -0.2287, -0.1143,  1.6164],
       [-2.1518, -1.3199, -1.0872, -0.6287,  0.083 ]])
```

Вероятно, вы обратили внимание, что ни у одного метода нет параметра, который задавал бы сортировку в порядке убывания. Это реальная проблема, потому что вырезание массива порождает представления, т. е. копия не создается, что позволяет избежать большого объема вычислений. Но многие пользователи Python знают, что если `values` – список, то `values[::-1]` возвращает его в обратном порядке. То же справедливо и для объектов `ndarray`:

```
In [178]: arr[:, ::-1]
Out[178]:
array([[ 2.1183,  1.1279,  0.3634, -0.2535, -0.6245],
       [ 1.6164, -0.1143, -0.2287, -0.6201, -1.2067],
       [ 0.083 , -0.6287, -1.0872, -1.3199, -2.1518]])
```

Косвенная сортировка: методы `argsort` и `lexsort`

В ходе анализа данных очень часто возникает необходимость переупорядочить набор данных по одному или нескольким ключам. Например, отсортировать таблицу, содержащую данные о студентах, сначала по фамилии, а потом по имени. Это пример *косвенной* сортировки, и если вы читали главы, относящиеся к библиотеке `pandas`, то видели много других примеров более высокого уровня. Имея один или несколько ключей (массив или несколько массивов значений), мы хотим получить массив целочисленных *индексов* (буду называть их просто *индексаторами*), который говорит, как переупорядочить данные в нужном порядке сортировки. Для этого существуют два основных метода: `argsort` и `numpy.lexsort`. Вот пример:


```
In [179]: values = np.array([5, 0, 1, 3, 2])
```

```
In [180]: indexer = values.argsort()
```

```
In [181]: indexer
```

```
Out[181]: array([1, 2, 4, 3, 0])
```

```
In [182]: values[indexer]
```

```
Out[182]: array([0, 1, 2, 3, 5])
```

А в следующем, более сложном примере двумерный массив переупорядочивается по первой строке:

```
In [183]: arr = rng.standard_normal((3, 5))
```

```
In [184]: arr[0] = values
```

```
In [185]: arr
```

```
Out[185]:
```

```
array([[ 5.      ,  0.      ,  1.      ,  3.      ,  2.      ],
       [-0.7503, -2.1268, -1.391 , -0.4922,  0.4505],
       [ 0.8926, -1.0479,  0.9553,  0.2936,  0.5379]])
```

```
In [186]: arr[:, arr[0].argsort()]
```

```
Out[186]:
```

```
array([[ 0.      ,  1.      ,  2.      ,  3.      ,  5.      ],
       [-2.1268, -1.391 ,  0.4505, -0.4922, -0.7503],
       [-1.0479,  0.9553,  0.5379,  0.2936,  0.8926]])
```

Метод `lexsort` аналогичен `argsort`, но выполняет косвенную лексикографическую сортировку по нескольким массивам ключей. Пусть требуется отсортировать данные, идентифицируемые именем и фамилией:

```
In [187]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])
```

```
In [188]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])
```

```
In [189]: sorter = np.lexsort((first_name, last_name))
```

```
In [190]: sorter
```

```
Out[190]: array([1, 2, 3, 0, 4])
```

```
In [191]: list(zip(last_name[sorter], first_name[sorter]))
```

```
Out[191]:
```

```
[('Arnold', 'Jane'),
 ('Arnold', 'Steve'),
 ('Jones', 'Bill'),
 ('Jones', 'Bob'),
 ('Walters', 'Barbara')]
```

Поначалу метод `lexsort` может вызвать недоумение, потому что первым для сортировки используется ключ, указанный в *последнем* массиве. Как видите, ключ `last_name` использовался раньше, чем `first_name`.

Альтернативные алгоритмы сортировки

Устойчивый алгоритм сортировки сохраняет относительные позиции равных элементов. Это особенно важно при косвенной сортировке, когда относительный порядок имеет значение:

```
In [192]: values = np.array(['2:first', '2:second', '1:first', '1:second',
.....:                      '1:third'])

In [193]: key = np.array([2, 2, 1, 1, 1])

In [194]: indexer = key.argsort(kind='mergesort')

In [195]: indexer
Out[195]: array([2, 3, 4, 0, 1])

In [196]: values.take(indexer)
Out[196]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<U8')
```

Единственный имеющийся устойчивый алгоритм сортировки с гарантированным временем работы $O(n \log n)$ – mergesort, но его производительность в среднем хуже, чем у алгоритма quicksort. В табл. А.3 перечислены имеющиеся алгоритмы, их сравнительное быстродействие и гарантированная производительность. Большинству пользователей эта информация не особенно интересна, но знать о ее существовании стоит.

Таблица А.3. Алгоритмы сортировки массива

Алгоритм	Быстродействие	Устойчивый	Рабочая память	В худшем случае
'quicksort'	1	Нет	0	$O(n^2)$
'mergesort'	2	Да	$n / 2$	$O(n \log n)$
'heapsort'	3	Нет	0	$O(n \log n)$

Частичная сортировка массивов

Одна из целей сортировки – найти наибольший или наименьший элемент массива. В NumPy имеются оптимизированные методы, `numpy.partition` и `np.argpartition`, для разделения массива по k -му наименьшему элементу:

```
In [197]: rng = np.random.default_rng(12345)

In [198]: arr = rng.standard_normal(20)

In [199]: arr
Out[199]:
array([-1.4238,  1.2637, -0.8707, -0.2592, -0.0753, -0.7409, -1.3678,
        0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,  0.9022,
       -0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ])

In [200]: np.partition(arr, 3)
Out[200]:
```

```
array([-1.9529, -1.4238, -1.3678, -1.2567, -0.8707, -0.7594, -0.7409,
       -0.0607,  0.3611, -0.0753, -0.2592, -0.467 ,  0.5759,  0.9022,
        0.9685,  0.6489,  0.7888,  1.2637,  1.399 ,  2.3474])
```

После вызова `partition(arr, 3)` первые три элемента результата – это три наименьших значения в произвольном порядке. Метод `numpy.argpartition`, похожий на `numpy.argsort`, возвращает индексы элементов, определяющие эквивалентный порядок:

```
In [201]: indices = np.argpartition(arr, 3)

In [202]: indices
Out[202]:
array([ 9,  0,  6, 17,  2, 12,  5, 15,  8,  4,  3, 14, 18, 13, 11,  7, 16,
        1, 19, 10])

In [203]: arr.take(indices)
Out[203]:
array([-1.9529, -1.4238, -1.3678, -1.2567, -0.8707, -0.7594, -0.7409,
       -0.0607,  0.3611, -0.0753, -0.2592, -0.467 ,  0.5759,  0.9022,
        0.9685,  0.6489,  0.7888,  1.2637,  1.399 ,  2.3474])
```

Метод `numpy.searchsorted`: поиск элементов в отсортированном массиве

Метод массива `searchsorted` производит двоичный поиск в отсортированном массиве и возвращает место, в которое нужно было бы вставить значение, чтобы массив оставался отсортированным:

```
In [204]: arr = np.array([0, 1, 7, 12, 15])

In [205]: arr.searchsorted(9)
Out[205]: 3
```

Можно передать также массив значений и получить в ответ массив индексов:

```
In [206]: arr.searchsorted([0, 8, 11, 16])
Out[206]: array([0, 3, 3, 5])
```

Вы, наверное, заметили, что `searchsorted` вернул индекс 0 для элемента 0. Это объясняется тем, что по умолчанию возвращается индекс самого левого из группы элементов с одинаковыми значениями:

```
In [207]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [208]: arr.searchsorted([0, 1])
Out[208]: array([0, 3])

In [209]: arr.searchsorted([0, 1], side='right')
Out[209]: array([3, 7])
```

Чтобы проиллюстрировать еще одно применение метода `searchsorted`, предположим, что имеется массив значений между 0 и 10 000 и отдельный массив «границ интервалов», который мы хотим использовать для распределения данных по интервалам:

```
In [210]: data = np.floor(rng.uniform(0, 10000, size=50))

In [211]: bins = np.array([0, 100, 1000, 5000, 10000])

In [212]: data
Out[212]:
array([ 815., 1598., 3401., 4651., 2664., 8157., 1932., 1294., 916.,
       5985., 8547., 6016., 9319., 7247., 8605., 9293., 5461., 9376.,
       4949., 2737., 4517., 6650., 3308., 9034., 2570., 3398., 2588.,
       3554., 50., 6286., 2823., 680., 6168., 1763., 3043., 4408.,
       1502., 2179., 4743., 4763., 2552., 2975., 2790., 2605., 4827.,
       2119., 4956., 2462., 8384., 1801.])
```

Чтобы теперь для каждой точки узнать, какому интервалу она принадлежит (считая, что 1 означает интервал `[0, 100)`), мы можем воспользоваться методом `searchsorted`:

```
In [213]: labels = bins.searchsorted(data)

In [214]: labels
Out[214]:
array([2, 3, 3, 3, 3, 4, 3, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 4,
       3, 4, 3, 3, 3, 3, 1, 4, 3, 2, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 4, 3])
```

В сочетании с методом `groupby` из библиотеки `pandas` этого достаточно, чтобы распределить данные по интервалам:

```
In [215]: pd.Series(data).groupby(labels).mean()
Out[215]:
1    50.000000
2    803.666667
3   3079.741935
4   7635.200000
dtype: float64
```

A.7. НАПИСАНИЕ БЫСТРЫХ ФУНКЦИЙ для NumPy с помощью Numba

Numba (<http://numba.pydata.org/>) – проект с открытым исходным кодом, предназначенный для создания быстрых функций для работы с данными NumPy и похожими на них с использованием CPU, GPU и другого оборудования. Для трансляции написанного на Python кода в машинные команды применяется проект LLVM (<http://llvm.org/>).

Чтобы составить представление о Numba, рассмотрим функцию на чистом Python, которая вычисляет выражение `(x - y).mean()` в цикле `for`:

```
import numpy as np

def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
```

```

for i in range(nx):
    result += x[i] - y[i]
    count += 1
return result / count

```

Эта функция работает медленно:

```
In [209]: x = rng.standard_normal(10_000_000)
```

```
In [210]: y = rng.standard_normal(10_000_000)
```

```
In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop
```

```
In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

Версия, встроенная в NumPy, быстрее в сто с лишним раз. Мы можем преобразовать написанную нами функцию в откомпилированную функцию Numba, воспользовавшись функцией `numba.jit`:

```
In [213]: import numba as nb
In [214]: numba_mean_distance = nb.jit(mean_distance)
```

Можно было бы оформить это и в виде декоратора:

```

@nb.jit
def numba_mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count

```

Получившаяся функция даже быстрее векторной версии из NumPy:

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba не умеет компилировать произвольный код на Python, но поддерживает обширное подмножество Python, наиболее полезное при реализации численных алгоритмов.

Numba – серьезная библиотека, поддерживающая различные виды оборудования, режимы компиляции и пользовательские расширения. Она способна откомпилировать значительное подмножество Python API библиотеки NumPy, не прибегая к явным циклам `for`. Кроме того, Numba умеет распознавать конструкции, допускающие встраивание на машинном коде, а если не знает, как откомпилировать код функции, то подставляет обращения к CPython API. У функции Numba `jit` имеется факультативный аргумент `nopython=True`, который разрешает использовать только такой код на Python, который можно транслировать на LLVM, не прибегая к вызовам Python C API. Вызов `jit(nopython=True)` имеет короткий псевдоним `numba.njit`.

Предыдущий пример можно было бы записать и так:

```
from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

Призываю вас ознакомиться с онлайн-документацией по Numba на сайте <http://numba.pydata.org/>. В следующем разделе приведен пример создания пользовательской u-функции для NumPy.

Создание пользовательских объектов `numpy.ufunc` с помощью Numba

Функция `numba.vectorize` создает откомпилированные u-функции NumPy, которые ведут себя так же, как встроенные. Рассмотрим реализацию `numpy.add` на Python:

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y

Имеем:
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([ 0.,  1.,  3.,  6., 10., 15., 21., 28., 36., 45.])
```

А.8. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О ВВОДЕ-ВЫВОДЕ МАССИВОВ

В главе 4 мы познакомились с методами `np.save` и `np.load` для хранения массивов в двоичном формате на диске. Но есть и целый ряд дополнительных возможностей на случай, когда нужно что-то более сложное. В частности, файлы, отображенные на память, позволяют работать с наборами данных, не уместяющимися в оперативной памяти.

Файлы, отображенные на память

Отображение файла на память – метод, позволяющий рассматривать потенциально очень большой набор данных на диске как массив в памяти. В NumPy объект `memmap` реализован по аналогии с `ndarray`, он позволяет читать и записывать небольшие сегменты большого файла, не загружая в память весь массив. Кроме того, у объекта `memmap` точно такие же методы, как у массива в памяти, поэтому его можно подставить во многие алгоритмы, ожидающие получить `ndarray`.

Для создания объекта `memmap` служит функция `np.memmap`, которой передается путь к файлу, `dtype`, форма и режим открытия файла:

```
In [217]: mmap = np.memmap('mymmap', dtype='float64', mode='w+',
.....: shape=(10000, 10000))
```

```
In [218]: mmap
Out[218]:
memmap([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```

При вырезании из `memmap` возвращается представление данных на диске:

```
In [219]: section = mmap[:5]
```

Если присвоить такому срезу значения, то они буферизуются в памяти, т. е. при попытке прочитать файл в другом приложении изменения могут быть не видны. Чтобы синхронизировать модификации с диском, воспользуйтесь методом `flush`.

```
In [220]: section[:] = rng.standard_normal((5, 10000))
```

```
In [221]: mmap.flush()
```

```
In [222]: mmap
Out[222]:
memmap([[ -0.9074, -1.0954,  0.0071, ...,  0.2753, -1.1641,  0.8521],
        [ -0.0103, -0.0646, -1.0615, ..., -1.1003,  0.2505,  0.5832],
        [  0.4583,  1.2992,  1.7137, ...,  0.8691, -0.7889, -0.2431],
        ...,
        [  0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
        [  0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
        [  0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ]])
```

```
In [223]: del mmap
```

Сброс на диск всех изменений автоматически происходит и тогда, когда объект `memmap` выходит из области видимости и передается сборщику мусора. При *открытии существующего файла* все равно необходимо указывать тип и форму данных, поскольку файл на диске – это просто блок двоичных данных, не содержащий информации о типе, форме и шагах:

```
In [224]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))
```

```
In [225]: mmap
Out[225]:
memmap([[ -0.9074, -1.0954,  0.0071, ...,  0.2753, -1.1641,  0.8521],
        [ -0.0103, -0.0646, -1.0615, ..., -1.1003,  0.2505,  0.5832],
        [  0.4583,  1.2992,  1.7137, ...,  0.8691, -0.7889, -0.2431],
        ...,
        [  0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
        [  0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
        [  0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ]])
```

Отображение на память работает также со структурными и вложенными типами dtype, описанными в предыдущем разделе.

После запуска этого примера на своем компьютере вы, наверное, захотите удалить созданный большой файл:

```
In [226]: %xdel mmap
```

```
In [227]: !rm mymmap
```

HDF5 и другие варианты хранения массива

PyTables и h5py – написанные на Python проекты, в которых реализован ориентированный на NumPy интерфейс для хранения массива в эффективном, допускающем сжатие формате HDF5 (HDF означает *hierarchical data format* – иерархический формат данных). В формате HDF5 можно без опаски хранить сотни гигабайтов и даже терабайты данных. Для получения дополнительных сведений о работе с HDF5 в Python обратитесь к документации по pandas.

А.9. ЗАМЕЧАНИЯ О ПРОИЗВОДИТЕЛЬНОСТИ

Адаптация кода обработки данных к NumPy обычно значительно ускоряет программу, поскольку операции над массивами, как правило, заменяют медленные по сравнению с ними циклы на чистом Python. Приведенные ниже советы позволят получить максимальную отдачу от использования библиотеки.

- Преобразуйте циклы и условную логику Python с операции с массивами и булевыми массивами.
- Всюду, где только можно, применяйте укладывание.
- Избегайте копирования данных с помощью представлений массивов (вырезание).
- Используйте и-функции и их методы.

Если с помощью одних лишь средств NumPy все же никак не удастся добиться требуемой производительности, то, возможно, имеет смысл написать часть кода на C, Fortran и особенно на Cython (подробнее об этом ниже). Лично я очень активно использую Cython (<http://cython.org>) в собственной работе как простой способ получить производительность, сравнимую с C, затратив минимум усилий.

Важность непрерывной памяти

Хотя полное рассмотрение заявленной темы выходит за рамки этой книги, в некоторых приложениях расположение массива в памяти может оказать существенное влияние на скорость вычислений. Отчасти это связано с иерархией процессорных кешей; операции, в которых осуществляется доступ к соседним адресам в памяти (например, суммирование по строкам в массиве, организованном как в C), обычно выполняются быстрее всего, потому что подсистема памяти буферизует соответствующие участки в кешах уровня L1 или L2 с низкой задержкой. Кроме того, некоторые ветви написанного на C кода NumPy оптимизированы для непрерывного случая, когда шагового доступа можно избежать.

Говоря о *непрерывной* организации памяти, мы имеем в виду, что элементы массива хранятся в памяти в том порядке, в котором видны в массиве, организованном по столбцам (как в Fortran) или по строкам (как в C). По умолчанию массивы в NumPy создаются *C-непрерывными*. О массиве, хранящемся по столбцам, например транспонированном C-непрерывном массиве, говорят, что он Fortran-непрерывный. Эти свойства можно явно опросить с помощью атрибута `flags` объекта `ndarray`:

```
In [228]: arr_c = np.ones((100, 10000), order='C')
```

```
In [229]: arr_f = np.ones((100, 10000), order='F')
```

```
In [230]: arr_c.flags
```

```
Out[230]:
```

```
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
Advanced NumPy | 505
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

```
In [231]: arr_f.flags
```

```
Out[231]:
```

```
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

```
In [232]: arr_f.flags.f_contiguous
```

```
Out[232]: True
```

В данном случае суммирование строк массива теоретически должно быть быстрее для `arr_c`, чем для `arr_f`, поскольку строки хранятся в памяти непрерывно. Я проверил это с помощью функции `%timeit` в IPython (на вашей машине результаты могут отличаться):

```
In [233]: %timeit arr_c.sum(1)
```

```
444 us +- 60.5 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

```
In [234]: %timeit arr_f.sum(1)
```

```
581 us +- 8.16 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

Часто именно в этом направлении имеет смысл прикладывать усилия, стремясь выжать всю возможную производительность из NumPy. Если массив размещен в памяти не так, как нужно, можно скопировать его методом `copy`, передав параметр `'C'` или `'F'`:

```
In [235]: arr_f.copy('C').flags
Out[235]:
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

При построении представления массива помните, что непрерывность результата не гарантируется:

```
In [236]: arr_c[:50].flags.contiguous
Out[236]: True

In [237]: arr_c[:, :50].flags
Out[237]:
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

Приложение В

Еще о системе IPython

В главе 2 мы рассмотрели основы оболочки IPython и Jupyter-блокнотов. В этом приложении мы поговорим о дополнительных возможностях IPython, которые можно использовать как из консоли, так и из Jupyter.

В.1. 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. В табл. В.1 перечислены наиболее употребительные комбинации, а на рис. В.1 некоторые из них, например перемещение курсора, проиллюстрированы.

Таблица 2.1. Стандартные комбинации клавиш IPython

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



Рис. В.1. Иллюстрация некоторых комбинаций клавиш IPython

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

В.2. О МАГИЧЕСКИХ КОМАНДАХ

В IPython есть много специальных команд, называемых «магическими», цель которых – упростить решение типичных задач и облегчить контроль над поведением всей системы IPython. Магической называется команда, которой предшествует знак процента %. Например, магическая функция `%timeit` позволяет замерить время выполнения любого предложения Python, а именно умножения матриц:

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

```
In [20]: %timeit np.dot(a, a)
```

```
92.5 µs ± 3.43 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

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?)¹³:

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

```
Docstring:
```

```
::
```

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

Активировать интерактивный отладчик.

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

Второй способ – активировать отладчик в постоперационном режиме. Для этого нужно просто выполнить команду `%debug` без аргументов. В случае исключения это позволит интерактивно просматривать кадры стека. Отметим,

¹³ Сообщения выводятся на английском языке, но для удобства читателя переведены. – Прим. перев.

что при этом всегда используется последняя трасса стека, так что анализировать ее надо сразу после возникновения исключения, поскольку следующее исключение затрет предыдущее.

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

`.. versionchanged:: 7.3`

При выполнении кода пользовательские переменные больше не расширяются, магическая строка никогда не модифицируется.

позиционные аргументы:

`statement` Код, подлежащий выполнению в отладчике. При работе в режиме ячейки можно опускать.

факультативные аргументы:

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

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

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

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

In [23]: foo = %pwd

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

Поскольку к документации по IPython легко можно обратиться из системы, я рекомендую изучить все имеющиеся специальные команды, набрав `%quickref` или `%magic`. Эта информация отображается консольным средством постраничного просмотра, для выхода из него нужно нажать клавишу `q`. В табл. В.2 описаны некоторые команды, наиболее важные для продуктивной работы в области интерактивных вычислений и разработки в среде IPython.

Таблица В.2. Часто используемые магические команды IPython

Команда	Описание
<code>%quickref</code>	Вывести краткую справку по IPython
<code>%magic</code>	Вывести подробную документацию по всем имеющимся магическим командам
<code>%debug</code>	Войти в интерактивный отладчик в точке последнего вызова, показанного в обратной трассировке исключения
<code>%hist</code>	Напечатать историю введенных команд (по желанию вместе с результатами)

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

Команда `%run`

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

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

```
a = 5
b = 6
c = 7.5
```

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

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

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

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

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

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

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



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

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

```
In [16]: %load script.py
```

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

```
a = 5
b = 6
c = 7.5
```

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

Прерывание выполняемой программы

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



Если Python-код вызвал откомпилированный модуль расширения, то нажатие `<Ctrl-C>` не всегда приводит к немедленному завершению. В таких случаях нужно либо дождаться возврата управления интерпретатору Python, либо – если случилось что-то ужасное – принудительно снять процесс Python средствами операционной системы (например, с помощью Диспетчера задач в Windows или команды `kill` в Linux).

Исполнение кода из буфера обмена

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

```
x = 5
y = 7
if x > 5:
    x += 1
    y = 8
```

Проще всего воспользоваться магическими функциями `%paste` и `%cpaste` (отметим, что они не работают в Jupyter, поскольку там в ячейке кода возможны операции копирования и вставки). Функция `%paste` берет текст, находящийся в буфере обмена, и выполняет его в оболочке как единый блок:

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

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

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

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

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

В.3. История команд

IPython хранит небольшую базу данных на диске, в которой находятся тексты всех выполненных команд. Она служит нескольким целям:

- поиск, автозавершение и повторное выполнение ранее выполненных команд с минимальными усилиями;
- сохранение истории команд между сеансами;
- протоколирование истории ввода-вывода в файле.

Эти средства больше полезны в оболочке, чем в блокноте, поскольку блокнот изначально хранит всю историю ввода-вывода в каждой ячейке.

Поиск в истории команд и повторное выполнение

IPython позволяет искать и повторно выполнять предыдущий код или другие команды. Это полезно, поскольку 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 выходные объекты хранятся соответственно в переменных `_` (один подчеркик) и `__` (два подчеркика):

```
In [18]: 'input1'
Out[18]: 'input1'

In [19]: 'input2'
Out[19]: 'input2'

In [20]: __
Out[20]: 'input1'

In [21]: 'input3'
Out[21]: 'input3'

In [22]: _
Out[22]: 'input3'
```

Входные команды хранятся в переменных с именами вида `_iX`, где `X` – номер входной строки. Каждой такой входной переменной соответствует выходная переменная `_X`. Поэтому после ввода строки 27 будут созданы две новые переменные: `_27` (для хранения выходного объекта) и `_i27` (для хранения входной команды).

```
In [26]: foo = 'bar'
```

```
In [27]: foo
Out[27]: 'bar'
```

```
In [28]: _i27
Out[28]: u'foo'
```

```
In [29]: _27
Out[29]: 'bar'
```

Поскольку входные переменные – это строки, то их можно повторно вычислить с помощью ключевого слова Python `eval`:

```
In [30]: eval(_i27)
Out[30]: 'bar'
```

Есть несколько магических функций, позволяющих работать с историей ввода и вывода. Функция `%hist` умеет показывать историю ввода полностью или частично, с номерами строк или без них. Функция `%reset` очищает интерактивное пространство имен и факультативно кеша ввода и вывода. Функция `%xdel` удаляет все ссылки на конкретный объект из внутренних структур данных IPython. Подробнее см. документацию по этим функциям.



Работая с очень большими наборами данных, имейте в виду, что объекты, хранящиеся в истории ввода-вывода IPython, не могут быть удалены из памяти сборщиком мусора – даже если вы удалите соответствующую переменную из интерактивного пространства имен встроенным оператором `del`. В таких случаях команды `%xdel` и `%reset` помогут избежать проблем с памятью.

В.4. ВЗАИМОДЕЙСТВИЕ С ОПЕРАЦИОННОЙ СИСТЕМОЙ

Еще одна особенность IPython – тесная интеграция с файловой системой и оболочкой операционной системы. Среди прочего это означает, что многие стандартные действия в командной строке можно выполнять в точности так же, как в оболочке Windows или UNIX (Linux, OS X), не выходя из IPython. Речь идет о выполнении команд оболочки, смене рабочего каталога и сохранении результатов команды в объекте Python (строке или списке). Существуют также простые средства для задания псевдонимов команд оболочки и создания закладок на каталоги.

Перечень магических функций и синтаксис вызова команд оболочки представлены в табл. В.3. В следующих разделах я кратко расскажу о них.

Таблица В.3. Команды IPython, относящиеся к операционной системе

Команда	Описание
<code>!cmd</code>	Выполнить команду в оболочке системы
<code>output = !cmd args</code>	Выполнить команду и сохранить в объекте <code>output</code> все выведенное на стандартный вывод
<code>%alias alias_name cmd</code>	Определить псевдоним команды оболочки
<code>%bookmark</code>	Воспользоваться системой закладок IPython
<code>%cd каталог</code>	Сделать указанный каталог рабочим
<code>%pwd</code>	Вернуть текущий рабочий каталог
<code>%pushd каталог</code>	Поместить текущий каталог в стек и перейти в указанный каталог
<code>%popd</code>	Извлечь каталог из стека и перейти в него
<code>%dirs</code>	Вернуть список, содержащий текущее состояние стека каталогов
<code>%dhist</code>	Напечатать историю посещения каталогов
<code>%env</code>	Вернуть переменные среды в виде словаря
<code>%matplotlib</code>	Задать параметры интеграции с matplotlib

Команды оболочки и псевдонимы

Восклицательный знак `!` в начале командной строки IPython означает, что все следующее за ним нужно выполнить в оболочке системы. Таким образом можно удалять файлы (командами `rm` или `del` в зависимости от ОС), изменять рабочий каталог или исполнять другой процесс.

Все, что команда выводит на консоль, можно сохранить в переменной, присвоив ей значение выражения, начинающегося со знака `!`. Например, на своей Linux-машине, подключенной к интернету Ethernet-кабелем, я могу следующим образом записать в переменную Python свой IP-адрес:

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11 Bcast:10.0.0.255 Mask:255.255.255.0'
```

Возвращенный объект Python `ip_info` – это специализированный список, содержащий различные варианты вывода на консоль.

IPython умеет также подставлять в команды, начинающиеся знаком `!`, значения переменных Python, определенных в текущем окружении. Для этого имени переменной нужно предпослать знак `$`:

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py test.py test.xml
```

Магическая функция `%alias` позволяет определять собственные сокращения для команд оболочки, например:

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x  2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x  2 root root   4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root  20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x  44 root root  69632 2011-12-26 18:08 lib32/
lrwxrwxrwx  1 root root    3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x  15 root root   4096 2011-10-13 19:03 local/
drwxr-xr-x  2 root root  12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root  12288 2011-11-04 22:53 share/
drwxrwsr-x  24 root src   4096 2011-07-17 18:38 src/
```

Несколько команд можно выполнить как одну, разделив их точками с запятой:

```
In [558]: %alias test_alias (cd examples; 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.

Система закладок на каталоги

В IPython имеется простая система закладок, позволяющая создавать псевдонимы часто используемых каталогов, чтобы упростить переход в них. Например, пусть требуется создать закладку, указывающую на дополнительные материалы к этой книге:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

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

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

Если имя закладки конфликтует с именем подкаталога вашего текущего рабочего каталога, то с помощью флага `-b` можно отдать приоритет закладке. Команда `%bookmark` с флагом `-l` выводит список всех закладок:

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

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

В.5. 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 удобные встроенные средства хронометража и профилирования. Ниже я расскажу об этих инструментах подробнее.

Интерактивный отладчик

Отладчик IPython дополняет `pdb` завершением по нажатии клавиши **Tab**, подсветкой синтаксиса и контекстом для каждой строки трассировки исключения. Отлаживать программу лучше всего немедленно после возникновения ошибки. Команда `%debug`, выполненная сразу после исключения, вызывает «посмертный» отладчик и переходит в то место стека вызовов, где было возбуждено исключение:

```
In [2]: run examples/ipython_bug.py
-----
AssertionError                                Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
    13     throws_an_exception()
    14
----> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
    11 def calling_things():
    12     works_fine()
----> 13     throws_an_exception()
    14
    15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
---->  9     assert(a + b == 10)
    10
    11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
      8     b = 6
---->  9     assert(a + b == 10)
    10
ipdb>
```

Находясь в отладчике, можно выполнять произвольный Python-код и просматривать все объекты и данные (которые интерпретатор «сохранил живыми») в каждом кадре стека. По умолчанию отладчик оказывается на самом нижнем уровне – там, где произошла ошибка. Клавиши **u** (вверх) и **d** (вниз) позволяют переходить с одного уровня стека на другой:

```

ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
12     works_fine()
--> 13     throws_an_exception()
14

```

Команда `%pdb` устанавливает режим, в котором IPython автоматически вызывает отладчик после любого исключения, многие считают этот режим особенно полезным.

Отладчик также помогает разрабатывать код, особенно когда хочется расставить точки останова либо пройти функцию или скрипт в пошаговом режиме, изучая состояние после каждого шага. Сделать это можно несколькими способами. Первый – воспользоваться функцией `%run` с флагом `-d`, которая вызывает отладчик, перед тем как начать выполнение кода в переданном скрипте. Для входа в скрипт нужно сразу же нажать `s` (step – пошаговый режим):

```

In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()
1----> 1 def works_fine():
2         a = 5
3         b = 6

```

После этого вы сами решаете, каким образом работать с файлом. Например, в приведенном выше примере исключения можно было бы поставить точку останова прямо перед вызовом метода `works_fine` и выполнить программу до этой точки, нажав `c` (continue – продолжить):

```

ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_things()
11 def calling_things():
2--> 12     works_fine()
13     throws_an_exception()

```

В этот момент можно войти внутрь `works_fine()` командой `step` или выполнить `works_fine()` без захода внутрь, т. е. перейти к следующей строке, нажав `n` (next – дальше):

```

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
2    12 works_fine()
--> 13 throws_an_exception()
14

```

Далее мы можем войти внутрь `throws_an_exception`, дойти до строки, где возникает ошибка, и изучить переменные в текущей области видимости. Отметим, что у команд отладчика больший приоритет, чем у имен переменных, поэтому для просмотра переменной с таким же именем, как у команды, необходимо предпослать ей знак `!`.

```

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(6)throws_an_exception()
5
----> 6 def throws_an_exception():
7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exception()
6 def throws_an_exception():
----> 7     a = 5
8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exception()
7     a = 5
----> 8     b = 6
9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
8     b = 6
----> 9     assert(a + b == 10)
10

ipdb> !a
5
ipdb> !b
6

```

Мой собственный опыт показывает, что для уверенного овладения интерактивным отладчиком нужно время и практика. В табл. В.4 приведен полный перечень команд отладчика. Если вы привыкли к IDE, то консольный отладчик на первых порах может показаться неуклюжим, но со временем это впечатление рассеется. В некоторых IDE для Python имеются отличные графические отладчики, так что всякий пользователь найдет что-то себе по вкусу.

Таблица В.4. Команды отладчика Python

Команда	Действие
<code>h(elp)</code>	Вывести список команд
<code>help</code> <i>команда</i>	Показать документацию по <i>команде</i>
<code>c(ontinue)</code>	Продолжить выполнение программы
<code>q(uit)</code>	Выйти из отладчика, прекратив выполнение кода
<code>b(reak)</code> <i>номер</i>	Поставить точку останова на строке с указанным <i>номером</i> в текущем файле
<code>b</code> <i>путь/к/файлу.ру:номер</i>	Поставить точку останова на строке с указанным <i>номером</i> в указанном файле
<code>s(step)</code>	Войти внутрь функции

Команда	Действие
<code>n(ext)</code>	Выполнить текущую строку и перейти к следующей на текущем уровне
<code>u(p) / d(own)</code>	Перемещение вверх и вниз по стеку вызовов
<code>a(rgs)</code>	Показать аргументы текущей функции
<code>debug предложение</code>	Выполнить <i>предложение</i> в новом (вложенном) отладчике
<code>l(ist) предложение</code>	Показать текущую позицию и контекст на текущем уровне стека
<code>w(HERE)</code>	Распечатать весь стек в контексте текущей позиции

Другие способы работы с отладчиком

Существует еще два полезных способа вызова отладчика. Первый – воспользоваться специальной функцией `set_trace` (названной так по аналогии с `pdb.set_trace`), которая по существу является упрощенным вариантом точки останова. Вот два небольших фрагмента, которые вы можете сохранить где-нибудь и использовать в разных программах (я, например, помещаю их в свой профиль IPython):

```
from IPython.core.debugger import Pdb

def set_trace():
    Pdb().set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb()
    return pdb.runcall(f, *args, **kwargs)
```

Первая функция, `set_trace`, совсем простая. Вызывайте ее в той точке кода, где хотели бы остановиться и оглядеться (например, прямо перед строкой, в которой происходит исключение):

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py(16)calling_things()
15     set_trace()
--> 16     throws_an_exception()
17
```

При нажатии `c` (продолжить) выполнение программы возобновится без каких-либо побочных эффектов.

Функция `debug` позволяет вызвать интерактивный отладчик в момент обращения к любой функции. Допустим, мы написали такую функцию и хотели бы пройти ее в пошаговом режиме:

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


Обычно `f` используется примерно так: `f(1, 2, z=3)`. А чтобы войти в эту функцию, передайте `f` в качестве первого аргумента функции `debug`, а затем ее позиционные и именованные аргументы:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
1 def f(x, y, z):
----> 2     tmp = x + y
3     return tmp / z
ipdb>
```

Мне эти две простенькие функции ежедневно экономят уйму времени.

Наконец, отладчик можно использовать в сочетании с функцией `%run`. Запустив скрипт командой `%run -d`, вы попадете прямо в отладчик и сможете расставить точки останова и начать выполнение:

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb>
```

Если добавить еще флаг `-b`, указав номер строки, то после входа в отладчик на этой строке уже будет стоять точка останова:

```
In [2]: %run -d -b2 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(2)works_fine()
1 def works_fine():
1---> 2     a = 5
3     b = 6
ipdb>
```

Хронометраж программы: `%time` и `%timeit`

Для больших или долго работающих аналитических приложений бывает желательно измерить время выполнения различных участков кода или даже отдельных предложений или вызовов функций. Интересно получить отчет о том, какие функции занимают больше всего времени в сложном процессе. По счастью, IPython позволяет без труда получить эту информацию по ходу разработки и тестирования программы.

Ручной хронометраж с помощью встроенного модуля `time` и его функций `time.clock` и `time.time` зачастую оказывается скучной и утомительной процедурой, поскольку приходится писать один и тот же трафаретный код:

```
import time
start = time.time()
for i in range(iterations):
    # здесь код, который требуется 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 600 000 строк, и два метода выборки тех, что начинаются с 'foo':

```
# очень длинный список строк
In [11]: strings = ['foo', 'foobar', 'baz', 'qux',
....:               'python', 'Guido Van Rossum'] * 100000

In [12]: method1 = [x for x in strings if x.startswith('foo')]

In [13]: 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 [14]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 52.5 ms, sys: 0 ns, total: 52.5 ms
Wall time: 52.1 ms

In [15]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 65.3 ms, sys: 0 ns, total: 65.3 ms
Wall time: 64.8 ms
```

Наибольший интерес представляет величина `Wall time` (фактическое время). Похоже, первый метод работает в два раза медленнее второго, но это не очень точное измерение. Если вы несколько раз сами замерите время работы этих двух предложений, то убедитесь, что результаты варьируются. Для более точного измерения воспользуемся магической функцией `%timeit`. Она получает произвольное предложение и, применяя внутренние эвристики, выполняет его столько раз, сколько необходимо для получения более точного среднего времени (на вашей машине результаты могут быть иными):

```
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: %run и %run -p

Профилирование кода тесно связано с хронометражем, только отвечает на вопрос, *где именно* тратится время. В Python основное средство профилирования – модуль `cProfile`, который предназначен отнюдь не только для IPython. `cProfile` исполняет программу или произвольный блок кода и следит за тем, сколько времени проведено в каждой функции.

Обычно `cProfile` запускают из командной строки, профилируют программу целиком и выводят агрегированные временные характеристики каждой функции. Пусть имеется простой скрипт, который выполняет в цикле какой-нибудь алгоритм линейной алгебры (скажем, вычисляет максимальное по абсолютной величине собственное значение для последовательности матриц размерности 100×100):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in range(niter):
        mat = np.random.standard_normal((K, K))
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print('Largest one we saw: {0}'.format(np.max(some_results)))
```

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
Largest one we saw: 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'}

```

Аналогично команда `%run -p -s cumulative cprof_example.py` дает тот же результат, что рассмотренный выше запуск из командной строки, только не приходится выходить из IPython.

В Jupyter-блокноте для профилирования целого блока кода можно использовать магическую команду `%%prun` (два знака `%`). Она открывает отдельное окно, в которое выводится профиль. Это полезно для быстрого ответа на вопросы типа «Почему этот блок так долго работает?».

Существуют и другие инструменты, которые помогают интерпретировать профиль при работе с IPython или Jupyter. Один из них – SnakeViz (<https://github.com/jiffyclub/snakeviz/>) – порождает интерактивную визуализацию результатов профилирования с помощью библиотеки D3.js.

Построчное профилирование функции

Иногда информации, полученной от `%prun` (или добытой иным способом профилирования на основе `cProfile`), недостаточно, чтобы составить полное представление о времени работы функции. Или она настолько сложна, что результаты, агрегированные по имени функции, с трудом поддаются интерпретации. На такой случай есть небольшая библиотека `line_profiler` (ее поможет установить PyPI или любой другой инструмент управления пакетами). Она содержит расширение IPython, включающее новую магическую функцию `%lprun`, которая строит построчный профиль выполнения одной или нескольких функций. Чтобы подключить это расширение, нужно модифицировать конфигурационный файл IPython (см. документацию по IPython или раздел, посвященный конфигурированию, ниже), добавив такую строку:

```
# Список имен загружаемых модулей с расширениями IPython.
c.InteractiveShellApp.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 (если хотите воспроизвести пример, поместите следующий ниже код в файл `prof_mod.py`):

```
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 мы задались целью оценить производительность функции `add_and_sum`, то команда `%prun` дала бы такие результаты:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
      4 function calls in 0.049 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      1   0.036   0.036   0.046   0.046  prof_mod.py:3(add_and_sum)
      1   0.009   0.009   0.009   0.009  {method 'sum' of 'numpy.ndarray'}
      1   0.003   0.003   0.049   0.049  <string>:1(<module>)
```

Не слишком полезно. Но после активации расширения IPython `line_profiler` становится доступна новая команда `%lprun`. От `%prun` она отличается только тем,

что мы указываем, какую функцию (или функции) хотим профилировать. Порядок вызова такой:

```
%lprun -f func1 -f func2 профилируемое_предложение
```

В данном случае мы хотим профилировать функцию `add_and_sum`, поэтому пишем:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
```

Line #	Hits	Time Per Hit	% Time	Line Contents
3				def add_and_sum(x, y):
4	1	36510	36510.0	79.5 added = x + y
5	1	9425	9425.0	20.5 summed = added.sum(axis=1)
6	1	1	1.0	0.0 return summed

Так гораздо понятнее. В этом примере мы профилировали ту же функцию, которая составляла предложение. Но можно было бы вызвать функцию `call_function` из показанного выше модуля и профилировать ее наряду с `add_and_sum`, это дало бы полную картину производительности кода:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: 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: 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	554	554.0	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.

В.6. СОВЕТЫ ПО ПРОДУКТИВНОЙ РАЗРАБОТКЕ КОДА С ИСПОЛЬЗОВАНИЕМ IPYTHON

Создание кода таким образом, чтобы его можно было разрабатывать, отлаживать и в конечном счете *использовать* интерактивно, многим может показаться сменой парадигмы. Придется несколько изменить подходы к таким процедурным деталям, как перезагрузка кода, а также сам стиль кодирования.

Поэтому реализация стратегий, описанных в этом разделе, – скорее искусство, чем наука, вы должны будете экспериментально определить наиболее эффективный для себя способ написания Python-кода. Конечная задача – структурировать код так, чтобы с ним было легко работать интерактивно и изучать результаты прогона всей программы или отдельной функции с наименьшими усилиями. Я пришел к выводу, что программу, спроектированную в расчете на IPython, использовать проще, чем аналогичную, но построенную как автономное командное приложение. Это становится особенно важно, когда возникает какая-то проблема и нужно найти ошибку в коде, написанном вами или кем-то еще несколько месяцев или лет назад.

Перезагрузка зависимостей модуля

Когда в Python-программе впервые встречается предложение `import some_lib`, выполняется код из модуля `some_lib` и все переменные, функции и импортированные модули сохраняются во вновь созданном пространстве имен модуля `some_lib`. При следующей обработке предложения `import some_lib` будет возвращена ссылка на уже существующее пространство имен модуля. При интерактивной разработке кода возникает проблема: как быть, когда, скажем, с помощью команды `%run` выполняется скрипт, зависящий от другого модуля, в который вы внесли изменения? Допустим, в файле `test_script.py` находится такой код:

```
import some_lib

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

Если выполнить `%run test_script.py`, а затем изменить `some_lib.py`, то при следующем выполнении `%run test_script.py` мы получим *старую версию some_lib.py* из-за принятого в Python механизма однократной загрузки. Такое поведение отличается от некоторых других сред анализа данных, например MATLAB, в которых изменения кода распространяются автоматически¹⁴. Справиться с этой проблемой можно двумя способами. Во-первых, использовать функцию `reload` из модуля `importlib` стандартной библиотеки:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

¹⁴ Поскольку модуль или пакет может импортироваться в нескольких местах программы, Python кеширует код модуля при первом импортировании, а не выполняет его каждый раз. В противном случае следование принципам модульности и правильной организации кода могло бы поставить под угрозу эффективность приложения.

При этом производится попытка предоставить новую копию *some_lib.py* при каждом запуске *test_script.py* (но не всегда такая попытка оказывается успешной). Очевидно, что если глубина вложенности зависимостей больше единицы, то вставлять *reload* повсюду становится утомительно. Поэтому в IPython имеется специальная функция *dreload* (не магическая), выполняющая «глубокую» (рекурсивную) перезагрузку модулей. Если в файле *some_lib.py* имеется предложение *dreload(some_lib)*, то интерпретатор постарается перезагрузить как модуль *some_lib*, так и все его зависимости. К сожалению, это работает не во всех случаях, но если работает, то оказывается куда лучше перезапуска всего IPython.

Советы по проектированию программ

Простых рецептов здесь нет, но некоторыми общими соображениями, которые лично мне кажутся эффективными, я все же поделюсь.

Сохраняйте ссылки на нужные объекты и данные

Программы, рассчитанные на запуск из командной строки, нередко структурируются, как показано в следующем тривиальном примере:

```
from my_functions import g

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

def main():
    x = 6
    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*. Это эквивалентно определению переменных верхнего уровня в ячейках Jupyter-блокнота.

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 «Дзен Python», применимая и к разработке кода, предназначенного для интерактивного использования. Чем более модульными являются классы и функции и чем меньше связей между ними, тем проще их тестировать (если вы пишете автономные тесты), отлаживать и использовать интерактивно.

Перестаньте бояться длинных файлов

Если вы раньше работали с Java (или аналогичным языком), то, наверное, вам говорили, что чем файл короче, тем лучше. Во многих языках это разумный совет; длинный файл несет в себе дурной «запах» 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 работа 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. помещать весь код в один гигантский файл. Для отыскания разумной и интуитивно очевидной структуры модулей и пакетов, составляющих большую программу, нередко приходится потрудиться, но при коллективной работе это очень важно. Каждый модуль должен обладать внутренней сцепленностью, а местонахождение функций и классов, относящихся к каждой области функциональности, должно быть как можно более очевидным.

В.7. Дополнительные возможности IPython

Если вы решите в полной мере задействовать систему IPython, то, наверное, станете писать код немного иначе или придется залезать в дебри конфигурационных файлов.

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`;
- включить расширения Jupyter;
- определить собственные магические функции или псевдонимы системных.

Конфигурационные параметры задаются в файле `ipython_config.py`, находящемся в подкаталоге `.ipython/` вашего домашнего каталога. Конфигурирование производится на основе конкретного *профиля*. При обычном запуске IPython загружается *профиль по умолчанию*, который хранится в каталоге

profile_default. Следовательно, в моей Linux-системе полный путь к конфигурационному файлу IPython по умолчанию будет таким:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

Для инициализации этого файла в своей системе выполните в терминале команду

```
ipython profile create default
```

Не стану останавливаться на технических деталях содержимого этого файла. По счастью, все параметры в нем подробно прокомментированы, так что оставляю их изучение и изменение читателю. Еще одна полезная возможность – поддержка сразу *нескольких профилей*. Допустим, имеется альтернативная конфигурация IPython для конкретного приложения или проекта. Чтобы создать новый профиль, нужно всего лишь ввести такую строку:

```
ipython profile create secret_project
```

Затем отредактируйте конфигурационные файлы во вновь созданном каталоге *profile_secret_project* и запустите IPython следующим образом:

```
$ ipython --profile=secret_project
Python 3.8.0 | packaged by conda-forge | (default, Nov 22 2019, 19:11:19)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
IPython profile: secret_project
```

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

Конфигурирование Jupyter устроено несколько иначе, потому что его блокноты можно использовать не только с Python, но и с другими языками. Чтобы создать аналогичный конфигурационный файл Jupyter, выполните команду:

```
jupyter notebook --generate-config
```

Она создаст конфигурационный файл *.jupyter/jupyter_notebook_config.py* по умолчанию в подкаталоге вашего начального каталога. Вы можете отредактировать его и переименовать, например:

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

Тогда при запуске Jupyter добавьте аргумент `--config`:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

В.8. ЗАКЛЮЧЕНИЕ

Когда будете прорабатывать примеры кода в этой книге и обогащать свои навыки программирования на Python, рекомендую постоянно интересоваться экосистемами IPython и Jupyter. Эти проекты создавались специально, чтобы повысить продуктивность пользователя, поэтому работать с ними проще, чем на самом языке Python с его вычислительными библиотеками.

А на сайте nbviewer (<https://nbviewer.jupyter.org/>) вы найдете много интересных Jupyter-блокнотов.

Предметный указатель

- %alias, команда 502
- %alias, магическая функция 503
- %automagic, магическая функция 496
- %bookmark, магическая функция 502
- %cpaste, магическая функция 499
- %debug команда 504
- %hist, магическая функция 496
- %lprun команда 512
- %lprun, магическая функция 513
- %magic, магическая функция 496
- %page, магическая функция 497
- %paste, магическая функция 497
- %pdb, магическая функция 505
- %prun 510
- %prun, магическая функция 497
- %quickref, магическая функция 496
- %reset, магическая функция 497
- %run команда 38, 497
 - с флагом -p 511
- %run, магическая функция 497
- %time() 509
- %timeit() 509
- %timeit, магическая функция 497
- %time, магическая функция 497
- %who_ls, магическая функция 497
- %whos, магическая функция 497
- %who, магическая функция 497
- %xdel, магическая функция 497

A

- агрегирование данных 326
 - agg() 327, 345
 - describe() 328, 333
 - возврат без индексов строк 332
 - методы GroupBy 326
 - пример групповой операции 318
 - методы массива NumPy 126
 - общие сведения 317
 - передискретизация OHLC 384
 - повышающая
 - передискретизация 384
 - понижающая
 - передискретизация 382

- применение нескольких функций 329
- применение функций, зависящих от столбца 328
- производительность 328
- сводные таблицы 347
- иерархическое индексирование 255
- перекрестная табуляция 350
- скользящие оконные функции 389
- арифметические операции 165
 - восполнение значений 166
 - между DataFrame и Series 168
- атрибуты
 - начинающиеся знаком подчеркивания 44
- атрибуты объектов Python 50

Б

- база данных Федеральной избирательной комиссии за 2012 год 451
- распределение суммы пожертвований по интервалам 457
- статистика пожертвований по роду занятий и месту работы 454
- статистика пожертвований по штатам 459
- базовые частоты 364
- базы данных, взаимодействие 209
- бета-распределение 120
- библиотеки
 - matplotlib 27
 - Numba 24
 - NumPy 25
 - pandas 25
 - pytz 369
 - scikit-learn 28
 - SciPy 28
 - statsmodels 29
 - соглашения об импорте 36
 - унаследованные 24
- бинарные операторы в Python 51

бинарные универсальные функции 122
 биномиальное распределение 120

В

векторизация 107
 определение 123
 векторные строковые функции 240
 визуализация 285
 matplotlib. См. matplotlib
 seaborn. См. seaborn
 другие средства 315
 вложенные типы данных 481
 восполнение отсутствующих данных 216
 временные метки 352, 356
 нормализованные 364
 временные ряды
 диапазоны дат, частоты и сдвиг 361
 генерирование диапазонов дат 362
 неделя месяца 365
 сдвиг данных 366
 смещения даты 364
 длинный и широкий формат 279
 индексирование 358
 неуникальные индексы 360
 общие сведения 352
 основы 356
 передискретизация и
 преобразование частоты 380
 групповая передискретизация
 по времени 387
 передискретизация OHLC 384
 повышающая
 передискретизация 384
 понижающая
 передискретизация 382
 периоды 373
 квартальные 376
 преобразование временных меток
 в 377
 создание PeriodIndex из
 массивов 379
 процентное изменение цен 180
 скользящие оконные функции. См.
 скользящие оконные функции
 типы данных 353
 преобразования 354
 форматирование, зависящее
 от локали 356

 фиксированная частота 352
 часовые пояса. См. часовые пояса
 выбросы, фильтрация 226
 выравнивание данных 165
 восполнение значений
 в арифметических методах 166
 операции между DataFrame
 и Series 168
 вырезание
 в массивах 108
 в списках 71

Г

гамма-распределение 120
 генераторы 87
 reversed() как 80
 генераторные выражения 88
 модуль itertools 89
 гистограммы 308
 глобальная блокировка
 интерпретатора (GIL) 25
 глобальные переменные
 область видимости функции 84
 предостережение 84
 графики плотности 308
 группировка
 база данных Федеральной
 избирательной комиссии за 2012
 год 451
 распределение суммы
 пожертвований по
 интервалам 457
 статистика пожертвований по роду
 занятий и месту работы 454
 статистика пожертвований по
 штатам 459
 групповое взвешенное среднее 341
 квантильный анализ 335
 линейная регрессия 343
 метод apply 332
 случайная выборка 339
 группировка по интервалам 223,
 335, 486
 групповые ключи 334
 групповые операции 317
 transform() 343
 перекрестная табуляция 350
 подавление групповых ключей 334
 примеры
 групповая линейная регрессия 343

- групповое взвешенное среднее и корреляция 341
- подстановка вместо отсутствующих данных 337
- случайная выборка и перестановка 339
- развернутые 346
- разделение–применение–объединение 318, 332

Д

- данные
 - загрузка 186
 - конструирование признаков 397
 - множества 76
 - наборы данных на GitHub 35
 - отсутствующие. См. отсутствующие данные
 - подготовка. См. подготовка данных
 - списки 68
- дата и время
 - диапазоны дат 362
 - типы данных 354
- _ (два знака подчеркивания) 500
- двоеточие 46
- двоичные форматы данных 203
 - Apache Parquet 204
 - HDF5 205, 491
 - Microsoft Excel 204
 - pickle 203
 - сохранение ndarray 131
 - файлы, отображенные на память 489
 - сохранение графиков 297
 - хранение массивов 131
- диаграмма рассеяния 310
- диаграммы размаха 314
- дискретизация 223

З

- закрытые атрибуты 44
- закрытые методы 44
- запись в файл
 - двоичных данных
 - сохранение ndarray 131
 - сохранение графиков 297
 - файлы, отображенные на память 489
 - формат Excel 205
 - формат HDF5 491
 - формат pickle 203
 - текстовых данных

- другие форматы
 - с разделителями 198
- отсутствующие значения 195
- подмножество столбцов 196
- формат CSV 195
- формат JSON 199
- _ (знак подчеркивания) 44, 500
- #, знак решетки 47

И

- иерархическое индексирование 254
 - в pandas
 - сводная статистика по уровню 258
 - столбцами DataFrame 258
 - уровни сортировки 257
 - изменение формы 276
 - изменение формы данных 276
 - имена уровней 256
 - количество уровней 257
 - переупорядочение и уровни сортировки 257
 - сводная статистика по уровню 258
 - частичный индекс 255
 - чтение данных из файла 189
- изменение формы
 - массива 464
 - определение 276
 - с помощью иерархического индексирования 276
- изменяемые объекты 53
- индексы
 - в pandas 175
 - в классе TimeSeries 357
 - массивов 108
 - определение 138
- индикаторные переменные 229
- интегрированные среды разработки (IDE) 33
- интервалы
 - временные 352
 - открытые и закрытые 224
- исключения
 - автоматический вход в отладчик 496
- исполнение кода из буфера
 - обмена 498
- история команд в IPython 499
- итератора протокол 50
- итераторы
 - list() 69
 - range() 63

read_csv() 192
 tuple() 65
 генераторы 87
 использование утиной типизации
 для проверки 50
 объект groupby() 322
 по ключам и значениям 74
 распаковка в цикле for 62

К

категориальные данные
 Categorical расширенный тип 245
 cut() и groupby() 335
 моделирование нечислового
 столбца 399
 создание из последовательности 247
 вычисления 248
 повышение производительности 249
 коды категорий 245
 методы 250
 общие сведения 244
 осмысленный порядок 247
 фасетные сетки 313
 формулы Patsy 404
 [] (квадратные скобки) 66
 квантильный анализ 335
 коллекции
 встроенные функции
 последовательностей 78
 множества 76
 словари 72
 списковое включение 80
 команды. См. также магические
 команды
 отладчика 506
 поиск 494
 команды оболочки и их
 псевдонимы 501
 комбинирование
 списков 70
 комментарии в Python 47
 контейнер однородных данных 480
 конференции 33
 концы интервалов 383
 координированное универсальное
 время (UTC) 369
 корреляция и ковариация в pandas 180
 групповое взвешенное среднее и
 корреляция 341
 функции бинарного скользящего
 окна 394

кортежи 65
 в результатах SQL-запроса 210
 методы 68
 неизменяемость 66
 распаковка 67
 синтаксис rest 68
 типов исключений 91

Л

лексикографическая сортировка
 определение 484
 линеаризация 465
 линейная регрессия 343
 линейные графики 299
 линейные модели
 Patsy. См. Patsy
 в библиотеке statsmodels 406
 групповая линейная регрессия 343
 регрессия методом обыкновенных
 наименьших квадратов 408
 свободный член 401, 404, 407
 лямбда-функции 87, 240, 330

М

магические команды 495
 магические методы 44
 манипулирование данными
 изменение формы 276
 маргиналы 347
 маркерное значение 195, 212
 NA и NULL 190
 маркеры 291
 массивы
 NumPy ndarray 101
 PandasArray 138
 pandas Series 26, 138
 ассоциативные 72
 булево индексирование 113
 булевы 128
 в NumPy 463
 изменение формы 464
 конкатенация 466
 получение и установка
 подмножеств 470
 разбиение 466
 размещение в памяти 465
 сохранение в файле 489
 вырезание 108
 дат 355
 индексы 108

логические условия как операции
 с массивами 125
 операции между 107
 перестановка осей 117
 поиск в отсортированном
 массиве 486
 прихотливое индексирование 116
 создание PeriodIndex 379
 структурные
 вложенные типы данных 481
 достоинства 482
 структурные ndarray 480
 типы данных 104
 установка элементов с помощью
 укладывания 476
 устранение дубликатов 130
 файловый ввод-вывод 130
 хранение массивов на диске
 в двоичном формате 131
 шаговое представление ndarray 461
 математические операторы в
 Python 51
 матрица плана 400
 множества
 методы pandas DataFrame 150
 множественное включение 80
 объединение 76
 пересечение 77
 модули
 csv 196
 datetime 59
 itertools 89
 os 207
 pickle 203
 random 119
 re 237
 requests 208
 импорт 51
 момент первого пересечения 134
 мультииндекс 254
 создание и повторное
 использование 257

Н

набор данных о частоте детских имен
 в США
 анализ тенденций в выборе имен 437
 имена и пол 444
 рост разнообразия 438
 революция последней буквы 441

неделя месяца 365
 непрерывная память 491
 нормализованный набор временных
 меток 364
 нормальное распределение 120, 136

О

объектная модель 47
 объекты Python
 атрибуты 50
 изменяемые и неизменяемые 53
 интроспекция в IPython 45
 методы 47, 50
 общие сведения 47
 оператор is 52
 переменные 47
 преобразование в строку 55
 скалярные 53
 утиная типизация 50
 функции 85
 Олсона база данных о часовых
 поясах 369
 описательные статистики 177
 оси
 метки 293
 переименование индексов 222
 перестановка 117
 отображение файла на память 489
 отсутствующие данные 212, 275
 fillna(), восполнение 214, 216
 groupby() по групповому ключу 321
 isna() и notna() 140, 214
 восполнение 216
 сводные таблицы 349
 средним значением 337
 маркеры
 NaN (pandas) 140, 212
 NA и NULL 190
 появляющиеся при сдвиге 366
 разбор даты и времени 356
 фильтрация 214
 очистка экрана, комбинация
 клавиш 494

П

передискретизация и преобразование
 частоты 380
 групповая передискретизация по
 времени 387
 передискретизация OHLC 384

- повышающая
 - передискретизация 384
 - понижающая передискретизация 382
 - перекрестный контроль при обучении
 - модели 413
 - переменные среды 502
 - перестановки 227
 - периоды 373
 - определение 352
 - передискретизация 386
 - преобразование частоты 374
 - подготовка данных
 - категориальные данные. См. категориальные данные
 - комбинирование и слияние наборов данных 260
 - комбинирование перекрывающихся данных 274
 - конкатенация вдоль оси 269
 - манипуляции со строками 235
 - встроенные методы строковых объектов 235
 - регулярные выражения 237
 - строковые функции в pandas 240
 - общие сведения 212
 - отсутствующие данные. См. отсутствующие данные
 - преобразование данных
 - агрегирование 326
 - дискретизация и группировка по интервалам 223, 335, 486
 - замена значений 221
 - индикаторные переменные 229
 - обнаружение и фильтрация выбросов 226
 - обнаружение, случайная выборка 228
 - перестановка 227
 - с помощью функции или отображения 220
 - устранение дубликатов 218
 - расширение типов данных 232, 241
 - соединение 260
 - по индексу 265
 - подсчет
 - вхождений подстроки 236
 - количества знаков новой строки 55
 - метод кортежа count() 68
 - отличных от null значений в группе 321
 - понижающая передискретизация 380
 - порядок по строкам и по столбцам 465
 - последовательности
 - range() 63
 - встроенные функции 78
 - кортежи 65
 - распаковка 67
 - создание из объектов
 - Categorical 247
 - создание словаря из 74
 - списки 68
 - строки как 55
 - посторонний столбец 321
 - поток управления
 - предложение if 61
 - поток управления в Python 61
 - представления 108, 147
 - преобразование
 - между строкой и datetime 354
 - преобразование данных
 - индексы осей, метод map() 222
 - прерывание программы 494, 498
 - приведение типов 58
 - ndarray 106
 - ValueError 106
 - привязанные смещения в частотах 365
 - сдвиг дат 367
 - >>> приглашение 38
 - признак конца строки (EOL) 93
 - примеры наборов данных
 - Bitly 415
 - MovieLens 424
 - о частоте детских имен в США 432
 - Федеральной избирательной комиссии за 2012 год 451
 - прихотливое индексирование 116, 470
 - пространства имен 83
 - локальное 83
 - профили в IPython 516
 - пустое пространство имен 497
- Р**
- рабочий каталог 502
 - разбор текстовых файлов 186
 - HTML 200
 - JSON 199
 - XML с помощью lxml.objectify 201
 - разделение–применение–объединение 318
 - ранжирование данных 172

расширение типов данных 232, 241
 `astype()` 234, 246
регулярные выражения 237
 выделение разделителей в текстовом файле 190
редукция 177

С

C/C#/C++ языки

 Cython 491
 библиотеки HDF5 С 205
 замечание о производительности 491
 массивы NumPy 25, 105
 унаследованные библиотеки 24

сводные статистики 177

`isin`, метод 182
 корреляция и ковариация 180
 по уровню 258
сводные таблицы 347
 `mean()`, агрегирование
 по умолчанию 348
 восполнение отсутствующих
 комбинаций 349
 иерархическое индексирование 255
 маргиналы 347
 перекрестная табуляция 350
 поворот данных 276

связывание переменных 48

сериализация данных 203

скалярные типы 53

скользящие оконные функции 389

`rolling`, оператор 389
 коэффициент затухания 392
 промежуток 392
 среднее с расширяющимся
 окном 391

словари 72

`defaultdict()` 75
 `get()` получение ответа на HTTP-запрос 208
 `setdefault()` 75
 группировка с помощью 324
 допустимые типы ключей 75
 методы `keys()` и `values()` 74
 объединение 74
 объект `Series` как 139
 словарное включение 80
 создание из последовательности 74
случайная выборка 228
 пример 339

случайное блуждание 133

 сразу несколько 135

смещение даты 364

 периоды 373

соединение данных 260

 внешнее 261

 внутреннее 261

 левое и правое 262

 метод `join()` 268

 пример использования 229

 по ключу в индексе 265

 при наличии столбцов с
 одинаковыми именами 264

 типа многие к одному 260

 типа многие ко многим 262

сортировка 128, 482

 в `pandas` 172

 косвенная 483

 на месте 71

 поиск в отсортированном
 массиве 486

 проблема сортировки в порядке
 убывания 483

 уровни 257

 частичная 485

списки 68

`range()` 63

`sorted()`, получение нового
 отсортированного списка 79

`sort()`, сортировка на месте 71

 в качестве ключа словаря 76

 вырезание 71

 добавление и удаление элементов 69

`append()` и `insert()` 69

 изменяемость 68

 конкатенация и
 комбинирование списков 70

 присваивание столбцу

`DataFrame` 146

 производительность по сравнению
 с `ndarray` 100

 списковое включение 80

 вложенное 81

 строки как списки 55

списковое включение 80

 вложенное 81

средства разработки программ. См.

 IPython, средства разработки

ссылки 47

столбчатые диаграммы 302
 строго типизированные языки 49
 структурные массивы 480
 вложенные типы данных 481
 достоинства 482

Т

таблицы измерений 244
 текстовые файлы
 вывод данных 195
 данные в формате JSON 198
 формат с разделителями 196
 чтение порциями 193

типы данных
 в NumPy 462
 в Python 53
 булев 58
 строки 54
 числовые 54
 для массивов 104
 точечная диаграмма 310
 точка останова 507

У

укладывание 471
 задание элементов массива 475
 замечание о производительности 491
 общие сведения 169, 471
 определение 108
 по другим осям 474
 правило 472
 установка элементов массива 476
 унарные универсальные функции 122
 унарные функции 120
 уровни
 определение 254
 сводная статистика 258
 устойчивые алгоритмы
 сортировки 485
 утиная типизация 50

Ф

фасетные сетки, 313
 фильтрация
 в pandas 155
 финансовый год 374
 форма 461
 функции 82
 return 82
 yield вместо return 88
 анонимные (лямбда-) 87

аргументы 47
 None в качестве значения по
 умолчанию 59
 динамические ссылки, строгие
 типы 49
 именованные 83
 позиционные 83
 функции как 87
 атрибут `_name_` 330
 возврат нескольких значений 84
 вызов 47
 генераторы 87
 генераторные выражения 88
 группировка с помощью 325
 агрегирование данных 328
 как объекты 85
 методы объектов Python 47, 50
 объявление с помощью `def` 82
 построчное профилирование 512
 пространства имен 83

Х

хешируемость 76
 хи-квадрат распределение 120

Ч

часовые пояса 369
 UTC 369
 библиотека `pytz` 369
 летнее время 369, 372
 локализация и преобразование 369
 набор данных Bitly
 подсчет часовых поясов на чистом
 Python 416
 подсчет часовых поясов с помощью
 pandas 418
 операции над датами из разных
 часовых поясов 372
 операции с учетом поясного
 времени 371
 частоты
 преобразование 374

Ш

шаговое представление 461
 экспоненциально взвешенные
 функции 392

Я

ядерная оценка плотности (KDE) 309
 ядра 309

А

accumulate() (u-функция NumPy) 478
 add_constant (statsmodels) 407
 add_subplot() (matplotlib) 287
 add() (u-функция NumPy) 120
 add() (множества) 77
 aggregate() или agg() 327, 345
 Altair библиотека визуализации 315
 and, ключевое слово 58, 61
 annotate() (matplotlib) 295
 Apache Parquet 204
 read_parquet() 204
 удаленные серверы для обработки
 данных 207
 apply() (GroupBy) 332
 apply() (функции скользящего окна) 395
 arange() (NumPy) 100, 104
 argsort() (NumPy) 483
 array() (NumPy) 102
 arrow() (matplotlib) 295
 asfreq() 374, 385
 astype() для расширенных типов
 данных 234, 246
 AxesSubplot, объект 288

В

Bitly набор данных 415
 подсчет часовых поясов на чистом
 Python 416
 подсчет часовых поясов с помощью
 pandas 418
 Bokeh библиотека визуализации 315
 BooleanDtype расширенный тип
 данных 234
 break, ключевое слово
 в циклах for 62
 в циклах while 62
 build_design_matrices() (Patsy) 403

С

calendar, модуль 353
 Categorical, объект 224
 Categorical расширенный тип
 данных. См. категориальные
 данные
 catplot() (seaborn) 313
 cat, команда Unix 188
 chain() (itertools) 90
 Circle() (matplotlib) 297
 clear() (множества) 77

clock() 508
 closed свойство 96
 close(), закрытие файла 93
 collections, модуль 75
 combinations() (itertools) 90
 combine_first() 275
 concatenate() для ndarray 269, 466
 объекты r_ и c_ 468
 concat() (pandas) 270
 conda
 активация 31
 о Miniconda 29
 обновление пакетов 32
 установка пакетов 32
 continue, ключевое слово 62
 corr() 180, 342, 394
 cov() 181
 cProfile модуль 510
 crosstab() 351
 вычисление таблицы частот 305
 cross_val_score() (scikit-learn) 413
 csv модуль
 Dialect 197
 импорт 196
 открытие файла с односимвольным
 разделителем 196
 CSV-файлы
 запись 195
 другие форматы с разделителями 198
 чтение
 read_csv() 188
 аргументы read_csv() 192
 другие форматы с разделителями 196
 определение формата и
 разделителя 197
 порциями 193
 cut(), распределение данных по
 интервалам 223
 qcut() и квантили 225
 в сочетании с groupby 335
 Cython, производительность,
 сравнимая с C 491

D

DataFrame (pandas) 142
 applymap() поэлементное
 применение функции 171
 apply() функция 171
 get_dummies() 229
 head() и tail() 143

- to_numpy() 148
 - unique и другие теоретико-множественные операции 181
 - варианты доступа по индексу 161
 - выборка с помощью loc и iloc 159
 - загрузка результатов SQL-запроса 210
 - и u-функции NumPy 170
 - иерархическое
 - индексирование 142, 255
 - изменение формы 255, 276
 - переупорядочение и уровни сортировки 257
 - сводная статистика по уровню 258
 - столбцами 258
 - извлечение столбца как Series 144
 - импорт в локальное пространство имен 137
 - импорт и экспорт в формате JSON 199
 - индексные объекты 149
 - индексы по осям с повторяющимися значениями 175
 - индексы по строкам и по столбцам 142
 - конкатенация вдоль оси 269
 - конструирование 142
 - аргументы конструктора 148
 - обнаружение и фильтрация выбросов 226
 - общие сведения 26
 - объекты с различными индексами 165
 - операции 165
 - отображение в Jupyter-блокноте 143
 - отсутствующие данные 212
 - переиндексация 151
 - подвохи целочисленного индексирования 162
 - подвохи цепного индексирования 163
 - преобразование столбцов в категориальную форму 246
 - ранжирование 174
 - сводные статистики 177
 - сортировка 172
 - список арифметических методов 168
 - статистические методы
 - корреляция и ковариация 180
 - удаление элементов из оси 154
 - устранение дубликатов 218
 - формат HDF5 205
 - date_range() 362
 - параметр normalize 364
 - DatetimeIndex 357
 - DatetimeTZDtype расширенный тип данных 234
 - datetime тип 59, 354
 - неизменяемость 60
 - основы временных рядов 356
 - преобразование между строкой и datetime 354
 - разбор формата ISO8601 355
 - сдвиг дат с помощью смещений 367
 - спецификаторы, зависящие от локали 356
 - форматирование в виде строки 60
 - date тип 59, 354
 - debug() вызов отладчика 507
 - def, ключевое слово 82
 - del, ключевое слово 73, 146
 - dict() 74
 - difference_update() (множества) 77
 - difference() метод Index 150
 - difference() (множества) 77
 - dmatrices() (Patsy) 400
 - dot() (NumPy) умножение матриц 131
 - drop_duplicates() удаление строк-дубликатов 219
 - dropna() фильтрация отсутствующих строк 214
 - dtype свойство 102, 104, 461
 - uplicated() проверка строк DataFrame на дубликаты 218
- Е**
- enumerate() 79
 - eval, ключевое слово 501
 - ewm() экспоненциально взвешенный сдвиг 393
 - ExcelFile, класс (pandas) 204
 - except, блок 90
 - exit, команда 38
 - expanding() функция скользящего окна 391
 - exp() (u-функция NumPy) 120
 - extract() получение запомненных групп 242
- F**
- figure() (matplotlib) 287
 - Figure, объект 287, 290

fillna() восполнение отсутствующих
 данных 216, 221
 аргументы 218
 передискретизация 386
 средним значением 337
 find() поиск в строке 236
 Float32Dtype расширенный тип
 данных 234
 Float64Dtype расширенный тип
 данных 234
 float, тип данных 54, 462
 flush() сброс буфера ввода-вывода 96
 FORTRAN язык
 замечание о производительности 491
 массивы NumPy 99, 105
 унаследованные библиотеки 23
 for, циклы 61, 81, 107
 from_codes() для Categorical 247
 fromfile() (NumPy) 482
 frompyfunc() (NumPy) 480
 f-строки 57

G

getattr() 50
 getdefaultencoding() (модуль sys) 94
 get_dummies() 229, 399
 groupby() (itertools) 89
 apply() 332
 transform() 343
 в сочетании с cut() и qcut() 335
 группировка size() 321
 группировка по ключу 344
 задание уровня 257
 исключение посторонних
 столбцов 321
 и смещение дат 368
 методы агрегирования 326
 обход групп 322
 объект GroupBy 319
 индексирование именами
 столбцов 323
 подавление групповых ключей 334
 разделение–применение–
 объединение 318, 332

H

hasattr() 50
 HDF5 двоичный формат данных 205, 491
 HDFStore() 206
 hist() (matplotlib) 288
 histplot() (seaborn) 310

Hour() 364
 how, аргумент 384
 hstack() (NumPy) 467

I

if, предложение 61
 iloc оператор
 индексирование DataFrame 159
 индексирование Series 157
 квадратные скобки ([]) 158
 import, директива
 в Python 51
 использование в этой книге 36
 Index объекты (pandas) 149
 метод map() 222
 методы и свойства 150
 index() поиск в строке 236
 insert метод 69
 Int8Dtype расширенный тип
 данных 234
 Int16Dtype расширенный тип
 данных 234
 Int32Dtype расширенный тип
 данных 234
 Int64Dtype расширенный тип
 данных 234
 intersection_update() 77
 intersection() метод Index 150
 intersection() (множества) 77
 IPython
 документация 44
 дополнительные возможности 516
 доступ к DataFrame 145
 завершение по нажатию клавиши
 Tab 43
 запуск 39
 интроспекция 45
 исключения 92
 исполнение кода из буфера
 обмена 498
 история команд 499
 входные и выходные переменные 500
 поиск и повторное выполнение 500
 команды операционной системы 501
 закладки на каталоги 503
 команды оболочки и
 псевдонимы 502
 комбинации клавиш 494
 конфигурирования 516
 краткая справка 496

- магические команды 495
- %alias 502
- %bookmark 502
- %debug 504
- %lprun 512
- %prun 510
- %run 38, 497
- общие сведения 27
- основы 39
- перезагрузка зависимостей
 - модуля 514
- профили 516
- советы по проектированию 515
- средства разработки 504
- отладчик 504
- построчное профилирование 512
- профилирование 510
- советы на тему продуктивности 514
- хронометраж 508
- ipython_config.py, файл 516
- isdisjoint() 77
- isinstance() 49
- isna() 140, 213, 214
- issubdtype() (NumPy) 462
- issubset() 77
- issuperset() 77
- is_unique() свойство индексов 176
- is оператор 52
- J**
- JSON (JavaScript Object Notation) 198, 415, 446
- Julia язык программирования 24
- Jupyter
 - блокноты в GitHub 28
 - исполнение кода из буфера обмена 498
 - как среда разработки 33
 - конфигурирование 517
 - общие сведения 27
 - основы 40
 - отображение объекта DataFrame 143
 - построение графиков и визуализация 285
- Jupyter-блокнот
 - нюансы построения графиков 287
- K**
- KeyboardInterrupt 498
- Komodo IDE 33
- L**
- label, аргумент 383
- legend() (matplotlib) 292
- lexsort() (NumPy) 483
- linalg, модуль 132
- line_profiler, расширение 512
- list() 68
- LLVM проект 487
- load() (NumPy) 131, 489
- loc оператор
 - индексирование DataFrame 159
 - индексирование Series 156
 - квадратные скобки [] 156, 158
- lxml, библиотека 200
- M**
- map(), преобразование данных 221
 - индексы осей 222
- matplotlib
 - введение в API 286
 - аннотирование и рисование на подграфике 295
 - риски, метки и надписи 292
 - рисунки и подграфика 287
 - сохранение графиков в файле 297
 - цвета, маркеры и стили линий 291
 - конфигурирование 298
 - общие сведения 27, 285
 - онлайнная документация 289
 - патчи 297
 - пояснительные надписи 294
- matplotlibrc, файл 298
- maximum() (u-функция NumPy) 120
- mean() 319
 - агрегирование сводной таблицы по умолчанию 348
 - группировка по ключу 344
 - замена отсутствующих данных средним 337
- melt(), объединение нескольких столбцов в один 282
- memmap() (NumPy) 489
- mergesort', алгоритм сортировки 485
- merge() слияние наборов данных 260
- meshgrid() (NumPy) 123
- Microsoft Excel файлы 204
- Miniconda менеджер пакетов 29
- modf() (u-функция NumPy) 121
- MovieLens, пример набора данных 424
 - измерение несогласия в оценках 428

N

- NaN (не число) 127, 140, 212
 - dropna() отбрасывание отсутствующих данных 214
 - fillna() восполнение отсутствующих данных 216
 - isna() и notna() обнаружение отсутствующих данных 214
 - отсутствующие данные при чтении файла 190
- NaT (не время) 356
- ndarray (NumPy) 118
 - dtype свойство 102, 104
 - u-функции. См. u-функции (универсальные функции) для массивов ndarray
 - арифметические операции 107
 - булево индексирование 113
 - индексирование и вырезание 108
 - @ инфиксный оператор 118
 - конкатенация массивов 269, 466
 - линейная алгебра 131
 - методы булевых массивов 128
 - общие сведения 101
 - объекты Patsy DesignMatrix 401
 - перестановка осей 117
 - порядок C и порядок Fortran 466
 - прихотливое индексирование 116
 - take() и put() 470
 - программирование на основе массивов 123
 - векторизация 123
 - логические условия как операции с массивами 125
 - случайное блуждание 133
 - сразу несколько 135
 - устранение дубликатов и другие теоретико-множественные операции 130
 - продвинутые возможности u-функции 122, 477
 - внутреннее устройство объекта 461
 - замощение массива 469
 - иерархия типов данных 462
 - изменение формы массива 464
 - информация о шаге 461
 - компиляция u-функций с помощью Numba 488
 - написание u-функций на Python 479
 - объекты r_ и c_ 468
 - повтор элементов 469
 - разбиение массива 466
 - размещение по строкам и по столбцам 465
 - сортировка 128, 482
 - укладывание 108, 169, 471
 - эквиваленты прихотливого индексирования 470
 - свойство shape 102
 - создание 102
 - сортировка 128, 482
 - альтернативные алгоритмы 485
 - косвенная 483
 - поиск в отсортированном массиве 486
 - проблема сортировки в порядке убывания 483
 - частичная 485
 - сравнение производительности с чистым Python 100
 - статистические операции 126
 - структурные массивы. См. структурные массивы
 - типы данных 104, 461
 - ValueError 106
 - иерархия 462
 - приведение типов 106
 - транспонирование 117
 - укладывание. См. укладывание
- None 54, 59, 213
 - отсутствие ключа в словаре 75
 - проверка с помощью оператора is 53
 - функция без return 82
- notna() проверка на NaN 140, 214
- npz, расширение имени файла 131
- Numba
 - JIT-компиляция 24
 - общие сведения 487
 - пользовательские u-функции NumPy 489
- NumPy
 - frompyfunc() создание u-функций 479
 - ndarray. См. ndarray
 - to_numpy() метод DataFrame 148
 - генерирование псевдослучайных чисел 119
 - методы 120

- линейная алгебра 131
 - методы булевых массивов 128
 - недостатки 232
 - общие сведения 25, 99
 - перестановка данных 227
 - программирование на основе массивов 123
 - unique и другие теоретико-множественные операции 130
 - векторизация 123
 - логические условия как операции с массивами 125
 - случайное блуждание 133, 135
 - производительность 491
 - непрерывная память 491
 - список рассылки 33
 - статистические операции 126
 - типы данных 105
 - string_предостережение 106
 - знаки подчеркивания в конце имени 463
 - иерархия 462
 - универсальные функции 120, 477
 - в pandas 170
 - методы экземпляра 477
 - файловый ввод-вывод массивов 130
- O**
- open(), открытие файла 93
 - or, ключевое слово 58, 61
 - outer() (u-функция NumPy) 478
- P**
- pairplot() (seaborn) 311
 - pandas
 - построение графиков
 - графики плотности 309
 - PandasArray 138
 - pass, предложение 63
 - Patsy
 - матрицы плана 400
 - метаданные модели 402
 - общее описание 400
 - передача объектов NumPy 402
 - преобразование данных 402
 - формулы 400
 - категориальные данные 404
 - Patsy общее описание 29
 - pdb, отладчик 504
 - PeriodIndex() 279
 - PeriodIndex, индексный объект 378
 - PeriodIndex объект 374
 - преобразование с изменением частоты 374
 - создание из массива 379
 - period_range() 373
 - Period объект 373
 - квартальная частота 376
 - преобразование временных меток 377
 - преобразование частоты 374
 - permutations() (itertools) 90
 - pickle модуль 203
 - read_pickle() 180, 204
 - to_pickle() 203
 - pivot() 281
 - pivot_table() 347
 - аргументы 350
 - Plotly библиотека визуализации 315
 - plot() (matplotlib) 288
 - Polygon() (matplotlib) 297
 - pop() удаление столбца из DataFrame 280
 - product() (itertools) 90
 - put() (NumPy) 470
 - pyarrow пакет для read_parguet() 204
 - PyCharm IDE 33
 - pydata, группа Google 33
 - PyDev IDE 33
 - PyTables пакет для работы с HDF5-файлами 206
 - Python Tools для Visual Studio (Windows) 33
- Q**
- qcut() распределение данных в соответствии с квантилями 225
 - в сочетании с groupby() 335
- R**
- raise_for_status() проверка ошибок HTTP 208
 - range() 63
 - ravel() (NumPy) 465
 - rc() конфигурирование matplotlib 298
 - readable() 95
 - read_csv() 188
 - read_excel() 204
 - read_hdf() 207
 - read_html() 200

- read_json() 199
- readlines() 95
- read_parquet() 204
- read_pickle() 180, 204
- read_sql() 211
- read_xml() 203
- read_* ? функции загрузки данных 186
- read() чтение файла 94
- Rectangle() (matplotlib) 297
- reduceat() (u-функция NumPy) 478
- reduce() (u-функция NumPy) 477
- regplot() (seaborn) 311
- reindex() (pandas) 151
 - аргументы 153
 - оператор loc 154
 - передискретизация 386
- remove() (множества) 77
- rename() преобразование данных 223
- repeat() (NumPy) 468
- replace()
 - преобразование данных 221
 - строки 236
- resample() 380
 - аргументы 381
 - передискретизация периодов 386
- reset_index() 259
- reshape() (NumPy) 117, 464
- return, предложение 82
- reversed(), генератор 80
- rolling() функция скользящего окна 389
- S**
- sample() случайная выборка 228
- savefig() (matplotlib) 297
- save() (NumPy) 131, 489
- scatter() (matplotlib) 288
- scikit-learn
 - восполнение неполных данных 412
 - запрет неполных данных 411
 - общие сведения 28, 410
 - перекрестный контроль 413
 - список рассылки 33
 - установка 410
- SciPy
 - общие сведения 28
 - список рассылки 33
 - установка 310
- seaborn библиотека
 - гистограммы 308
 - графики плотности 309
 - диаграммы размаха 314
 - диаграммы рассеяния 311
 - общие сведения 285, 299
 - онлайн-документация 315
 - столбчатые диаграммы 307
- seaborn, библиотека 299
- searchsorted() (NumPy) 486
- seekable() 96
- seek() поиск в файле 94
- Series (pandas) 138
 - get_dummies() 230
 - map() преобразование данных 220
 - replace() 221
 - unique и другие теоретико-множественные операции 181
- арифметические операции 165
- восполнение значений 166
- атрибут array 138
- атрибут index 138
- атрибут name 141
- группировка 324
- и u-функции NumPy 170
- импорт и экспорт в формате JSON 199
- индекс 138
- индексы по осям с повторяющимися значениями 175
- конкатенация вдоль оси 269
- мультииндекс 254
- объекты Index 149
- отсутствующие данные 214, 218
- подвохи целочисленного индексирования 162
- преобразование в словарь и обратно 139
- ранжирование 174
- расширенные типы данных 232
- сортировка 172
- список арифметических методов 168
- статистические методы
 - корреляция и ковариация 180
 - сводные статистики 177, 258
 - таблицы измерений 244
 - удаление элементов из оси 154
- формат HDF5 205
- set() 76
- setattr() 50
- set_index() установить столбец DataFrame в качестве индексного 259

set_title() (matplotlib) 294
 set_trace() 507
 set_xlabel() (matplotlib) 294
 set_xlim() (matplotlib) 297
 set_xticklabels() (matplotlib) 293
 set_xticks() (matplotlib) 293
 set_ylim() (matplotlib) 297
 size() размер группы 321
 sorted(), возврат нового списка 79
 sort_index() по всем или подмножеству уровней 257
 split(), разбиение массива (NumPy) 467
 split() разбиение строки 235
 Spyder IDE 33
 SQLAlchemy проект 211
 SQLite3 база данных 210
 sqrt() (u-функция NumPy) 120
 stack() 256, 276, 280
 standardize() (Patsy) 402
 statsmodels
 анализ временных рядов 409
 запрет неполных данных 411
 модели линейной регрессии 407
 общие сведения 29, 406
 statsmodels, список рассылки 33
 statsmodels установка 343
 str() 55
 strftime() 60, 354
 strip() удаление пробелов 235
 strptime() 60, 355
 str (строка) скалярный тип 54
 f-строки 57
 в кодировке UTF-8,
 декодирование 96
 встроенные методы 235
 многострочные 55
 неизменяемость 55
 онлайн-документация 57
 преобразование в тип datetime
 и обратно 60
 преобразование объектов в 55
 регулярные выражения 237
 список методов 242
 тип NumPy string_ 106
 форматирование 56
 subplots_adjust() (matplotlib) 290
 subplots() (matplotlib) 289
 swapaxes() (NumPy) 118
 swaplevel() (NumPy) 257

symmetric_difference_update()
 (множества) 77
 symmetric_difference() (множества) 77

T

take() (NumPy) 471
 tell() позиция в файле 94
 TextFileReader объект 193
 text () (matplotlib) 295
 tile() (NumPy) 469
 timedelta тип 60, 354
 Timestamp (pandas)
 основы временных рядов 356
 сдвиг дат с помощью смещений 367
 с учетом поясного времени 371
 форматирование 354
 timezone() (pytz) 369
 time тип 59, 354
 to_csv() 195
 to_datetime() 355
 to_excel() 205
 to_json() 199
 to_numpy() 398
 to_period() 377
 to_pickle() 203
 to_timestamp() 279
 transform() 343
 transpose() и атрибут T 117
 try/except блоки 90
 tuple() 65
 TypeError, исключение 91, 106
 tz_convert() 370
 tzinfo тип 354
 tz_localize() 370

U

UInt8Dtype расширенный тип
 данных 235
 UInt16Dtype расширенный тип
 данных 235
 UInt32Dtype расширенный тип
 данных 235
 UInt64Dtype расширенный тип
 данных 35
 unicode, тип 105
 union() 76
 метод Index 150
 unstack() 255, 276
 update() (множества) 77
 UTF-8 кодировка 57, 93

u-функции (универсальные функции)
 для массивов ndarray 120
 и объекты pandas 170
 компиляция с помощью
 Numba 489
 методы 477
 написание на Python 479
 унарные 120

V

value_counts() 182, 244
 категориальные объекты 251
 столбчатые диаграммы 305
ValueError, исключение 90
vectorize() (NumPy) 480
vstack() (NumPy) 467

W

Web API, взаимодействие с 208
where() (NumPy) 274

while циклы 62
 замечание о производительности 491
Windows
 Python Tools для Visual Studio 33
 команда type вывода на терминал 188
 установка Miniconda 30
writable() 96
writelines() вывод в файл 95
write() вывод в файл 95

X

xlim() (matplotlib) 292
XML формат файлов 200
 разбор 201

Y

yield в функциях 88

Z

zip() для списка кортежей 79

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, пр. Андропова д. 38 оф. 10.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.galaktika-dmk.com.
Оптовые закупки: тел. (499) 782-38-89.
Электронный адрес: books@aliens-kniga.ru.

Уэс Маккинни

Python и анализ данных

Третье издание

Главный редактор	Мовчан Д. А. dmkpress@gmail.com
Зам. главного редактора	Сенченкова Е. А.
Перевод	Слинкин А. А.
Корректор	Синяева Г. И.
Верстка	Луценко С. В.
Дизайн обложки	Мовчан А. Г.

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 43,55. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Перед вами авторитетное руководство по перемативированию, очистке и обработке наборов данных на Python. Третье издание, переработанное с учетом версией Python 3.10 и pandas 1.4, содержит практические примеры, демонстрирующие эффективное решение широкого круга задач анализа данных.

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

Файлы данных и прочие материалы к книге находятся в репозитории на GitHub и на сайте издательства dmkpress.com.

- Использование Jupyter-блокнотов для разведывательного анализа данных.
- Основы и более сложные средства библиотеки NumPy.
- Введение в средства анализа данных в библиотеке pandas.
- Использование гибких инструментов для загрузки, очистки, преобразования, слияния и изменения формы данных.
- Информативная визуализация с помощью библиотеки matplotlib.
- Применение встроенного в pandas механизма группировки для формирования продольных и поперечных срезов, а также агрегирования наборов данных.
- Анализ регулярных и нерегулярных временных рядов и манипуляции с ними.
- Методы решения реальных задач анализа данных, проиллюстрированные подробными примерами.

«В это новое издание Уэс внес изменения, так чтобы книга и дальше оставалась востребованным источником по всем аспектам анализа данных с применением Python и pandas. Горячо и настоятельно рекомендую».

*Пол Берри,
лектор и автор книги
«Head First Python»*



Уэс Маккинни — соучредитель и технический директор компании Voltron Data, является активным членом сообщества обработки данных на Python и агитирует за использование Python в анализе данных, финансовых задачах и математической статистике. Выпускник MIT, он также является членом комитетов по управлению проектами Apache Arrow и Apache Parquet, курируемых фондом Apache Software Foundation.