

Pandas

В ДЕЙСТВИИ

Борис Пасхавер



Pandas in Action

BORIS PASKHAVER



MANNING
SHELTER ISLAND

Pandas В ДЕЙСТВИИ

Борис Пасхавер



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.43
П19

Пасхавер Борис

П19 Pandas в действии. — СПб.: Питер, 2023. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1941-7

Язык Python помогает упростить анализ данных. Если вы научились пользоваться электронными таблицами, то сможете освоить и pandas! Несмотря на сходство с табличной компоновкой Excel, pandas обладает большей гибкостью и более широкими возможностями. Эта библиотека для Python быстро выполняет операции с миллионами строк и способна взаимодействовать с другими инструментами. Она дает идеальную возможность выйти на новый уровень анализа данных.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617297434 англ.
ISBN 978-5-4461-1941-7

© 2021 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022

Краткое содержание

Предисловие	17
Благодарности	19
О книге.	22
Об авторе	26
Иллюстрация на обложке	27
От издательства	28

Часть I Основы pandas

Глава 1. Знакомство с библиотекой pandas	30
Глава 2. Объект Series	53
Глава 3. Методы класса Series	91
Глава 4. Объект DataFrame	120
Глава 5. Фильтрация объектов DataFrame	160

Часть II Библиотека pandas на практике

Глава 6. Работа с текстовыми данными	198
Глава 7. Мультииндексные объекты DataFrame	220
Глава 8. Изменение формы и сводные таблицы	260

6 Краткое содержание

Глава 9. Объект GroupBy	286
Глава 10. Слияние, соединение и конкатенация	307
Глава 11. Дата и время	332
Глава 12. Импорт и экспорт данных	366
Глава 13. Настройка pandas	390
Глава 14. Визуализация	401

Приложения

Приложение А. Установка и настройка	414
Приложение Б. Экспресс-курс по языку Python	437
Приложение В. Экспресс-курс по библиотеке NumPy	481
Приложение Г. Генерирование фиктивных данных с помощью Faker	490
Приложение Д. Регулярные выражения	497

Оглавление

Предисловие	17
Благодарности	19
О книге.	22
Для кого она предназначена	22
Как организована эта книга: дорожная карта	22
О коде	24
Источники информации в Интернете	25
Об авторе	26
Иллюстрация на обложке	27
От издательства	28

Часть I Основы pandas

Глава 1. Знакомство с библиотекой pandas	30
1.1. Данные в XXI веке	31
1.2. Знакомство с pandas	31
1.2.1. Библиотека pandas по сравнению с визуальными приложениями электронных таблиц	34
1.2.2. Pandas по сравнению с конкурентами	36

8 Оглавление

1.3. Обзор библиотеки pandas.	38
1.3.1. Импорт набора данных	38
1.3.2. Операции над объектами DataFrame.	41
1.3.3. Подсчет значений в Series	44
1.3.4. Фильтрация столбца по одному или нескольким критериям.	45
1.3.5. Группировка данных.	48
Резюме	51
Глава 2. Объект Series	53
2.1. Обзор Series	54
2.1.1. Классы и экземпляры	55
2.1.2. Наполнение объекта Series значениями.	56
2.1.3. Пользовательские индексы для Series	58
2.1.4. Создание объекта Series с пропущенными значениями	62
2.2. Создание объектов Series на основе объектов языка Python	63
2.3. Атрибуты Series.	65
2.4. Извлечение первой и последней строк	68
2.5. Математические операции	70
2.5.1. Статистические операции	70
2.5.2. Арифметические операции.	78
2.5.3. Транслирование.	81
2.6. Передача объектов Series встроенным функциям языка Python	84
2.7. Упражнения	86
2.7.1. Задачи.	86
2.7.2. Решения	87
Резюме	89
Глава 3. Методы класса Series	91
3.1. Импорт набора данных с помощью функции read_csv	92
3.2. Сортировка объектов Series	98
3.2.1. Сортировка значений с помощью метода sort_values	98
3.2.2. Сортировка по индексу с помощью метода sort_index.	101
3.2.3. Получение минимального и максимального значений с помощью методов nsmallest и nlargest	102

3.3. Перезапись объекта Series с помощью параметра inplace	104
3.4. Подсчет количества значений с помощью метода value_counts.	106
3.5. Вызов функции для каждого из значений объекта Series с помощью метода apply	112
3.6. Упражнение	116
3.6.1. Задача	116
3.6.2. Решение.	117
Резюме	119
Глава 4. Объект DataFrame	120
4.1. Обзор DataFrame	121
4.1.1. Создание объекта DataFrame на основе ассоциативного массива	121
4.1.2. Создание объекта DataFrame на основе ndarray библиотеки NumPy	123
4.2. Общие черты Series и DataFrame	125
4.2.1. Импорт объекта DataFrame с помощью функции read_csv. . .	125
4.2.2. Атрибуты Series и DataFrame: сходство и различие	127
4.2.3. Общие методы Series и DataFrame	130
4.3. Сортировка объекта DataFrame.	134
4.3.1. Сортировка по одному столбцу	134
4.3.2. Сортировка по нескольким столбцам	135
4.4. Сортировка по индексу	137
4.4.1. Сортировка по индексу строк	138
4.4.2. Сортировка по индексу столбцов	139
4.5. Задание нового индекса.	140
4.6. Извлечение столбцов из объектов DataFrame	141
4.6.1. Извлечение одного столбца из объекта DataFrame.	141
4.6.2. Извлечение нескольких столбцов из объекта DataFrame . . .	142
4.7. Извлечение строк из объектов DataFrame	144
4.7.1. Извлечение строк по метке индекса	144
4.7.2. Извлечение строк по позиции индекса	146
4.7.3. Извлечение значений из конкретных столбцов	149

4.8. Извлечение значений из объектов Series	152
4.9. Переименование столбцов и строк	152
4.10. Замена индекса	154
4.11. Упражнения	155
4.11.1. Задачи	155
4.11.2. Решения.	155
Резюме	159
Глава 5. Фильтрация объектов DataFrame	160
5.1. Оптимизация памяти, используемой набором данных	161
5.1.1. Преобразование типов данных с помощью метода astype.	163
5.2. Фильтрация по одному условию	168
5.3. Фильтрация по нескольким условиям	173
5.3.1. Условие И	173
5.3.2. Условие ИЛИ	174
5.3.3. Логическое отрицание (~)	175
5.3.4. Методы для работы с булевыми значениями	176
5.4. Фильтрация по условию	177
5.4.1. Метод isin.	177
5.4.2. Метод between.	178
5.4.3. Методы isnull и notnull	180
5.4.4. Обработка пустых значений	182
5.5. Решение проблемы дубликатов	185
5.5.1. Метод duplicated	185
5.5.2. Метод drop_duplicates.	187
5.6. Упражнения	191
5.6.1. Задачи.	191
5.6.2. Решения	192
Резюме	196

Часть II

Библиотека pandas на практике

Глава 6. Работа с текстовыми данными	198
6.1. Регистр букв и пробелы.	199
6.2. Срезы строковых значений.	203
6.3. Срезы строковых значений и замена символов	205
6.4. Булевы методы	207
6.5. Разбиение строковых значений	210
6.6. Упражнение	215
6.6.1. Задача	215
6.6.2. Решение.	215
6.7. Примечание относительно регулярных выражений	217
Резюме	218
Глава 7. Мультииндексные объекты DataFrame	220
7.1. Объект MultiIndex	222
7.2. Объекты DataFrame с мультииндексами	226
7.3. Сортировка мультииндексов	232
7.4. Выборка данных с помощью мультииндексов.	236
7.4.1. Извлечение одного или нескольких столбцов	237
7.4.2. Извлечение одной или нескольких строк с помощью loc	240
7.4.3. Извлечение одной или нескольких строк с помощью iloc.	246
7.5. Поперечные срезы	248
7.6. Операции над индексом.	249
7.6.1. Замена индекса	250
7.6.2. Задание индекса	253
7.7. Упражнения	255
7.7.1. Задачи.	255
7.7.2. Решения	257
Резюме	259

Глава 8. Изменение формы и сводные таблицы	260
8.1. Широкие и узкие данные.	261
8.2. Создание сводной таблицы из объекта DataFrame	263
8.2.1. Метод <code>pivot_table</code>	264
8.2.2. Дополнительные возможности для работы со сводными таблицами	268
8.3. Перенос уровней индексов с оси столбцов на ось строк и наоборот.	271
8.4. Расплавление набора данных	273
8.5. Развертывание списка значений	278
8.6. Упражнения	280
8.6.1. Задачи.	280
8.6.2. Решения	281
Резюме	285
Глава 9. Объект GroupBy	286
9.1. Создание объекта GroupBy с нуля	287
9.2. Создание объекта GroupBy из набора данных.	289
9.3. Атрибуты и методы объекта GroupBy	292
9.4. Агрегатные операции	296
9.5. Применение собственных операций ко всем группам набора	300
9.6. Группировка по нескольким столбцам	301
9.7. Упражнения	303
9.7.1. Задачи.	303
9.7.2. Решения	304
Резюме	306
Глава 10. Слияние, соединение и конкатенация	307
10.1. Знакомство с наборами данных	309
10.2. Конкатенация наборов данных	311
10.3. Отсутствующие значения в объединенных DataFrame	314
10.4. Левые соединения	316
10.5. Внутренние соединения.	318
10.6. Внешние соединения.	320

10.7. Слияние по индексным меткам	323
10.8. Упражнения	325
10.8.1. Задачи	327
10.8.2. Решения.	327
Резюме	330
Глава 11. Дата и время	332
11.1. Знакомство с объектом Timestamp	333
11.1.1. Как Python работает с датой и временем	333
11.1.2. Как pandas работает с датой и временем.	336
11.2. Хранение нескольких отметок времени в DatetimeIndex	339
11.3. Преобразование значений столбцов или индексов в дату и время	341
11.4. Использование объекта DatetimeProperties	343
11.5. Сложение и вычитание интервалов времени.	348
11.6. Смещение дат	350
11.7. Объект Timedelta	353
11.8. Упражнения	358
11.8.1. Задачи	358
11.8.2. Решения.	360
Резюме	364
Глава 12. Импорт и экспорт данных	366
12.1. Чтение и запись файлов JSON.	367
12.1.1. Загрузка файла JSON в DataFrame	369
12.1.2. Экспорт содержимого DataFrame в файл JSON	376
12.2. Чтение и запись файлов CSV	377
12.3. Чтение книг Excel и запись в них	380
12.3.1. Установка библиотек xlrd и openpyxl в среде Anaconda	380
12.3.2. Импорт книг Excel	381
12.3.3. Экспорт книг Excel	384
12.4. Упражнения	386
12.4.1. Задачи	387
12.4.2. Решения.	387
Резюме	389

Глава 13. Настройка pandas390
13.1. Получение и изменение параметров настройки pandas391
13.2. Точность396
13.3. Максимальная ширина столбца397
13.4. Порог округления до нуля397
13.5. Параметры контекста398
Резюме400
Глава 14. Визуализация401
14.1. Установка Matplotlib.401
14.2. Линейные графики402
14.3. Гистограммы408
14.4. Круговые диаграммы.410
Резюме412

Приложения

Приложение А. Установка и настройка414
А.1. Дистрибутив Anaconda414
А.2. Процесс установки в macOS.416
А.2.1. Установка Anaconda в macOS.416
А.2.2. Запуск терминала417
А.2.3. Типичные команды, доступные в терминале418
А.3. Процесс установки в Windows419
А.3.1. Установка Anaconda в Windows419
А.3.2. Запуск командной оболочки Anaconda421
А.3.3. Типичные команды, доступные в Anaconda Prompt422
А.4. Создание новых окружений Anaconda424
А.5. Anaconda Navigator429
А.6. Основы Jupyter Notebook432
Приложение Б. Экспресс-курс по языку Python437
Б.1. Простые типы данных438
Б.1.1. Числа439

Б.1.2. Строки439
Б.1.3. Логические значения443
Б.1.4. Объект None.443
Б.2. Операторы.444
Б.2.1. Математические операторы444
Б.2.2. Операторы проверки на равенство и неравенство446
Б.3. Переменные448
Б.4. Функции449
Б.4.1. Аргументы и возвращаемые значения.450
Б.4.2. Пользовательские функции454
Б.5. Модули.456
Б.6. Классы и объекты457
Б.7. Атрибуты и методы458
Б.8. Методы строк459
Б.9. Списки463
Б.9.1. Итерации по спискам469
Б.9.2. Генераторы списков470
Б.9.3. Преобразование строки в список и обратно471
Б.10. Кортежи472
Б.11. Словари.474
Б.11.1. Итерации по словарям477
Б.12. Множества478
Приложение В. Экспресс-курс по библиотеке NumPy.481
В.1. Измерения.481
В.2. Объект ndarray483
В.2.1. Создание набора последовательных чисел с помощью метода arange483
В.2.2. Атрибуты объекта ndarray484
В.2.3. Метод reshape485
В.2.4. Функция randint487
В.2.5. Функция randn488
В.3. Объект nan.489

Приложение Г. Генерирование фиктивных данных с помощью Faker490
Г.1. Установка Faker490
Г.2. Начало работы с Faker491
Г.3. Заполнение набора данных DataFrame фиктивными значениями494
Приложение Д. Регулярные выражения497
Д.1. Введение в модуль re498
Д.2. Метасимволы499
Д.3. Расширенные шаблоны поиска503
Д.4. Регулярные выражения и pandas.507

Предисловие

Честно говоря, я наткнулся на Pandas совершенно случайно.

В 2015 году я проходил собеседование на должность аналитика по обработке данных на Indeed.com, крупнейшем сайте в мире по поиску работы. В качестве последнего технического задания меня попросили извлечь полезную информацию из внутреннего набора данных с помощью электронных таблиц Microsoft Excel. Стремясь впечатлить работодателя, я постарался вытянуть все, что только можно, из своего набора инструментов анализа данных: использовал сортировки столбцов, операции над текстом, сводные таблицы и, конечно, легендарную функцию VLOOKUP (ну, может, «*легендарная*» — небольшое преувеличение).

Как ни странно, тогда я не знал, что существуют и другие инструменты анализа данных, кроме Excel. Электронные таблицы Excel были повсюду: их использовали мои родители, мои преподаватели и мои коллеги. Они казались уже установившимся стандартом. Так что после получения письменного приглашения на работу я сразу же купил книг по Excel примерно на 100 долларов и начал их изучать, чтобы стать специалистом по электронным таблицам!

В первый день я пришел на работу с распечаткой списка 50 чаще всего используемых функций Excel. Но едва я успел войти в учетную запись на рабочем компьютере, как начальник вызвал меня в конференц-зал и сообщил, что приоритеты изменились. Возникли проблемы. Первая: наборы данных команды аналитиков разрослись до не поддерживаемых Excel размеров. Вторая: все члены команды по-прежнему или даже в еще большей степени нуждались в автоматизации рутинных операций при создании ежедневных и еженедельных отчетов и усиленно искали инструменты и способы осуществить ее.

К счастью, наш начальник придумал решение обеих проблем. Он спросил меня, слышал ли я о Pandas.

— О таких пушистых зверьках?¹ — переспросил я в недоумении.

— Нет, — сказал он, — о библиотеке Python для анализа данных.

Итак, после всей проведенной подготовки мне пришлось осваивать совершенно новую технологию, изучать ее с нуля. Я нервничал, ведь раньше мне не приходилось писать код. Я ведь был специалистом по Excel, правда? Сумею ли я? Был только один способ узнать. Я углубился в официальную документацию Pandas, учебные видео на YouTube, книги, материалы семинаров, вопросы на Stack Overflow и во все наборы данных, какие только мог найти. И был приятно удивлен тому, насколько легко и просто начать работать с этой библиотекой. Код был интуитивно понятен и прост. Сама библиотека функционировала очень быстро. Хорошо проработанных возможностей было много. Pandas позволяла произвести множество операций над данными при помощи очень небольшого объема кода.

Истории, подобные моей, нередки в сообществе Python. Баснословный рост популярности этого языка за последнее десятилетие часто связывают с легкостью изучения его новыми разработчиками. Я убежден, что в схожей с моей ситуации вы сможете изучить Pandas столь же легко. Если вы хотите вывести свои навыки анализа данных за пределы электронных таблиц Excel, эта книга для вас.

После того как я научился уверенно работать с Pandas, я продолжил изучать Python, а затем и другие языки программирования. Во многом библиотека Pandas стала моей стартовой площадкой к переходу в профессиональные разработчики ПО. Я обязан этой замечательной библиотеке очень многим и с радостью передаю эстафету ее освоения вам. Надеюсь, вы оцените волшебные возможности, открываемые для вас программированием.

¹ На английском языке название библиотеки Pandas омонимично слову «панды». — *Примеч. пер.*

Благодарности

Довести эту книгу до успешного финиша было нелегко, и я хотел бы выразить искреннюю признательность всем тем, кто поддерживал меня на протяжении двух лет ее написания.

Прежде всего самые теплые благодарности моей замечательной девушке, Мередит. С самой первой фразы книги я чувствовал ее неизменную поддержку. Мередит — жизнерадостная, забавная и добрая душа, всегда ободряющая меня, когда дела идут плохо. Эта книга стала намного лучше благодаря ей. Спасибо, Мермишка.

Спасибо моим родителям, Ирине и Дмитрию, за гостеприимный дом, в котором я всегда могу получить передышку.

Спасибо моим сестрам-близняшкам, Маше и Саше. Они такие умные, любознательные и трудолюбивые для своего возраста, и я очень ими горжусь. Удачи в колледже!

Спасибо Ватсону, нашему золотистому ретриверу. Недостаток знаний Python он компенсирует веселым и дружелюбным нравом.

Огромное спасибо моему редактору, Саре Миллер (Sarah Miller), работа с которой была сплошным удовольствием. Я благодарен ей за терпение и ценные идеи по ходу дела. Она была подлинным капитаном нашего судна, благодаря которому плавание проходило без штормов.

Я не стал бы разработчиком программного обеспечения, если бы не возможности, предоставленные мне компанией Indeed. Хотелось бы сердечно поблагодарить моего бывшего начальника, Срджана Бодружича (Srdjan Bodruzic), за щедрость

и наставничество (а также за то, что взял меня на работу!). Спасибо коллегам по команде CX — Томми Винчелу (Tommy Winschel), Мэтью Морину (Matthew Morin), Крису Хаттону (Chris Hatton), Чипу Борси (Chip Borsi), Николь Салимбене (Nicole Saglimbene), Дэниэль Сколи (Danielle Scoli), Блэр Суэйн (Blair Swayne) и Джорджу Имроглу (George Improglou). Спасибо всем, с кем я обедал в Sophie's Cuban Cuisine!

Я начал писать эту книгу, будучи разработчиком программного обеспечения в Stride Consulting. Потому хотел бы поблагодарить множество сотрудников Stride за их поддержку: Дэвида «Доминатора» Дипанфило (David «The Dominator» DiPanfilo), Мина Квака (Min Kwak), Бена Блэра (Ben Blair), Кирстен Нордин (Kirsten Nordine), Майкла «Бобби» Нуньеса (Michael «Bobby» Nunez), Джея Ли (Jay Lee), Джеймса Ю (James Yoo), Рэя Велиса (Ray Veliz), Нэйтана Римера (Nathan Riemer), Джулию Берчем (Julia Berchem), Дэна Плэйна (Dan Plain), Ника Чера (Nick Char), Гранта Циолковски (Grant Ziolkowski), Мелиссу Ваниш (Melissa Wahnish), Дэйва Эндерсона (Dave Anderson), Криса Апорты (Chris Aporta), Майкла Карлсона (Michael Carlson), Джона Галиото (John Galioto), Шона Марцуг-Маккарти (Sean Marzug-McCarthy), Трэвиса Вандерхупа (Travis Vander Hoop), Стива Соломона (Steve Solomon) и Яна Млкоха (Jan Mlčoch).

Спасибо вам, друзья-коллеги, встречавшиеся мне в те времена, когда я был разработчиком программного обеспечения и консультантом: Фрэнсис Хван (Francis Hwang), Инхак Ким (Inhak Kim), Лиана Лим (Liana Lim), Мэтт Бамбах (Matt Bambach), Брентон Моррис (Brenton Morris), Иэн Макнэлли (Ian McNally), Джош Филиппс (Josh Philips), Артем Кочнев (Artem Kochnev), Эндрю Кан (Andrew Kang), Эндрю Фейдер (Andrew Fader), Карл Смит (Karl Smith), Брэдли Уитвелл (Bradley Whitwell), Брэд Пополек (Brad Popiolek), Эдди Вартон (Eddie Wharton), Джен Квок (Jen Kwok), и вам, мои любимые соратники по кофе: Адам Макамис (Adam McAmis) и Энди Фриц (Andy Fritz).

Благодарю также Ника Бьянко (Nick Bianco), Кэма Штира (Cam Stier), Кейт Дэвид (Keith David), Майкла Чена (Michael Cheung), Томаса Филиппо (Thomas Philipreau), Николь Диандреа (Nicole DiAndrea) и Джеймса Рокича (James Rokeach) за все, что они сделали для меня.

Спасибо моей любимой группе, New Found Glory, за музыку, которую я часто слушал при написании этой книги. Поп-панк жив!

Выражаю благодарность сотрудникам издательства Manning, доведшим этот проект до завершения и помогавшим с его продвижением: Дженнифер Уль (Jennifer Houle), Александару Драгосавлевицу (Aleksandar Dragosavljević), Радмиле Эрцеговац (Radmila Ercegovic), Кэндес Джиллхул (Candace Gillhoolley), Степану Джурековичу (Stjepan Jureković) и Лукасу Веберу (Lucas Weber). Спасибо также тем сотрудникам Manning, которые проверяли все материалы книги: Саре Миллер, моему редактору-консультанту по аудитории; Дейдре Хиама (Deirdre

Niam), моему менеджеру по выпуску; Кейр Симпсон (Keir Simpson), моему выпускающему редактору; и Джейсону Эверетту (Jason Everett), моему корректору.

Спасибо всем техническим рецензентам, которые помогли мне довести книгу до ума: Элу Пежевски (Al Pezewski), Альберто Кьярланти (Alberto Ciarlanti), Бену Макнамаре (Ben McNamara), Бьерну Нойхаусу (Björn Neuhaus), Кристоферу Коттмайеру (Christopher Kottmyer), Дэну Шейху (Dan Sheikh), Драгошу Манайлу (Dragos Manailoiu), Эрико Ленцзяну (Erico Lenzian), Джеффу Смитту (Jeff Smith), Жерому Батону (Jérôme Bâton), Хоакину Белтрану (Joaquín Beltrán), Джонатану Шарли (Jonathan Sharley), Хосе Апаблазе (José Apablaza), Кену В. Элджеру (Ken W. Alger), Мартину Цыгану (Martin Czygan), Маттейсу Афуртиту (Mathijs Affourtit), Мэтиасу Бушу (Matthias Busch), Майку Кадди (Mike Cuddy), Монике И. Гимарайеш (Monica E. Guimaraes), Нинославу Черкезу (Ninoslav Cerkez), Рикку Принсу (Rick Prins), Саиду Хасани (Syed Hasany), Витону Витани (Viton Vitanis) и Вибхавредди Камиредди Шангальредди (Vybhavreddy Kammireddy Changanreddy). Благодаря вашим стараниям я научился писать и преподавать лучше.

Наконец, выражаю признательность городу Хобокен — месту моего проживания на протяжении последних шести лет. Я написал многие части этой рукописи в его общественной библиотеке, местных кафе и чайных лавках. Немало шагов в своей жизни я сделал в этом городе, и он навсегда останется в моей памяти. Спасибо тебе, Хобокен!

ДЛЯ КОГО ОНА ПРЕДНАЗНАЧЕНА

«Pandas в действии» представляет собой полезное, полное и понятное введение в библиотеку Pandas¹, предназначенную для анализа данных. Pandas позволяет с легкостью производить множество операций над данными: сортировку, соединение, создание сводных таблиц, очистку, удаление повторов, агрегирование и многое другое. Все перечисленное рассматривается в книге по нарастающей сложности. Вы познакомитесь с pandas по частям, начиная с самых мелких «кирпичиков» и постепенно переходя к более крупным структурам данных.

Книга предназначена для специалистов по анализу данных, ранее работавших с программами электронных таблиц (например, Microsoft Excel, Google Sheets и Apple Numbers) и/или альтернативными инструментами анализа данных (например, R и SAS). Подходит она и для разработчиков Python, интересующихся анализом данных.

КАК ОРГАНИЗОВАНА ЭТА КНИГА: ДОРОЖНАЯ КАРТА

«Pandas в действии» состоит из 14 глав, сгруппированных в две части.

Часть I «Основы pandas» поэтапно знакомит вас с основной спецификой работы с библиотекой pandas.

¹ Далее в тексте — pandas. — *Примеч. ред.*

- В главе 1 с помощью `pandas` анализируется пример набора данных для демонстрации ее возможностей.
- В главе 2 вы познакомитесь с объектом `Series` — основной структурой данных `pandas`, предназначенной для хранения набора упорядоченных данных.
- В главе 3 мы подробнее рассмотрим объект `Series`, в частности различные операции `Series`, включая сортировку значений, отбрасывание дубликатов, извлечение минимальных и максимальных значений и многое другое.
- В главе 4 вы познакомитесь с объектом `DataFrame`, двумерной таблицей данных. Мы применим идеи из предыдущих глав к этой новой структуре данных и рассмотрим дополнительные операции.
- В главе 5 вы научитесь отфильтровывать поднаборы строк из объекта `DataFrame` с помощью различных логических условий: равенства, неравенства, включения, исключения и т. д.

Часть II «Применение `pandas` на практике» посвящена более продвинутым возможностям библиотеки `pandas` и задачам, решаемым с их помощью в реальных наборах данных.

- В главе 6 вы научитесь работать в `pandas` с содержащими ошибки текстовыми данными. Мы обсудим решение таких задач, как удаление пробельных символов, исправление регистра символов и извлечение нескольких значений из одного столбца.
- Глава 7 обсуждает `MultiIndex`, предназначенный для объединения значений из нескольких столбцов под единым идентификатором для строки данных.
- Глава 8 описывает агрегирование данных в сводных таблицах, перенос заголовков с оси строк на ось столбцов и преобразование данных из широкого формата в узкий.
- В главе 9 изучается вопрос группировки строки по корзинам и агрегирование полученных коллекций при помощи объекта `GroupBy`.
- Глава 10 описывает объединение нескольких наборов данных в один с помощью различных видов соединений.
- Глава 11 демонстрирует возможности работы с датами и временем в библиотеке `pandas` и охватывает такие вопросы, как сортировка дат, вычисление длительности и определение того, приходится ли дата на начало месяца или квартала.
- Глава 12 показывает, как импортировать дополнительные типы файлов в библиотеке `pandas`, включая данные Excel и JSON. Также вы научитесь экспортировать данные из библиотеки `pandas`.

- Глава 13 посвящена настройке параметров библиотеки pandas. Мы обсудим изменение количества отображаемых строк, настройку точности чисел с плавающей точкой, округление значений и многое другое.
- В главе 14 вы откроете для себя возможности визуализации данных с помощью библиотеки matplotlib. Мы увидим, как создавать на основе данных из библиотеки pandas линейные, столбчатые, круговые диаграммы и многое другое.

Содержимое каждой новой главы основывается на материале предыдущих. Изучающим pandas с нуля я рекомендую читать их последовательно. Но в то же время, чтобы книгу можно было использовать в качестве справочного руководства, я старался делать все главы независимыми, с отдельными наборами данных. В каждой главе мы будем начинать писать код с чистого листа, так что вы можете начать читать с любой из них.

В конце большинства глав вы найдете упражнения, на которых сможете опробовать изложенные принципы на практике. Я настоятельно рекомендую не игнорировать их.

Библиотека pandas основана на языке программирования Python, так что вам следует озаботиться знанием его основ перед чтением книги. Те, у кого мало опыта работы с Python, могут основательно познакомиться с ним в приложении Б.

О КОДЕ

Эта книга содержит множество примеров исходного кода, который набран **вот таким моноширинным шрифтом**, чтобы его можно было отличить от обычного текста.

Исходный код для всех листингов данной книги доступен для скачивания с GitHub по адресу <https://github.com/paskhaver/pandas-in-action>. Если вы не сталкивались с Git и GitHub — найдите кнопку **Download Zip** на странице репозитория. Те же, у кого есть опыт работы с Git и GitHub, могут клонировать репозиторий из командной строки.

Репозиторий также включает полные наборы данных для книги. Когда я изучал библиотеку pandas, меня очень раздражало, что в руководствах часто используются сгенерированные случайным образом данные. Никакой согласованности, контекста, последовательности изложения, никакого интереса, наконец. В этой книге мы будем работать с множеством реальных наборов данных, охватывающих все на свете, от зарплат баскетболистов и типов покемонов до санитарных инспекций ресторанов. Данные повсюду вокруг нас, и библиотека pandas — один из лучших инструментов для их анализа. Надеюсь, вам понравится прикладная направленность этих наборов данных.

ИСТОЧНИКИ ИНФОРМАЦИИ В ИНТЕРНЕТЕ

- Официальную документацию библиотеки pandas можно найти по адресу <https://pandas.pydata.org/docs>.
- В свободное время я создал технические видеокурсы на платформе Udemy. Найти их (20-часовой курс pandas и 60-часовой курс языка Python) можно по адресу <https://www.udemy.com/user/borispaskhaver>.
- Не стесняйтесь писать мне в Twitter (<https://twitter.com/borispaskhaver>) и LinkedIn (<https://www.linkedin.com/in/boris-paskhaver>).

Об авторе

Борис Пасхавер (Boris Paskhaver) — разработчик полного цикла, консультант и преподаватель из Нью-Йорка. На платформе дистанционного обучения Udey у него уже шесть курсов с более чем 140 часами видео, 300 тысячами слушателей, 20 тысячами отзывов и 1 миллионом минут ежемесячных просмотров. Прежде чем стать разработчиком программного обеспечения, Борис работал специалистом по анализу данных и системным администратором. В 2013 году он окончил Нью-Йоркский университет с двумя магистерскими дипломами — по экономике бизнеса и маркетингу.

Иллюстрация на обложке

Рисунок на обложке озаглавлен *Dame de Calais* («Леди из Кале»). Это иллюстрация из набора костюмов различных стран в книге Жака Грассе де Сан-Савье (Jacques Grasset de Saint-Sauveur) *Costumes de Différents Pays* («Наряды разных стран»), опубликованной во Франции в 1797 году. Все иллюстрации в издании прекрасно нарисованы и раскрашены вручную. Многообразие нарядов, приведенное Грассе де Сан-Савье, напоминает нам, насколько далеко отстояли друг от друга различные регионы мира всего 200 лет назад. Изолированные друг от друга, люди говорили на различных диалектах и языках. На улицах городов и в деревнях по одной только манере одеваться можно было легко определить, откуда человек, каким ремеслом занимается и каково его социальное положение.

Стили одежды с тех пор изменились, столь богатое разнообразие и самобытность различных регионов сошли на нет. Зачастую непросто отличить даже жителя одного континента от жителя другого, не говоря уже о городах, регионах и странах. Возможно, мы пожертвовали культурным многообразием в пользу большей вариативности личной жизни и определенно в пользу более разнообразной и динамичной жизни технологической.

В наше время, когда технические книги так мало отличаются друг от друга, издательство Manning изобретательно и инициативно радует читателя обложками книг, акцентируя удивительные различия в жизни регионов двухвековой давности, воплощенные в иллюстрациях Жака Грассе де Сан-Савье.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I

Основы *pandas*

Добро пожаловать! В этой части книги вы познакомитесь с основными принципами работы библиотеки *pandas* и ее двумя главными структурами данных: одномерным объектом **Series** и двумерным **DataFrame**. Глава 1 открывает книгу с анализа набора данных с помощью *pandas* для демонстрации ее возможностей. Далее мы перейдем к подробному изучению объекта **Series** в главах 2 и 3. Вы научитесь создавать объекты **Series** с нуля, импортировать их из внешних наборов данных, а также применять к ним массу различных математических, статистических и логических операций. В главе 4 вы познакомитесь с табличным объектом **DataFrame** и с разнообразными способами извлечения строк и столбцов, а также значений из содержащихся в них данных. Наконец, глава 5 посвящена извлечению поднаборов строк из объектов **DataFrame** путем применения различных логических критериев. А попутно мы рассмотрим восемь наборов данных, охватывающих все на свете: от театральных кассовых сборов и игроков НБА до покерменов.

Эта часть книги охватывает важнейшие стороны библиотеки *pandas*, основу, без которой невозможно работать с ней эффективно. Я изо всех сил постарался начать с самого нуля, с наименьших возможных «кирпичиков» и постепенно переходить к более крупным и сложным элементам. Следующие пять глав закладывают фундамент вашего мастерского владения *pandas*. Удачи!

1

Знакомство с библиотекой pandas

В этой главе

- ✓ Рост популярности науки о данных в XXI веке.
- ✓ История библиотеки pandas, предназначенной для анализа данных.
- ✓ Достоинства и недостатки библиотеки pandas и ее конкурентов.
- ✓ Анализ данных в Excel в сравнении с анализом данных с помощью языков программирования.
- ✓ Обзор возможности библиотеки pandas на рабочем примере.

Добро пожаловать в «Pandas в действии»! Pandas — библиотека для анализа данных, основанная на языке программирования Python. *Библиотека (пакет)* — набор кода для решения задач в определенной сфере деятельности. Библиотека pandas представляет собой набор инструментов для операций с данными: сортировки, фильтрации, очистки, удаления дубликатов, агрегирования, создания сводных таблиц и т. д. Являясь центром обширной экосистемы исследования данных, реализованной в среде языка Python, pandas хорошо сочетается с другими библиотеками для статистики, обработки естественного языка, машинного обучения, визуализации данных и многого другого.

В этой вводной главе мы изучим историю развития современных инструментов анализа данных. Увидим, как библиотека pandas выросла из проекта-хобби одного финансового аналитика в отраслевой стандарт, используемый такими компаниями, как Stripe, Google и J. P. Morgan. Нам предстоит сравнить эту библиотеку с ее конкурентами, включая Excel и язык R, обсудить различия между работой с языком программирования и визуальным приложением электронных таблиц. Наконец, мы проанализируем с помощью pandas реальный набор данных. Можете считать эту главу предварительным обзором всего того, чем вам предстоит овладеть в процессе чтения книги. Приступим!

1.1. ДАННЫЕ В XXI ВЕКЕ

«Теоретизировать без достаточных для того данных — грубейшая ошибка, — говорил Шерлок Холмс своему другу Джону Ватсону в «Скандале в Богемии», одном из классических рассказов сэра Артура Конан Дойла. — Незаметно для себя человек начинает подгонять факты к теории, а не формировать теорию по фактам».

Слова мудрого детектива не потеряли актуальности и сейчас, спустя более чем столетие после публикации рассказов Дойла. Трудно оспорить, что данные, информация играют все большую роль во всех аспектах нашей жизни. «Самый ценный ресурс в мире более не нефть, а данные», — заявил журнал *The Economist* в авторской колонке 2017 года. Данные — это *факты*, а факты жизненно важны для компаний, правительств, учреждений и отдельных лиц, решающих все более сложные задачи в нашем мире, в котором все взаимосвязано. По всем отраслям промышленности самые успешные в мире компании, от Facebook и Amazon до Netflix, называют информацию, данные самым ценным активом в своем портфолио. Генсек ООН Антониу Гутерреш назвал точные данные «жизненными соками правильной стратегии и принятия решений». Данные — движитель всего, от предпочтений в подборе фильмов до медицинского обслуживания, от сферы снабжения до инициатив по искоренению бедности. Успех компаний и даже стран в XXI веке будет зависеть от их способности добывать, агрегировать и анализировать данные.

1.2. ЗНАКОМСТВО С PANDAS

Количество систем, инструментов для работы с данными за последнее десятилетие резко выросло. На общем фоне библиотека pandas с открытым исходным кодом — одно из самых популярных сегодня решений для анализа данных и операций над ними. «С открытым исходным кодом» означает, что исходный

код библиотеки доступен всем для скачивания, использования, модификации и распространения. Ее лицензия дает пользователям больше прав, чем у проприетарного программного обеспечения, такого как Excel. Pandas бесплатна. Глобальная команда разработчиков поддерживает библиотеку на общественных началах, и вы можете найти ее полный исходный код на GitHub (<https://github.com/pandas-dev/pandas>).

Pandas сравнима с электронными таблицами Excel и работающим в браузере приложением «Google Таблицы». Во всех трех технологиях пользователь взаимодействует с таблицами, состоящими из строк и столбцов данных. Строка соответствует записи или, что эквивалентно, одному набору значений столбцов. Для приведения данных в желаемую форму к ним применяются различные преобразования.

На рис. 1.1 приведен пример преобразования набора данных. Специалист по анализу данных применяет операцию к набору данных слева, состоящему из пяти строк, чтобы получить в итоге набор данных из двух строк, показанный справа. Например, ему может быть необходимо выбрать строки, удовлетворяющие какому-либо условию, или удалить дублирующиеся строки из исходного набора данных.

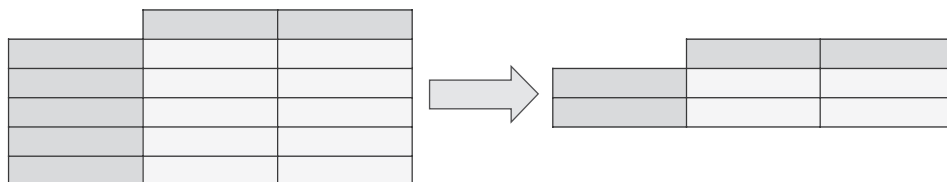


Рис. 1.1. Пример преобразования набора табличных данных

Уникальной библиотеку pandas делает баланс между возможностями обработки и продуктивностью пользователя. Поскольку для большинства ее операций применяются низкоуровневые языки программирования, например С, библиотека pandas способна эффективно преобразовывать наборы данных из миллионов строк за считанные миллисекунды. В то же время набор ее команд прост и интуитивно понятен. При использовании pandas можно с помощью очень короткого кода добиться очень многого.

На рис. 1.2 приведен пример кода pandas для импорта и сортировки набора данных в формате CSV. Не задумывайтесь пока над кодом, просто обратите внимание, что вся операция занимает всего две строки кода.

Pandas прекрасно работает с числами, текстом, датами, временем, пропущенными данными и многим другим. В этой книге в процессе работы более чем с 30 наборами данных мы увидим на практике ее универсальность.

Первая версия библиотеки pandas была разработана в 2008 году программистом Уэсом Маккини (Wes McKinney), занятым в нью-йоркской инвестиционной компании AQR Capital Management. Маккини, не удовлетворенный возможностями как Excel, так и языка статистического программирования R, искал инструмент для простого решения распространенных в сфере финансов задач, связанных с обработкой данных, в частности с их очисткой и агрегированием. Не найдя подходящего программного продукта, он решил создать его своими руками. В те времена Python еще не занимал такого лидирующего положения, как сейчас, но красота этого языка вдохновила Маккини использовать именно его для своей библиотеки. «Я полюбил Python за его лаконичность, — сказал Маккини в интервью *Quartz* (<http://mng.bz/w0Na>). — На Python можно выразить сложные идеи при помощи очень малого объема кода, причем кода довольно удобочитаемого».

In [2]:			<code>populations = pd.read_csv("populations.csv")</code>
			<code>populations.sort_values(by = "Population", ascending = False)</code>
Out[2]:			
	Country	Population	
144	China	1433783686	
21	India	1366417754	
156	United States	329064917	
76	Indonesia	270625568	
147	Pakistan	216565318	
79	Brazil	211049527	
6	Nigeria	200963599	
123	Bangladesh	163046161	

Рис. 1.2. Пример кода, импортирующего и сортирующего набор данных в pandas

С момента первого выпуска в декабре 2009 года популярность библиотеки pandas непрерывно росла. Количество ее пользователей сейчас оценивается в 5–10 миллионов¹. По состоянию на июнь 2021-го библиотеку pandas скачали с PyPI, централизованного онлайн-репозитория пакетов Python, более 750 миллионов раз (<https://pepy.tech/project/pandas>). Репозиторию ее кода на GitHub поставили более 30 000 звезд (звезда на этой платформе эквивалентна «лайку» в соцсетях). Доля вопросов по pandas на Stack Overflow, платформе-агрегаторе вопросов/ответов, непрерывно растет, демонстрируя тем самым рост интереса к ней со стороны пользователей.

¹ См. статью What's the future of the pandas library? («Будущее библиотеки pandas»), Data School, <https://www.dataschool.io/future-of-pandas>.

Я бы даже рискнул утверждать, что своим невероятно широким распространением язык Python во многом обязан библиотеке pandas. Ведь популярности Python достиг благодаря своему господству в сфере науки о данных, а в ней немалую роль играет библиотека pandas. Python сейчас чаще всего изучают в качестве первого языка программирования в колледжах и университетах. Согласно рейтингу ТЮБЕ популярности языков программирования по объему посвященного им трафика поисковых систем, Python стал самым популярным языком программирования 2021 года¹, оставив позади C и Java². А ведь при изучении pandas вы одновременно учите и Python — еще один плюс библиотеки.

1.2.1. Библиотека pandas по сравнению с визуальными приложениями электронных таблиц

Для работы с pandas нужен иной склад ума, чем для визуального приложения электронных таблиц, наподобие Excel. Программирование по своей природе связано с текстом, а не визуальными образами. Мы обмениваемся с компьютером информацией посредством команд, а не щелчков кнопкой мыши. Языки программирования делают меньше предварительных допущений о том, что хочет сделать программист, а потому не прощают ошибок. Они требуют явно и в деталях *указать*, что нужно сделать, без какой-либо неопределенности. Необходимо задавать правильные инструкции с правильными входными данными в правильном порядке, иначе программа работать не будет.

Вследствие таких более строгих требований кривая сложности обучения у pandas круче, чем у Excel или «Google Таблиц». Но не волнуйтесь, если у вас мало опыта работы с Python или программирования вообще! При работе с такими функциями Excel, как SUMIF и VLOOKUP, вы уже мыслите как программист. Идея та же самая: найти подходящую функцию и передать ей правильные входные данные в правильном порядке. Для библиотеки pandas требуется тот же набор навыков; разница лишь в том, что мы общаемся с компьютером более многословно.

Когда вы познакомитесь со всеми ее нюансами, библиотека pandas откроет для вас гораздо более широкие возможности, придаст гибкости в работе с данными. Помимо расширения спектра доступных процедур, программирование дает возможность их автоматизировать. Можно написать фрагмент кода один раз

¹ См. <https://www.tiobe.com/tiobe-index/python/>.

² Также Python выигрывал эту награду в 2018 и 2020 годах. — *Примеч. пер.*

и затем использовать его многократно во множестве файлов — идеальный инструмент для этих надоедливых ежедневных и еженедельных отчетов. Важно отметить, что пакет Excel включает в себя Visual Basic for Applications (VBA) — язык программирования, позволяющий автоматизировать процедуры работы с электронными таблицами. Но мне кажется, что Python изучить проще, чем VBA, и он пригоден не только для анализа данных, что делает его лучшим вложением ваших сил и времени.

У перехода с Excel на Python есть и дополнительные плюсы. Блокноты Jupyter — среда программирования, часто сочетающаяся с pandas, позволяет создавать более функциональные, интерактивные и исчерпывающие отчеты. Блокноты Jupyter состоят из ячеек, каждая из которых содержит фрагмент исполняемого кода. Специалисту по анализу данных можно сочетать эти ячейки с заголовками, графиками, описаниями, комментариями, картинками, видео, диаграммами и многим другим. Читатели этих блокнотов могут не только увидеть конечный результат, но и проследить шаг за шагом логику специалиста по анализу данных и увидеть, как он пришел к своим умозаключениям.

Еще одно достоинство pandas — унаследованная ею обширная экосистема науки о данных из языка Python. Pandas легко интегрируется с библиотеками для статистики, обработки естественного языка, машинного обучения, веб-скрапинга, визуализации данных и многого другого. Каждый год в языке появляются новые библиотеки. Эксперименты приветствуются. Нововведения возникают постоянно. Подобные ошибкоустойчивые инструменты у корпораций-конкурентов pandas частенько оказываются нереализованными или недоработанными по той причине, что последним не хватает поддержки обширного глобального сообщества участников.

По мере роста объемов обрабатываемых данных пользователи визуальных приложений электронных таблиц начинают испытывать трудности; возможности pandas в этом отношении намного шире, чем, к примеру, у Excel, они ограничиваются лишь объемом оперативной памяти и производительностью компьютера. На самых современных машинах pandas играючи справляется с многогигабайтными наборами данных из миллионов строк, особенно если разработчик хорошо умеет пользоваться возможностями оптимизации производительности. В сообщении из блога, описывающем ограничения библиотеки pandas, ее создатель Уэс Маккини пишет: «На сегодняшний день я применяю для pandas эмпирическое правило: оперативной памяти должно быть в 5–10 раз больше, чем объем набора данных» (<http://mng.bz/qeK6>).

Чтобы выбрать оптимальный инструмент для работы, необходимо, в частности, определиться, что для вашей организации и вашего проекта значат такие понятия, как *анализ данных* и *большие данные*. Электронные таблицы Excel,

используемые для работы примерно 750 миллионами пользователей по всему миру, ограничивают размер таблицы 1 048 576 строками данных¹. Для некоторых специалистов по анализу данных миллиона строк с лихвой хватает для любого отчета; для других миллион строк — мелочь.

Я рекомендую рассматривать pandas не как идеальное решение для анализа данных, а как продукт, обладающий большими возможностями, который необходимо использовать в сочетании с другими современными технологиями. Excel по-прежнему остается замечательным вариантом для быстрых простых операций над данными. Приложения электронных таблиц обычно делают определенные допущения о намерениях пользователя, поэтому импорт CSV-файла или сортировка столбца из 100 значений требует лишь нескольких щелчков мышью. При таких простых задачах у pandas нет никаких преимуществ (хотя она более чем способна на их решение). Но чем, скажите, вы воспользуетесь, чтобы очистить текстовые значения в двух наборах данных по десять миллионов строк каждый, удалить дубликаты записей, соединить их и повторить всю эту логику для 100 пакетов файлов? В подобных сценариях легче и быстрее будет выполнить работу с помощью Python и библиотеки pandas.

1.2.2. Pandas по сравнению с конкурентами

Энтузиасты науки о данных часто сравнивают pandas с языком программирования с открытым исходным кодом R и проприетарным набором программного обеспечения SAS. У каждого из этих решений есть свои сторонники.

R — специализированный язык, ориентированный на статистические расчеты, в то время как Python — универсальный язык, применяемый в различных областях науки и техники. Неудивительно, что эти два языка привлекают пользователей, специализирующихся в своих областях. Хэдли Викхэм (Hadley Wickham), видный разработчик из сообщества R, создавший набор пакетов для исследования данных tidyverse, советует пользователям считать эти два языка скорее компаньонами, а не соперниками. «Они существуют независимо друг от друга, и оба великолепны по-своему», — сказал он в интервью Quartz (<http://mng.bz/3v9V>). «Я наблюдаю следующую закономерность: в компаниях команды исследователей данных используют R, а команды проектирования данных — Python. Специалисты по Python обычно выходят из разработчиков и чувствуют себя уверенно в программировании. Пользователям R очень нравится сам язык R, но у них не хватает аргументов, чтобы спорить с проектировщиками данных». У каждого из этих языков есть свои возможности, которых нет у другого, но,

¹ См.: *Patrizio Andy*. Excel: Your entry into the world of data analytics, Computer World, <http://mng.bz/qe6r>.

когда речь идет о распространенных задачах анализа данных, они оказываются практически равны. Разработчики и исследователи данных просто тяготеют к тому, что знают лучше всего и к чему привыкли.

SAS — набор взаимодополняющих программных утилит для статистики, интеллектуального анализа данных, эконометрики и тому подобного — представляет собой коммерческий программный продукт, разработанный компанией SAS Institute из Северной Каролины. Стоимость годовой подписки на него различается в зависимости от выбранного комплекта программного обеспечения. Преимущества поддерживаемого корпорацией программного продукта включают технологическое и визуальное единообразие утилит, документацию без ошибок и дорожную карту развития продукта, ориентированную на нужды клиентов. Технологии с открытым исходным кодом, такие как pandas, придерживаются подхода «куча мала», разработчики стремятся удовлетворить потребности свои и других разработчиков, при этом иногда упуская тенденции рынка.

Существуют некоторые технологии, определенные возможности которых совпадают с возможностями pandas, но служат принципиально другим целям. Один из примеров — SQL. *SQL* (Structured Query Language, структурированный язык запросов) предназначен для взаимодействия с реляционными базами данных. *Реляционная база данных* (relational database) состоит из таблиц данных, связанных общими ключами. С помощью SQL можно производить основные операции над данными, например извлечение столбцов из таблиц и фильтрацию строк по критерию, но его функциональность обширнее и так или иначе связана с управлением данными. Базы данных созданы для *хранения* данных; анализ данных — лишь побочный сценарий использования. SQL позволяет создавать новые таблицы, обновлять значения уже существующих записей, удалять существующие записи и т. д. По сравнению с ним библиотека pandas полностью ориентирована на анализ данных: статистические вычисления, «выпас», преобразование данных (wrangling, munging), слияние данных и многое другое. В типичной рабочей среде эти два инструмента часто взаимно дополняют друг друга. Специалист по анализу данных может, например, использовать SQL для извлечения начального кластера данных, а затем воспользоваться библиотекой pandas для операций над ними.

Подытожим: библиотека pandas не единственный инструмент, но она обладает большими возможностями и представляет собой популярный и полезный продукт для решения большинства задач анализа данных. И опять же Python поистине блистает со своим упором на лаконичность и производительность. Как отметил его создатель, Гвидо ван Россум (Guido van Rossum): «Написание кода Python должно доставлять радость своими короткими, лаконичными, удобочитаемыми классами, выражающими большой объем операций в маленьком

количестве кода» (<http://mng.bz/7jo7>). Библиотека pandas полностью соответствует всем этим характеристикам и представляет собой следующий этап обработки информации, прекрасно подходящий для специалистов по ее анализу, желающих улучшить свои навыки программирования с помощью обладающего большими возможностями современного набора инструментов анализа данных.

1.3. ОБЗОР БИБЛИОТЕКИ PANDAS

Лучше всего открывать для себя возможности библиотеки pandas на практике. Пройдемся по возможностям библиотеки на примере анализа набора данных 700 самых кассовых фильмов всех времен. Мне кажется, вы будете приятно удивлены, насколько интуитивно понятен синтаксис pandas даже для программистов-новичков.

По мере чтения изложенного ниже материала этой главы старайтесь не слишком вникать в примеры кода; вам даже не обязательно их копировать. Наша цель сейчас — взглянуть на укрупненную картину возможностей и функциональности библиотеки pandas. Думайте о том, на *что* способна эта библиотека; позднее мы рассмотрим подробнее, *как это возможно*.

Для написания кода в этой книге в качестве среды программирования используются блокноты Jupyter. Если вам нужна помощь в настройке pandas и блокнотов Jupyter на вашем компьютере — загляните в приложение А. Скачать все наборы данных и готовые блокноты Jupyter можно по адресу <https://www.github.com/paskhaver/pandas-in-action>.

1.3.1. Импорт набора данных

Приступим! Во-первых, создадим новый блокнот Jupyter в том же каталоге, что и файл `movies.csv`, затем импортируем библиотеку pandas для доступа ко всем ее возможностям:

```
In [1] import pandas as pd
```

Поле слева от кода (с номером [1] в предыдущем примере) отражает порядок выполнения ячеек, отсчитываемый от запуска или перезапуска блокнота Jupyter. Можно выполнять ячейки в произвольном порядке и даже выполнять одну ячейку несколько раз.

Советую вам в ходе чтения данной книги экспериментировать, выполняя различные фрагменты кода в своих Jupyter. Так что не обращайте внимания, если счетчики выполнения у вас будут отличаться от приведенных в тексте.

Наши данные хранятся в одном файле `movies.csv`. Файлы CSV (comma-separated values — значения, отделенные друг от друга запятыми) представляют собой файлы с открытым текстом, в которых строки данных разделяются символом переноса строки, а значения внутри строк — запятыми. Первая строка файла содержит названия столбцов данных. Вот первые три строки нашего файла:

```
Rank,Title,Studio,Gross,Year
1,Avengers: Endgame,Buena Vista,"$2,796.30",2019
2,Avatar,Fox,"$2,789.70",2009
```

Первая строка перечисляет названия пяти столбцов в наборе данных: `Rank` (Позиция), `Title` (Название), `Studio` (Студия), `Gross` (Кассовые сборы) и `Year` (Год). Во второй строке содержится первая запись, то есть данные для первого фильма. Позиция у этого фильма — 1, название — "Avengers: Endgame", студия — "Buena Vista", кассовые сборы — "\$2,796.30", а год — 2019. В следующей строке содержатся значения для следующего фильма и т. д. во всех 750 с лишним строках файла.

Библиотека `pandas` может импортировать разнообразные типы файлов, каждому из которых соответствует функция импорта на верхнем уровне библиотеки. Функция в библиотеке `pandas` эквивалентна функции в Excel и представляет собой команду, которую мы отдаем библиотеке или какой-либо сущности из нее. В данном сценарии мы воспользовались для импорта файла `movies.csv` функцией `read_csv`:

```
In [2] pd.read_csv("movies.csv")
```

```
Out [2]
```

	Rank	Title	Studio	Gross	Year
0	1	Avengers: Endgame	Buena Vista	\$2,796.30	2019
1	2	Avatar	Fox	\$2,789.70	2009
2	3	Titanic	Paramount	\$2,187.50	1997
3	4	Star Wars: The Force Awakens	Buena Vista	\$2,068.20	2015
4	5	Avengers: Infinity War	Buena Vista	\$2,048.40	2018
...
777	778	Yogi Bear	Warner Brothers	\$201.60	2010
778	779	Garfield: The Movie	Fox	\$200.80	2004
779	780	Cats & Dogs	Warner Brothers	\$200.70	2001
780	781	The Hunt for Red October	Paramount	\$200.50	1990
781	782	Valkyrie	MGM	\$200.30	2008

```
782 rows x 5 columns
```

Библиотека `pandas` импортирует содержимое CSV-файла в объект `DataFrame`, своего рода контейнер для хранения данных. Различные объекты оптимизируются

для различных типов данных, и взаимодействовать с ними тоже надо по-разному. Библиотека pandas использует один тип объектов (**DataFrame**) для хранения наборов данных с несколькими столбцами и другой тип (**Series**) для хранения наборов данных из одного столбца. **DataFrame** можно сравнить с многостолбцовой таблицей в Excel.

Чтобы не загромождать экран, pandas отображает только первые и последние пять строк **DataFrame**¹. Пропущенные данные отмечаются строкой с многоточием (...).

Наш **DataFrame** состоит из пяти столбцов (**Rank**, **Title**, **Studio**, **Gross**, **Year**) и индекса. Индекс — это столбец слева от **DataFrame**, он содержит числа, расположенные в порядке возрастания. Метки индекса служат идентификаторами строк данных. Индексом **DataFrame** может служить любой столбец. Если не указать библиотеке pandas явным образом, какой столбец использовать, она сгенерирует числовой индекс, начинающийся с нуля.

Какой столбец подходит в качестве индекса? Тот, значения из которого могут выступать в роли первичных идентификаторов строк (ссылок на строки). Из наших пяти столбцов на эту роль лучше всего подходят **Rank** и **Title**. Заменяем сгенерированный автоматически числовой индекс на значения из столбца **Title**. Сделать это можно непосредственно во время импорта CSV:

```
In [3] pd.read_csv("movies.csv", index_col = "Title")
```

```
Out [3]
```

Title	Rank	Studio	Gross	Year
Avengers: Endgame	1	Buena Vista	\$2,796.30	2019
Avatar	2	Fox	\$2,789.70	2009
Titanic	3	Paramount	\$2,187.50	1997
Star Wars: The Force Awakens	4	Buena Vista	\$2,068.20	2015
Avengers: Infinity War	5	Buena Vista	\$2,048.40	2018
...
Yogi Bear	778	Warner Brothers	\$201.60	2010
Garfield: The Movie	779	Fox	\$200.80	2004
Cats & Dogs	780	Warner Brothers	\$200.70	2001
The Hunt for Red October	781	Paramount	\$200.50	1990
Valkyrie	782	MGM	\$200.30	2008

782 rows × 4 columns

¹ Количество отображаемых библиотекой pandas строк **DataFrame** можно настраивать при помощи функции `set_option()`. Например, чтобы отображать десять первых и десять последних строк, выполните команду: `pd.set_option('display.max_rows', 20)`. В книге настройка количества отображаемых строк упоминается и описывается в нескольких разделах. — *Примеч. пер.*

Далее присваиваем этот `DataFrame` переменной `movies`, чтобы можно было ссылаться на него в других местах программы. *Переменная* — назначаемое пользователем имя какого-либо объекта в программе:

```
In [4] movies = pd.read_csv("movies.csv", index_col = "Title")
```

Узнать больше о переменных вы можете из приложения Б.

1.3.2. Операции над объектами `DataFrame`

Объект `DataFrame` можно рассматривать с множества различных ракурсов. Можно, например, извлечь несколько строк из его начала:

```
In [5] movies.head(4)
```

Out [5]

Title	Rank	Studio	Gross	Year
Avengers: Endgame	1	Buena Vista	\$2,796.30	2019
Avatar	2	Fox	\$2,789.70	2009
Titanic	3	Paramount	\$2,187.50	1997
Star Wars: The Force Awakens	4	Buena Vista	\$2,068.20	2015

Или, наоборот, заглянуть в конец набора данных:

```
In [6] movies.tail(6)
```

Out [6]

Title	Rank	Studio	Gross	Year
21 Jump Street	777	Sony	\$201.60	2012
Yogi Bear	778	Warner Brothers	\$201.60	2010
Garfield: The Movie	779	Fox	\$200.80	2004
Cats & Dogs	780	Warner Brothers	\$200.70	2001
The Hunt for Red October	781	Paramount	\$200.50	1990
Valkyrie	782	MGM	\$200.30	2008

Можно узнать, сколько строк содержит объект `DataFrame`:

```
In [7] len(movies)
```

Out [7] 782

Можно также запросить у `pandas` число строк и столбцов в `DataFrame`. Наш текущий набор данных содержит 782 строки и 4 столбца.

```
In [8] movies.shape
```

Out [8] (782, 4)

42 Часть I. Основы pandas

Можно выяснить общее количество ячеек:

```
In [9] movies.size
```

```
Out [9] 3128
```

Можно запросить типы данных наших четырех столбцов. В следующих ниже выведенных результатах `int64` означает целочисленный столбец, а `object` — текстовый столбец:

```
In [10] movies.dtypes
```

```
Out [10]
```

```
Rank      int64
Studio     object
Gross      object
Year       int64
dtype: object
```

Можно извлечь строку из набора данных по ее порядковому номеру, то есть индексу. В большинстве языков программирования отсчет индексов начинается с 0. Следовательно, если нужно извлечь 500-й фильм в наборе данных, надо выбрать строку с индексом 499:

```
In [11] movies.iloc[499]
```

```
Out [11] Rank      500
         Studio     Fox
         Gross    $288.30
         Year      2018
         Name: Maze Runner: The Death Cure, dtype: object
```

Pandas вернула тут новый объект **Series** — одномерный маркированный массив значений, нечто вроде столбца данных с идентификаторами для каждой строки. Обратите внимание, что метки объекта **Series** (`Rank`, `Studio`, `Gross`, `Year`) соответствуют четырем столбцам из объекта **DataFrame** `movies`. Таким образом, pandas изменила представление значений исходной строки.

Можно также воспользоваться меткой индекса для обращения к строке объекта **DataFrame**. Напомним, что индекс нашего **DataFrame** содержит названия фильмов. Извлечем значения для строки любимой всеми мелодрамы *Forrest Gump*. В следующем примере строка извлекается по метке индекса, а не числовой позиции:

```
In [12] movies.loc["Forrest Gump"]
```

```
Out [12] Rank      119
         Studio    Paramount
```

```
Gross      $677.90
Year       1994
Name: Forrest Gump, dtype: object
```

Метки индекса могут повторяться. Например, два фильма из нашего `DataFrame` называются "101 Dalmatians" (оригинальный, 1961 года, и ремейк 1996-го):

```
In [13] movies.loc["101 Dalmatians"]
```

```
Out [13]
```

Title	Rank	Studio	Gross	Year
101 Dalmatians	425	Buena Vista	\$320.70	1996
101 Dalmatians	708	Buena Vista	\$215.90	1961

И хотя библиотека `pandas` допускает повторы, я рекомендую стремиться к уникальным меткам индекса. Уникальный набор меток позволяет `pandas` быстрее находить и извлекать конкретные строки.

Фильмы в рассматриваемом CSV-файле отсортированы по столбцу `Rank`. Как быть, если необходимо получить пять наиболее свежих фильмов? Можно отсортировать `DataFrame` по значениям другого столбца, в данном случае — `Year`:

```
In [14] movies.sort_values(by = "Year", ascending = False).head()
```

```
Out [14]
```

Title	Rank	Studio	Gross	Year
Avengers: Endgame	1	Buena Vista	2796.3	2019
John Wick: Chapter 3 - Parab...	458	Lionsgate	304.7	2019
The Wandering Earth	114	China Film Corporation	699.8	2019
Toy Story 4	198	Buena Vista	519.8	2019
How to Train Your Dragon: Th...	199	Universal	519.8	2019

Можно также сортировать объекты `DataFrame` по значениям из нескольких столбцов. Отсортируем `movies` сначала по столбцу `Studio`, а затем — по столбцу `Year`. И получим фильмы, отсортированные в алфавитном порядке по студии и году выхода:

```
In [15] movies.sort_values(by = ["Studio", "Year"]).head()
```

```
Out [15]
```

Title	Rank	Studio	Gross	Year
The Blair Witch Project	588	Artisan	\$248.60	1999
101 Dalmatians	708	Buena Vista	\$215.90	1961
The Jungle Book	755	Buena Vista	\$205.80	1967
Who Framed Roger Rabbit	410	Buena Vista	\$329.80	1988
Dead Poets Society	636	Buena Vista	\$235.90	1989

44 Часть I. Основы pandas

Можно также отсортировать индекс для отображения фильмов в алфавитном порядке названий:

```
In [16] movies.sort_index().head()
```

```
Out [16]
```

Title	Rank	Studio	Gross	Year
10,000 B.C.	536	Warner Brothers	\$269.80	2008
101 Dalmatians	708	Buena Vista	\$215.90	1961
101 Dalmatians	425	Buena Vista	\$320.70	1996
2 Fast 2 Furious	632	Universal	\$236.40	2003
2012	93	Sony	\$769.70	2009

Все выполненные нами операции возвращали *новые*, созданные ими самими объекты **DataFrame**. Библиотека pandas не меняла исходный объект **movies** из CSV-файла. То, что эти операции не разрушают данные, очень удобно и служит стимулом для экспериментов. Всегда можно убедиться, что результат правильный, прежде чем его фиксировать в качестве окончательного.

1.3.3. Подсчет значений в Series

Попробуем более сложный вариант анализа. Пусть необходимо выяснить, у какой киностудии больше всего кассовых фильмов. Для решения этой задачи нужно подсчитать, сколько раз каждая студия встречается в столбце **Studio**.

Можно извлечь отдельный столбец данных из **DataFrame** в виде **Series**. Обратите внимание, что библиотека pandas сохраняет индекс **DataFrame**, то есть названия фильмов, в полученном объекте **Series**:

```
In [17] movies["Studio"]
```

```
Out [17] Title
```

Avengers: Endgame	Buena Vista
Avatar	Fox
Titanic	Paramount
Star Wars: The Force Awakens	Buena Vista
Avengers: Infinity War	Buena Vista
...	...
Yogi Bear	Warner Brothers
Garfield: The Movie	Fox
Cats & Dogs	Warner Brothers
The Hunt for Red October	Paramount
Valkyrie	MGM

Name: Studio, Length: 782, dtype: object

При большом количестве строк в **Series** pandas сокращает набор данных и отображает только пять первых и пять последних строк.

Мы отделили столбец `Studio` и можем теперь подсчитать число вхождений каждого из уникальных значений кинокомпаний. Ограничим выборку десятью наиболее успешными киностудиями:

```
In [18] movies["Studio"].value_counts().head(10)
```

```
Out [18] Warner Brothers    132
         Buena Vista       125
         Fox               117
         Universal         109
         Sony              86
         Paramount        76
         Dreamworks        27
         Lionsgate         21
         New Line          16
         MGM               11
         Name: Studio, dtype: int64
```

Приведенное выше возвращаемое значение — еще один объект `Series`! В нем библиотека pandas использует студии из столбца `Studio` в качестве меток индекса, а соответствующие им значения количества фильмов — как значения `Series`.

1.3.4. Фильтрация столбца по одному или нескольким критериям

Нередко бывает необходимо извлечь подмножество строк на основе одного или нескольких критериев. Для этой цели в Excel служит инструмент **Фильтр**.

Допустим, что нам нужно найти фильмы, выпущенные Universal Studios. Решить эту задачу в pandas можно с помощью одной строки кода:

```
In [19] movies[movies["Studio"] == "Universal"]
```

```
Out [19]
```

Title	Rank	Studio	Gross	Year
-----	-----	-----	-----	-----
Jurassic World	6	Universal	\$1,671.70	2015
Furious 7	8	Universal	\$1,516.00	2015
Jurassic World: Fallen Kingdom	13	Universal	\$1,309.50	2018
The Fate of the Furious	17	Universal	\$1,236.00	2017
Minions	19	Universal	\$1,159.40	2015
...
The Break-Up	763	Universal	\$205.00	2006
Everest	766	Universal	\$203.40	2015
Patch Adams	772	Universal	\$202.30	1998
Kindergarten Cop	775	Universal	\$202.00	1990
Straight Outta Compton	776	Universal	\$201.60	2015

```
109 rows x 4 columns
```

46 Часть I. Основы pandas

Можно также присвоить условие фильтрации переменной, чтобы читатели кода понимали контекст:

```
In [20] released_by_universal = (movies["Studio"] == "Universal")
        movies[released_by_universal].head()
```

Out [20]

Title	Rank	Studio	Gross	Year
Jurassic World	6	Universal	\$1,671.70	2015
Furious 7	8	Universal	\$1,516.00	2015
Jurassic World: Fallen Kingdom	13	Universal	\$1,309.50	2018
The Fate of the Furious	17	Universal	\$1,236.00	2017
Minions	19	Universal	\$1,159.40	2015

Есть возможность фильтровать строки `DataFrame` и по нескольким критериям. В примере ниже мы найдем все фильмы, выпущенные Universal Studios в 2015 году:

```
In [21] released_by_universal = movies["Studio"] == "Universal"
        released_in_2015 = movies["Year"] == 2015
        movies[released_by_universal & released_in_2015]
```

Out [21]

Title	Rank	Studio	Gross	Year
Jurassic World	6	Universal	\$1,671.70	2015
Furious 7	8	Universal	\$1,516.00	2015
Minions	19	Universal	\$1,159.40	2015
Fifty Shades of Grey	165	Universal	\$571.00	2015
Pitch Perfect 2	504	Universal	\$287.50	2015
Ted 2	702	Universal	\$216.70	2015
Everest	766	Universal	\$203.40	2015
Straight Outta Compton	776	Universal	\$201.60	2015

Рассмотренный пример включает строки, удовлетворяющие обоим условиям. Мы также можем осуществить выборку фильмов, удовлетворяющих хотя бы одному из этих условий: выпущенные Universal Studios *или* выпущенные в 2015 году. В итоге получается более длинный объект `DataFrame`, поскольку вероятность удовлетворить только одному условию выше, чем сразу двум:

```
In [22] released_by_universal = movies["Studio"] == "Universal"
        released_in_2015 = movies["Year"] == 2015
        movies[released_by_universal | released_in_2015]
```

Out [22]

Title	Rank	Studio	Gross	Year
Star Wars: The Force Awakens	4	Buena Vista	\$2,068.20	2015
Jurassic World	6	Universal	\$1,671.70	2015

Furious 7	8	Universal	\$1,516.00	2015
Avengers: Age of Ultron	9	Buena Vista	\$1,405.40	2015
Jurassic World: Fallen Kingdom	13	Universal	\$1,309.50	2018
...
The Break-Up	763	Universal	\$205.00	2006
Everest	766	Universal	\$203.40	2015
Patch Adams	772	Universal	\$202.30	1998
Kindergarten Cop	775	Universal	\$202.00	1990
Straight Outta Compton	776	Universal	\$201.60	2015

140 rows × 4 columns

Библиотека pandas позволяет фильтровать `DataFrame` и другими способами. Можно выбрать, скажем, значения столбцов больше или меньше конкретного значения. В следующем примере мы выбираем фильмы, выпущенные до 1975 года:

```
In [23] before_1975 = movies["Year"] < 1975
        movies[before_1975]
```

Out [23]

Title	Rank	Studio	Gross	Year
The Exorcist	252	Warner Brothers	\$441.30	1973
Gone with the Wind	288	MGM	\$402.40	1939
Bambi	540	RKO	\$267.40	1942
The Godfather	604	Paramount	\$245.10	1972
101 Dalmatians	708	Buena Vista	\$215.90	1961
The Jungle Book	755	Buena Vista	\$205.80	1967

Можно также задать диапазон для значений. Например, извлечь фильмы, выпущенные с 1983 по 1986 год:

```
In [24] mid_80s = movies["Year"].between(1983, 1986)
        movies[mid_80s]
```

Out [24]

Title	Rank	Studio	Gross	Year
Return of the Jedi	222	Fox	\$475.10	1983
Back to the Future	311	Universal	\$381.10	1985
Top Gun	357	Paramount	\$356.80	1986
Indiana Jones and the Temple of Doom	403	Paramount	\$333.10	1984
Crocodile Dundee	413	Paramount	\$328.20	1986
Beverly Hills Cop	432	Paramount	\$316.40	1984
Rocky IV	467	MGM	\$300.50	1985
Rambo: First Blood Part II	469	TriStar	\$300.40	1985
Ghostbusters	485	Columbia	\$295.20	1984
Out of Africa	662	Universal	\$227.50	1985

Можно воспользоваться для фильтрации строк индексом `DataFrame`. Например, код ниже преобразует все названия фильмов в индексе в нижний регистр и находит фильмы, в названии которых содержится слово `dark`:

```
In [25] has_dark_in_title = movies.index.str.lower().str.contains("dark")
        movies[has_dark_in_title]
```

Out [25]

Title	Rank	Studio	Gross	Year
transformers: dark of the moon	23	Paramount	\$1,123.80	2011
the dark knight rises	27	Warner Brothers	\$1,084.90	2012
the dark knight	39	Warner Brothers	\$1,004.90	2008
thor: the dark world	132	Buena Vista	\$644.60	2013
star trek into darkness	232	Paramount	\$467.40	2013
fifty shades darker	309	Universal	\$381.50	2017
dark shadows	600	Warner Brothers	\$245.50	2012
dark phoenix	603	Fox	\$245.10	2019

Обратите внимание, что библиотека `pandas` находит все фильмы, в названиях которых содержится слово `dark`, вне зависимости от того, в каком именно месте (позиции) в названии оно встречается.

1.3.5. Группировка данных

Следующая задача — самая сложная из встретившихся нам к данному моменту. Нас интересует, у какой студии самые высокие кассовые сборы из всех. Давайте агрегируем значения из столбца `Gross` по студии.

Первая проблема: значения в столбце `Gross` хранятся в виде текста, а не чисел. Библиотека `pandas` импортирует значения столбца как текст, чтобы сохранить символы доллара и запятые из исходного CSV-файла. Можно преобразовать значения столбца в десятичные числа, но для этого необходимо удалить оба упомянутых символа. В примере ниже мы заменяем все вхождения "\$" и ",", пустыми строками. Данная операция аналогична функции Найти и заменить в Excel:

```
In [26] movies["Gross"].str.replace(
        "$", "", regex = False
    ).str.replace(",", "", regex = False)
```

```
Out [26] Title
Avengers: Endgame      2796.30
Avatar                 2789.70
Titanic                2187.50
Star Wars: The Force Awakens 2068.20
Avengers: Infinity War 2048.40
...
Yogi Bear              201.60
```



```

Garfield: The Movie          200.80
Cats & Dogs                  200.70
The Hunt for Red October     200.50
Valkyrie                     200.30
Name: Gross, Length: 782, dtype: object

```

Удалив символы, мы можем преобразовать значения столбца `Gross` из текста в числа с плавающей точкой:

```

In [27] (
    movies["Gross"]
    .str.replace("$", "", regex = False)
    .str.replace(",", "", regex = False)
    .astype(float)
)

Out [27] Title
Avengers: Endgame          2796.3
Avatar                     2789.7
Titanic                    2187.5
Star Wars: The Force Awakens 2068.2
Avengers: Infinity War     2048.4
...
Yogi Bear                  201.6
Garfield: The Movie        200.8
Cats & Dogs                 200.7
The Hunt for Red October    200.5
Valkyrie                   200.3
Name: Gross, Length: 782, dtype: float64

```

Опять же эти операции временные и не модифицируют исходного `Series Gross`. Во всех предыдущих примерах библиотека `pandas` создавала копию исходной структуры данных, выполняла операцию и возвращала новый объект. В следующем примере мы явным образом заменяем столбец `Gross` в `movies` новым столбцом, содержащим числа с плавающей точкой. Теперь преобразование зафиксировано в наборе как результат всех выполненных операций:

```

In [28] movies["Gross"] = (
    movies["Gross"]
    .str.replace("$", "", regex = False)
    .str.replace(",", "", regex = False)
    .astype(float)
)

```

Наше преобразование типа данных открывает возможности для других вычислений и операций. В следующем примере вычисляются средние кассовые сборы по всем фильмам:

```

In [29] movies["Gross"].mean()

Out [29] 439.0308184143222

```

50 Часть I. Основы pandas

Возвращаемся к изначальной задаче — вычислению кассовых сборов, агрегированных по студиям. В первую очередь необходимо идентифицировать студии и разбить на подмножества фильмы (строки), относящиеся к каждой из них. Этот процесс называется *группировкой*. Приведенный ниже код служит для группирования строк объекта `DataFrame` по значениям из столбца `Studio`:

```
In [30] studios = movies.groupby("Studio")
```

Можно попросить `pandas` подсчитать количество фильмов каждой из студий:

```
In [31] studios["Gross"].count().head()
```

```
Out [31] Studio
          Artisan          1
        Buena Vista      125
           CL            1
    China Film Corporation    1
        Columbia          5
      Name: Gross, dtype: int64
```

Приведенные выше результаты отсортированы по названию студии. Можно вместо этого отсортировать `Series` по количеству фильмов в порядке убывания:

```
In [32] studios["Gross"].count().sort_values(ascending = False).head()
```

```
Out [32] Studio
    Warner Brothers    132
        Buena Vista    125
           Fox         117
        Universal     109
           Sony        86
      Name: Gross, dtype: int64
```

Далее просуммируем значения из столбца `Gross` по студиям. `Pandas` распознает подмножество фильмов, относящихся к каждой из студий, извлекает из строки соответствующие значения столбца `Gross` и суммирует их:

```
In [33] studios["Gross"].sum().head()
```

```
Out [33] Studio
          Artisan          248.6
        Buena Vista      73585.0
           CL           228.1
    China Film Corporation    699.8
        Columbia      1276.6
      Name: Gross, dtype: float64
```

Опять же библиотека `pandas` сортирует результаты по названию студии. Мы хотим найти студии с самыми высокими кассовыми сборами, так что отсортируем

значения `Series` в порядке убывания сборов. Вот пять студий с наибольшими кассовыми сборами:

```
In [34] studios["Gross"].sum().sort_values(ascending = False).head()
```

```
Out [34] Studio
          Buena Vista      73585.0
          Warner Brothers  58643.8
          Fox             50420.8
          Universal       44302.3
          Sony            32822.5
          Name: Gross, dtype: float64
```

С помощью всего нескольких строк кода мы смогли извлечь из этого непростого набора данных немало интересной информации. Например, у студии Warner Brothers в этом списке больше фильмов, чем у Buena Vista, зато совокупные кассовые сборы у Buena Vista выше. А значит, средние кассовые сборы фильмов студии Buena Vista выше, чем у фильмов студии Warner Brothers.

Мы затронули лишь верхушку айсберга возможностей библиотеки pandas. Надеюсь, эти примеры показали все множество разнообразных способов преобразования данных и операций над ними в этой замечательной библиотеке. Нам предстоит обсудить все коды из этой главы намного подробнее в последующих главах книги. А в главе 2 займемся одним из основных «кирпичиков» библиотеки pandas: объектом `Series`.

РЕЗЮМЕ

- Pandas — библиотека для анализа данных, основанная на языке программирования Python.
- Pandas превосходно справляется со сложными операциями над большими наборами данных и отличается сжатым синтаксисом.
- В качестве альтернатив использованию библиотеки pandas можно рассматривать визуальное приложение электронных таблиц Excel, статистический язык программирования R и комплект программ SAS.
- Программирование требует от специалиста иного набора навыков, чем работа с Excel или «Google Таблицами».
- Pandas способна импортировать множество разнообразных форматов файлов. Один из популярных форматов — CSV, в нем строки разделяются разрывами строк, а значения внутри строк — запятыми.
- `DataFrame` — основная структура данных библиотеки pandas. По существу, она представляет собой таблицу данных с несколькими столбцами.

- **Series** представляет собой одномерный маркированный массив. Его можно рассматривать просто как отдельный столбец данных.
- Обращаться к строкам в **Series** и **DataFrame** можно по номеру строки или метке индекса.
- Можно сортировать **DataFrame** по значениям одного или нескольких столбцов.
- Для извлечения подмножеств данных из **DataFrame** можно использовать логические условия.
- Можно группировать строки **DataFrame** по значениям какого-либо столбца и затем производить над полученными группами операции агрегирования, например суммирование.

2

Объект Series

В этой главе

- ✓ Создание объектов Series из списков, ассоциативных массивов, кортежей и других типов и источников данных.
- ✓ Создание пользовательского индекса для Series.
- ✓ Обращение к атрибутам и вызов методов Series.
- ✓ Математические операции над одним или несколькими объектами Series.
- ✓ Передача объектов Series встроенным функциям языка Python.

Одна из основных структур данных библиотеки pandas, **Series**, представляет собой одномерный маркированный массив однородных данных. *Массив* (array) — это упорядоченный набор значений, сравнимый со списком языка Python. Термин «*однородный*» означает, что все значения — одного и того же типа (например, все целые числа или булевы значения).

Каждому значению **Series** в Pandas соответствует *метка* (label) — идентификатор, с помощью которого можно сослаться на это значение. Кроме того, значениям **Series** присуща *упорядоченность* — позиция по порядку. Отсчет их номеров начинается с 0; первому значению объекта **Series** соответствует позиция 0, второму — 1 и т. д. **Series** — одномерная структура данных, поскольку для обращения к значениям достаточно указать одну координату: метку или позицию.

Series сочетает в себе и расширяет лучшие возможности нативных структур данных Python. Как и список, он содержит значения в определенном порядке. Как в ассоциативном массиве, каждому значению в нем соответствует ключ/метка. Работая с **Series**, мы можем использовать все преимущества обеих этих структур данных плюс задействовать более чем 180 методов для различных операций над данными.

В этой главе вы познакомитесь со спецификой объектов **Series**, научитесь вычислять сумму и среднее значений **Series**, применять ко всем значениям **Series** математические операции и делать многое другое. Объект **Series**, как основной «строительный блок» библиотеки pandas, — прекрасная отправная точка для знакомства с этой библиотекой.

2.1. ОБЗОР SERIES

Создадим несколько объектов **Series**. Начнем с импорта пакетов pandas и NumPy с помощью ключевого слова `import`; второй из этих библиотек мы воспользуемся в подразделе 2.1.4. Для pandas и numpy принято использовать псевдонимы `pd` и `np`. Назначить псевдоним импортируемой библиотеке можно с помощью ключевого слова `as`:

```
In [1] import pandas as pd
      import numpy as np
```

Пространство имен `pd` включает высокоуровневые экспорты пакета pandas — набор из более чем 100 классов, функций, исключений, констант и т. д. Больше информации о них можно получить из приложения Б.

`pd` можно рассматривать как своего рода вестибюль библиотеки — прихожую, через которую можно получить доступ к возможностям pandas. Экспорты библиотеки pandas доступны через атрибуты `pd`. Обращаться к атрибутам можно посредством синтаксиса с использованием точки:

```
pd.атрибут
```

Блокноты Jupyter обладают удобной для поиска атрибутов возможностью автодополнения. Введите название библиотеки, добавьте точку и нажмите кнопку **Tab**, чтобы увидеть модальное окно с экспортами данного пакета. По мере ввода дальнейших символов блокнот будет фильтровать результаты в соответствии с критерием поиска.

На рис. 2.1 показана возможность автодополнения в действии. После ввода заглавной буквы **S** и нажатия **Tab** мы видим все экспорты `pd`, начинающиеся с этого символа. Обратите внимание, что в этом поиске учитывается регистр. Если

возможность автодополнения не работает, добавьте следующий код в ячейку блокнота, выполните его и попробуйте произвести поиск еще раз:

```
%config Completer.use_jedi = False
```

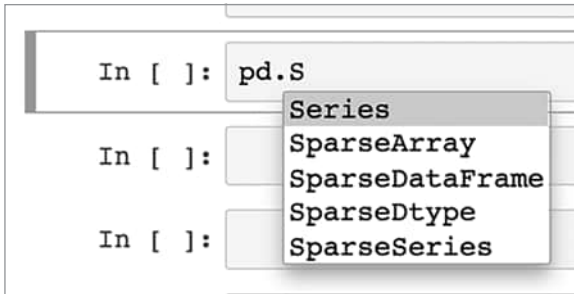


Рис. 2.1. Демонстрация возможностей автодополнения в блокнотах Jupyter на примере отображения экспортов pandas, начинающихся с буквы S

Для навигации по результатам поиска в модальном окне можно использовать стрелки вверх/вниз на клавиатуре. К счастью, класс `Series` — первый в списке результатов поиска. Нажмите клавишу `Enter` для автоматического дополнения его названия.

2.1.1. Классы и экземпляры

Класс (class) — это схема объекта Python. Класс `pd.Series` представляет собой шаблон, так что далее нам нужно создать конкретный его экземпляр. Мы создаем экземпляр (объект) класса с помощью открывающей и закрывающей скобок. Создадим объект `Series` на основе класса `Series`:

```
In [2] pd.Series()
```

```
Out [2] Series([], dtype: float64)
```

Вместе с результатами может появиться предупреждение, акцентированное красным прямоугольником:

```
DeprecationWarning: The default dtype for empty Series will be 'object'
instead of 'float64' in a future version. Specify a dtype explicitly to
silence this warning.1
```

¹ В будущих версиях dtype по умолчанию для пустых объектов Series будет 'object', а не 'float64'. Задайте dtype явным образом для подавления этого предупреждения. — *Примеч. пер.*

Это предупреждение обращает наше внимание: поскольку мы не указали никаких сохраняемых значений, библиотека pandas не делает никаких выводов о типе хранимых этим **Series** значений. Не волнуйтесь: указанное предупреждение было ожидаемым.

Мы успешно создали наш первый объект **Series**! Да, пока он не содержит никаких данных. Наполним его значениями.

2.1.2. Наполнение объекта **Series** значениями

Для создания объекта из класса служит специальный тип метода — *конструктор* (constructor). Мы уже пользовались конструктором **Series** для создания нового объекта **Series** в подразделе 2.1.1, когда формировали `pd.Series()`.

При создании объекта часто бывает нужно задать его начальное состояние. Можно считать начальное состояние объекта его исходной конфигурацией — его «настройками». Часто требуется задать состояние путем передачи аргументов конструктору, с помощью которого создается объект. *Аргумент* — это передаваемое методу входное значение.

Давайте потренируемся в создании объектов **Series** из введенных вручную данных. Наша цель — освоиться с видом и особенностями этой структуры. В будущем мы будем наполнять наши **Series** значениями на основе импортированных наборов данных.

Первый аргумент конструктора **Series** — итерируемый объект, значениями из которого и будет наполняться объект **Series**. Можно передавать различные входные значения, включая списки, ассоциативные массивы, кортежи и NumPy-объекты `ndarray`.

Создадим объект **Series** на основе данных из списка Python. В следующем примере список из четырех строк объявляется, присваивается переменной `ice_cream_flavors`, а затем передается в конструктор **Series**:

```
In [3] ice_cream_flavors = [
        "Chocolate",
        "Vanilla",
        "Strawberry",
        "Rum Raisin",
    ]
    pd.Series(ice_cream_flavors)
```

```
Out [3] 0    Chocolate
        1    Vanilla
        2    Strawberry
        3    Rum Raisin
        dtype: object
```


Чудесно! Мы создали новый объект `Series`, содержащий четыре значения из списка `ice_cream_flavors`. Обратите внимание, что pandas сохраняет порядок строковых значений из входного списка. Чуть позже мы вернемся к числам слева от выведенного списка значений `Series`.

Параметр (parameter) — название, присваиваемое входному значению функции или метода. «За кулисами» выполнения команд Python сопоставляет каждый передаваемый аргумент параметру. Можно просмотреть параметры конструктора непосредственно в блокноте Jupyter. Введите `pd.Series()` в новой ячейке, поместите курсор мыши между круглыми скобками и нажмите Shift+Tab. На рис. 2.2 приведено отображаемое при этом модальное окно.

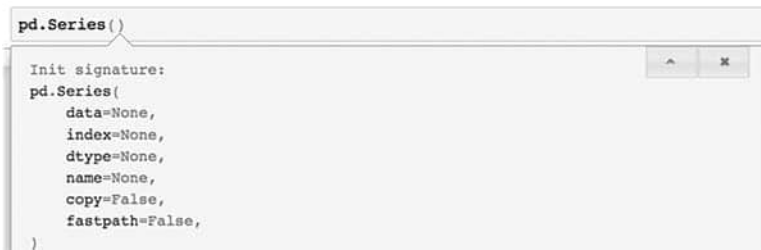


Рис. 2.2. Модальное окно документации с параметрами и аргументами по умолчанию для конструктора класса `Series`

Нажмите Shift+Tab несколько раз, чтобы увидеть дополнительную информацию. В конце концов Jupyter прикрепит панель документации к нижней части экрана.

В конструкторе `Series` описано шесть параметров: `data`, `index`, `dtype`, `name`, `copy` и `fastpath`. С помощью этих параметров можно задать начальное состояние объекта. Можно считать параметры своего рода настройками конфигурации объекта `Series`.

В документации приведены и описаны все параметры вместе с соответствующими аргументами по умолчанию. *Аргумент по умолчанию* (default argument) — резервное значение, используемое Python, если аргумент для параметра не указан пользователем явно. Например, если не передать значение для параметра `name`, Python воспользуется значением `None`. Задание значений параметрам с аргументами по умолчанию по своей сути является необязательным. Какой-то аргумент всегда будет, либо явный, из вызова метода, либо неявный, из его описания. Выше мы смогли создать объект класса `Series` без аргументов, поскольку все шесть параметров его конструктора — необязательные.

Первый параметр конструктора `Series`, `data`, должен содержать объект, значениями которого будет заполняться объект `Series`. Если передать конструктору аргументы без названий параметров, Python будет предполагать, что они

передаются последовательно. В приведенном выше примере кода мы передали список `ice_cream_flavors` в качестве первого аргумента конструктора, следовательно, Python сопоставит его с `data` — первым параметром конструктора. Кроме того, Python прибегнет к аргументам по умолчанию: `None` использует для параметров `index`, `dtype` и `name` и `False` для `copy` и `fastpath`.

Можно связывать параметры и аргументы явным образом с помощью ключевых аргументов (см. приложение Б). Введите название параметра, а за ним — знак равенства и аргумент. В следующем примере в первой строке используются аргументы, передаваемые позиционно, а во второй — ключевые аргументы, но конечные результаты выполнения команд одинаковы:

```
In [4] # Две строки ниже эквивалентны
      pd.Series(ice_cream_flavors)
      pd.Series(data = ice_cream_flavors)
```

```
Out [4] 0      Chocolate
        1      Vanilla
        2    Strawberry
        3    Rum Raisin
        dtype: object
```

Преимущество ключевых аргументов в том, что они позволяют явно указать, что означает и какому параметру предназначается каждый из аргументов конструктора. Вторая строка в примере более информативна, из нее понятнее, что `ice_cream_flavors` соответствует параметру `data` конструктора `Series`.

2.1.3. Пользовательские индексы для Series

Давайте взглянем внимательнее на наш объект `Series`:

```
0      Chocolate
1      Vanilla
2    Strawberry
3    Rum Raisin
dtype: object
```

Ранее я упоминал, что в библиотеке pandas каждому значению `Series` соответствует порядковый номер. Набор целых чисел по возрастанию слева от выведенных значений называется индексом. Каждое из этих чисел означает порядковый номер значения в объекте `Series`. Отсчет индекса начинается с 0. Строковому значению "Chocolate" соответствует индекс 0, "Vanilla" — 1 и т. д. В визуальных приложениях электронных таблиц нумерация строк данных начинается с 1 — важное различие между pandas и Excel.

Термин *индекс* (index) служит для обозначения и описания как набора идентификаторов, так и отдельного идентификатора. Приведу два выражения, они оба

вполне осмысленны: «Индекс объекта **Series** состоит из целочисленных значений» и «Значение 'Strawberry' располагается по индексу 2 в объекте **Series**»¹.

Индекс последней позиции на единицу меньше, чем общее количество значений. Текущий объект **Series** содержит четыре вкуса мороженого, так что индекс доходит до значения 3.

Помимо позиций индекса, можно присваивать значения в **Series** посредством меток индекса. Допустим любой неизменяемый тип меток индекса: строковые значения, кортежи, метки даты/времени и многое другое. Подобная гибкость очень существенно расширяет возможности **Series**, позволяя ссылаться на значение по порядковому номеру или по ключу/метке. В каком-то смысле у каждого значения есть два идентификатора.

Второй параметр конструктора **Series**, **index**, задает метки индекса объекта **Series**. Если не передать аргумент для этого параметра, по умолчанию **pandas** использует числовой индекс, начинающийся с 0. При таком типе индекса идентификаторы метки и позиции совпадают.

Сформируем объект **Series** с пользовательским индексом. Можно передавать в параметрах **data** и **index** объекты различных типов, но длины их должны быть одинаковыми, чтобы **pandas** смогла сопоставить их значения. В следующем примере мы передаем список строковых значений в параметр **data** и кортеж строковых значений — в параметр **index**. Длина как списка, так и кортежа равна 4:

```
In [5] ice_cream_flavors = [
        "Chocolate",
        "Vanilla",
        "Strawberry",
        "Rum Raisin",
    ]

    days_of_week = ("Monday", "Wednesday", "Friday", "Saturday")

    # Две строки ниже эквивалентны
    pd.Series(ice_cream_flavors, days_of_week)
    pd.Series(data = ice_cream_flavors, index = days_of_week)
```

```
Out [5] Monday      Chocolate
        Wednesday   Vanilla
        Friday      Strawberry
        Saturday    Rum Raisin
        dtype: object
```

Библиотека **pandas** связывает значения из списка **ice_cream_flavors** и кортежа **days_of_week** в соответствии с совпадением индексов. Например, **pandas** видит

¹ Синонимично фразе «Значение 'Strawberry' располагается на позиции с индексом 2 в объекте **Series**». — *Примеч. пер.*

"Rum Raisin" и "Saturday" на позиции с индексом 3 в соответствующих объектах и поэтому связывает их в объекте `Series` между собой.

Несмотря на то, что индекс состоит из строковых меток, pandas все равно ставит каждому значению `Series` в соответствие позицию индекса. Другими словами, можно обращаться к значению "Vanilla" либо по метке индекса "Saturday", либо по индексу 1. Мы обсудим вопросы обращения к элементам `Series` по номерам строк и меткам в главе 4.

Значения индекса могут дублироваться — нюанс, отличающий `Series` от ассоциативного массива языка Python. В следующем примере строковое значение "Wednesday" встречается среди меток индекса объекта `Series` дважды:

```
In [6] ice_cream_flavors = [
        "Chocolate",
        "Vanilla",
        "Strawberry",
        "Rum Raisin",
    ]

    days_of_week = ("Monday", "Wednesday", "Friday", "Wednesday")

    # Две строки ниже эквивалентны
    pd.Series(ice_cream_flavors, days_of_week)
    pd.Series(data = ice_cream_flavors, index = days_of_week)
```

```
Out [6] Monday      Chocolate
        Wednesday   Vanilla
        Friday      Strawberry
        Wednesday   Rum Raisin
        dtype: object
```

Хотя библиотека pandas допускает наличие дубликатов, лучше избегать их, используя для этого любую возможность, поскольку при уникальном индексе библиотека может находить метки индекса быстрее.

Еще одно преимущество ключевых аргументов — возможность передачи параметров в любом порядке, в то время как последовательные/позиционные аргументы требуют передачи в порядке, ожидаемом конструктором. В следующем примере мы поменяли местами ключевые параметры `index` и `data`, а библиотека pandas создает тот же объект `Series`:

```
In [7] pd.Series(index = days_of_week, data = ice_cream_flavors)
```

```
Out [7] Monday      Chocolate
        Wednesday   Vanilla
        Friday      Strawberry
        Wednesday   Rum Raisin
        dtype: object
```

Один элемент этих выведенных результатов мы еще не обсуждали, а именно: оператор `dtype` внизу, который отражает тип данных, присвоенный значениям в объекте `Series`. Отображаемый `dtype` для большинства типов данных будет вполне предсказуемым (например, `bool`, `float` или `int`). Для строковых значений и более сложных объектов (например, вложенных структур данных) pandas будет отображать `dtype: object`¹.

Ниже следуют примеры, в которых мы создаем объекты `Series` из списков булевых, целочисленных значений и значений с плавающей точкой. Обратите внимание на схожесть и различия объектов `Series`:

```
In [8] bunch_of_bools = [True, False, False]
      pd.Series(bunch_of_bools)
```

```
Out [8] 0    True
        1   False
        2   False
        dtype: bool
```

```
In [9] stock_prices = [985.32, 950.44]
      time_of_day = ["Open", "Close"]
      pd.Series(data = stock_prices, index = time_of_day)
```

```
Out [9] Open    985.32
        Close   950.44
        dtype: float64
```

```
In [10] lucky_numbers = [4, 8, 15, 16, 23, 42]
      pd.Series(lucky_numbers)
```

```
Out [10] 0     4
         1     8
         2    15
         3    16
         4    23
         5    42
         dtype: int64
```

Типы данных `float64` и `int64` означают, что каждое значение с плавающей точкой (целочисленное) в объекте `Series` занимает 64 бита (8 байт) оперативной памяти компьютера. Для изучения эффективной работы с библиотекой pandas углубляться в эти нюансы прямо сейчас нам не нужно.

Pandas делает все возможное для назначения подходящего типа данных в `Series`, исходя из значений параметра `data`. Но можно и принудительно

¹ См. объяснение, почему pandas отображает "object" в качестве dtype для строковых значений, по адресу <http://mng.bz/7j6v>.

выполнить преобразование в другой тип данных при помощи параметра `dtype` конструктора. Приведу такой пример: в конструктор передается список целочисленных значений, но запрашивается объект **Series** со значениями с плавающей точкой:

```
In [11] lucky_numbers = [4, 8, 15, 16, 23, 42]
        pd.Series(lucky_numbers, dtype = "float")
```

```
Out [11] 0      4.0
         1      8.0
         2     15.0
         3     16.0
         4     23.0
         5     42.0
         dtype: float64
```

В этом примере используются как позиционные, так и ключевые аргументы. Мы передаем список `lucky_numbers` последовательно в параметр `data`, а явным образом передаем параметр `dtype` при помощи ключевого аргумента. Конструктор **Series** ожидает, что параметр `dtype` будет третьим в последовательности, так что мы не можем просто передать его сразу после `lucky_numbers`, а должны воспользоваться ключевым аргументом.

2.1.4. Создание объекта **Series** с пропущенными значениями

Итак, пока все хорошо. Наш объект **Series** — простой и завершенный. При создании наборов данных очень удобно, когда исходные данные идеальны. На практике же данные обычно намного более запутанны. Вероятно, самая распространенная проблема, встречающаяся специалистам по анализу данных, — пропущенные значения.

При обнаружении пропущенного значения во время импорта файла библиотека `pandas` заменяет его NumPy объектом `nan`. Акроним `nan` означает «*нечисловое значение*» (not a number) и представляет собой собирательный термин для неопределенных значений. Другими словами, `nan` — условный объект, обозначающий пустое или отсутствующее значение.

Давайте незаметно протащим в наш объект **Series** отсутствующее значение. При импорте библиотеки NumPy ранее мы задали для нее псевдоним `np`. Атрибут `nan` доступен на верхнем уровне этой библиотеки, в виде экспорта. В следующем примере `np.nan` уютно угнездится в списке температур, передаваемом конструктору **Series**. Обратите внимание, что `NaN` располагается в результатах работы

кода на позиции с индексом 2. Привыкайте к этой буквенной триаде — NaN, она будет очень часто встречаться нам в книге:

```
In [12] temperatures = [94, 88, np.nan, 91]
        pd.Series(data = temperatures)

Out [12] 0    94.0
         1    88.0
         2     NaN
         3    91.0
        dtype: float64
```

Обратите внимание, что `dtype` у `Series` — `float64`. При обнаружении значения `nan` pandas автоматически преобразует числовые значения из целых в числа с плавающей точкой; такое внутреннее техническое требование позволяет библиотеке хранить числовые значения и отсутствующие значения в одном и том же однородном `Series`.

2.2. СОЗДАНИЕ ОБЪЕКТОВ SERIES НА ОСНОВЕ ОБЪЕКТОВ ЯЗЫКА PYTHON

В параметр `data` конструктора `Series` можно передавать различные входные данные, включая нативные структуры данных Python и объекты из других библиотек. В этом разделе мы обсудим, как конструктор `Series` обрабатывает ассоциативные массивы, кортежи, множества и массивы NumPy. Объект `Series`, возвращаемый pandas, функционирует одинаково, вне зависимости от источника данных.

Ассоциативный массив (dictionary) — это набор пар «ключ/значение» (см. приложение Б). При получении ассоциативного массива конструктор превращает каждый ключ в соответствующую метку индекса в `Series`:

```
In [13] calorie_info = {
        "Cereal": 125,
        "Chocolate Bar": 406,
        "Ice Cream Sundae": 342,
    }

    diet = pd.Series(calorie_info)
    diet

Out [13] Cereal          125
         Chocolate Bar    406
         Ice Cream Sundae  342
        dtype: int64
```

Кортеж (tuple) — это неизменяемый список. После создания кортежа добавлять, удалять или заменять его элементы нельзя (см. приложение Б). При получении кортежа конструктор заполняет объект **Series** вполне ожидаемым образом:

```
In [14] pd.Series(data = ("Red", "Green", "Blue"))
```

```
Out [14] 0      Red
         1    Green
         2     Blue
         dtype: object
```

Для создания содержащего кортежи объекта **Series** необходимо заключить эти кортежи в список. Кортежи хорошо подходят для строк значений, состоящих из множества частей/компонентов, например адресов:

```
In [15] rgb_colors = [(120, 41, 26), (196, 165, 45)]
         pd.Series(data = rgb_colors)
```

```
Out [15] 0      (120, 41, 26)
         1      (196, 165, 45)
         dtype: object
```

Множество (set) — это неупорядоченный набор уникальных значений. Множество объявляется с помощью фигурных скобок, точно так же, как ассоциативный массив. Python различает эти две структуры данных по наличию пар «ключ/значение» (см. приложение Б).

Если попытаться передать множество в конструктор **Series**, pandas сгенерирует исключение **TypeError**, поскольку для множеств не определены ни порядок (как у списка), ни связь (как у словаря). А значит, библиотека pandas не может знать, в каком порядке хранить значения множества¹:

```
In [16] my_set = {"Ricky", "Bobby"}
         pd.Series(my_set)
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-16-bf85415a7772> in <module>
      1 my_set = { "Ricky", "Bobby" }
----> 2 pd.Series(my_set)
```

```
TypeError: 'set' type is unordered
```

Если в вашей программе используется множество, лучше преобразуйте его в упорядоченную структуру данных, прежде чем передавать конструктору **Series**.

¹ См. Constructing a Series with a set returns a set and not a Series, <https://github.com/pandas-dev/pandas/issues/1913>.

В следующем примере множество `my_set` преобразуется в список с помощью встроенной функции `list` языка Python:

```
In [17] pd.Series(list(my_set))
```

```
Out [17] 0    Ricky
         1    Bobby
         dtype: object
```

Поскольку множество не упорядочено, гарантировать порядок элементов списка (а значит, и элементов объекта `Series`) нельзя.

В параметр `data` конструктора `Series` можно также передать объект `ndarray` библиотеки NumPy. Массивы NumPy применяются во множестве библиотек для исследования данных, это распространенный формат хранения данных для их последующего перемещения. В примере ниже в конструктор `Series` передается объект `ndarray`, сгенерированный функцией `randint` библиотеки NumPy (см. приложение В):

```
In [18] random_data = np.random.randint(1, 101, 10)
         random_data
```

```
Out [18] array([27, 16, 13, 83,  3, 38, 34, 19, 27, 66])
```

```
In [19] pd.Series(random_data)
```

```
Out [19] 0    27
         1    16
         2    13
         3    83
         4     3
         5    38
         6    34
         7    19
         8    27
         9    66
         dtype: int64
```

Как и в случае любых других входных данных, библиотека `pandas` сохраняет порядок значений объекта `ndarray` в новом объекте `Series`.

2.3. АТРИБУТЫ SERIES

Атрибут — это относящийся к объекту элемент данных. Атрибуты раскрывают информацию о внутреннем состоянии объекта. Значение атрибута может представлять собой другой объект. См. подробный обзор в приложении Б.

Объект `Series` состоит из нескольких меньших объектов. Можете считать эти объекты своего рода фрагментами пазла, которые в совокупности составляют единое целое. Возьмем для примера объект `Series` `calorie_info` из раздела 2.2:

```
Cereal          125
Chocolate Bar   406
Ice Cream Sundae 342
dtype: int64
```

Для хранения количества калорий в этом объекте `Series` используется объект `ndarray` библиотеки NumPy, а для хранения названий пищевых продуктов в индексе — объект `Index` библиотеки pandas. Получить доступ к этим вложенным объектам можно через атрибуты объекта `Series`. Например, атрибут `values` позволяет обращаться к объекту `ndarray`, в котором хранятся значения:

```
In [20] diet.values

Out [20] array([125, 406, 342])
```

Если тип объекта или соответствующая библиотека точно не известны, можно передать объект встроенной функции `type` языка Python. А она вернет класс, экземпляром которого является данный объект:

```
In [21] type(diet.values)

Out [21] numpy.ndarray
```

Сделаем небольшую паузу и подумаем. Pandas делегирует ответственность за хранение значений `Series` объекту из другой библиотеки. Именно поэтому NumPy — одна из зависимостей pandas. Объект `ndarray` оптимизирован по скорости и эффективности обработки за счет использования для значительной доли вычислений низкоуровневого языка программирования C. Во многих своих качествах `Series` представляет собой своеобразную обертку — дополнительный слой функциональности, окружающий базовый объект библиотеки pandas.

Конечно, в pandas есть и свои типы объектов. Например, атрибут `index` возвращает объект `Index` с метками `Series`:

```
In [22] diet.index

Out [22] Index(['Cereal', 'Chocolate Bar', 'Ice Cream Sundae'],
              dtype='object')
```

Объекты для индексов, например `Index`, встроены в pandas:

```
In [23] type(diet.index)

Out [23] pandas.core.indexes.base.Index
```

Некоторые атрибуты позволяют увидеть полезные подробности устройства объектов. `dtype`, например, возвращает тип данных значений объекта `Series`:

```
In [24] diet.dtype
```

```
Out [24] dtype('int64')
```

Атрибут `size` возвращает количество значений в объекте `Series`:

```
In [25] diet.size
```

```
Out [25] 3
```

Дополняющий его атрибут `shape` возвращает кортеж с размерностями структуры данных библиотеки `pandas`. Для одномерного объекта `Series` единственное значение этого кортежа совпадает с размером этого `Series`. Запятая после 3 — стандартное наглядное представление кортежей Python из одного элемента:

```
In [26] diet.shape
```

```
Out [26] (3,)
```

Атрибут `is_unique` возвращает `True`, если все значения `Series` уникальны:

```
In [27] diet.is_unique
```

```
Out [27] True
```

Атрибут `is_unique` возвращает `False`, если объект `Series` содержит повторяющиеся значения:

```
In [28] pd.Series(data = [3, 3]).is_unique
```

```
Out [28] False
```

Атрибут `is_monotonic` возвращает `True`, если каждое значение объекта `Series` больше предыдущего или равно ему. Приращение значений при этом не обязано быть одинаковым:

```
In [29] pd.Series(data = [1, 3, 6]).is_monotonic
```

```
Out [29] True
```

Атрибут `is_monotonic` возвращает `False`, если хотя бы один элемент объекта `Series` меньше предыдущего:

```
In [30] pd.Series(data = [1, 6, 3]).is_monotonic
```

```
Out [30] False
```

Подытожим: атрибуты возвращают информацию о внутреннем состоянии объекта. Атрибуты открывают доступ к вложенным объектам, возможно обладающим собственной функциональностью. В Python объектами является все, включая целочисленные, строковые и булевы значения. Таким образом, атрибут, возвращающий число, формально не отличается от возвращающего сложный объект, наподобие `ndarray`.

2.4. ИЗВЛЕЧЕНИЕ ПЕРВОЙ И ПОСЛЕДНЕЙ СТРОК

Вы уже, наверное, освоились с созданием объектов `Series`. Ничего страшного, если технической терминологии оказалось для вас слишком много; мы уже охватили большой пласт информации и еще не раз будем возвращаться к ней в этой книге. А сейчас начнем обсуждать возможные операции над объектами `Series`.

Объект языка Python включает как атрибуты, так и методы. *Атрибут* — это какие-либо относящиеся к объекту данные — открытое для доступа свойство структуры данных или подробности ее внутреннего устройства. В разделе 2.3 мы обращались к таким атрибутам объектов `Series`, как `size`, `shape`, `values` и `index`.

Метод же — это относящаяся к объекту функция — какое-либо действие или команда, которые должен выполнить объект. Методы обычно связаны с каким-либо анализом, вычислениями или операциями над атрибутами объекта. Атрибуты определяют *состояние* (state) объекта, а методы — его *поведение* (behavior).

Создадим самый большой из проработанных нами к настоящему моменту объектов `Series`. Воспользуемся встроенной функцией `range` языка Python для генерации последовательности всех целых чисел между начальным и конечным значением. Три аргумента функции `range`: нижняя граница диапазона, верхняя граница диапазона и шаг последовательности (интервал между соседними числами).

Фрагмент кода ниже генерирует диапазон чисел из 100 значений, от 0 до 500 с шагом 5, после чего передает полученный объект в конструктор `Series`:

```
In [31] values = range(0, 500, 5)
        nums = pd.Series(data = values)
        nums
Out [31] 0          0
         1          5
         2         10
         3         15
         4         20
         ...
```

```

95    475
96    480
97    485
98    490
99    495
Length: 100, dtype: int64

```

У нас теперь есть объект **Series** со ста значениями. Вот это да! Обратите внимание на пропуск (многоточие) в середине выведенных результатов. Этим pandas сообщает, что сократила их, скрыв часть строк. Библиотека, что очень удобно, обрезает объект **Series**, отображая только первые и последние пять строк. Вывод слишком большого количества строк замедлил бы работу блокнота Jupyter.

Вызвать метод можно путем указания после его названия пары скобок. Вызовем несколько простых методов **Series**. Начнем с метода **head**, возвращающего строки из начала набора данных. Он принимает один аргумент **n** — число извлекаемых строк:

```

In [32] nums.head(3)

Out [32] 0      0
         1      5
         2     10
         dtype: int64

```

При вызовах методов можно указывать ключевые аргументы, как и при вызове конструкторов и функций. Результаты работы следующего кода совпадают с результатами работы предыдущего:

```

In [33] nums.head(n = 3)

Out [33] 0      0
         1      5
         2     10
         dtype: int64

```

Подобно функциям, можно объявлять аргументы по умолчанию для параметров методов. Аргумент по умолчанию параметра **n** метода **head** — 5. Если не передать явным образом аргумент для параметра **n**, pandas вернет пять строк (так задано разработчиками библиотеки pandas):

```

In [34] nums.head()

Out [34] 0      0
         1      5
         2     10
         3     15
         4     20
         dtype: int64

```

Дополняющий его метод `tail` возвращает строки из конца объекта `Series`:

```
In [35] nums.tail(6)
```

```
Out [35] 94      470
          95      475
          96      480
          97      485
          98      490
          99      495
          dtype: int64
```

Аргумент по умолчанию параметра `n` метода `tail` — тоже 5:

```
In [36] nums.tail()
```

```
Out [36] 95      475
          96      480
          97      485
          98      490
          99      495
          dtype: int64
```

`head` и `tail` — два метода, которые я чаще всего использую в своей работе: они удобны для быстрого предварительного просмотра начала и конца набора данных. Далее мы рассмотрим несколько более продвинутых методов `Series`.

2.5. МАТЕМАТИЧЕСКИЕ ОПЕРАЦИИ

Объекты `Series` включают множество статистических и математических методов. Давайте оценим некоторые из этих методов в действии. Можете спокойно просмотреть этот раздел по диагонали и вернуться к нему, когда понадобится найти конкретную функцию.

2.5.1. Статистические операции

Начнем с создания объекта `Series` на основе списка чисел по возрастанию с затесавшимся посередине значением `np.nan`. Напомню, что при наличии в источнике данных хотя бы одного отсутствующего значения библиотека `pandas` преобразует все целочисленные значения в значения с плавающей точкой:

```
In [37] numbers = pd.Series([1, 2, 3, np.nan, 4, 5])
        numbers
```

```
Out [37] 0      1.0
          1      2.0
          2      3.0
```

```

3    NaN
4    4.0
5    5.0
dtype: float64

```

Метод `count` служит для подсчета числа непустых значений:

```
In [38] numbers.count()
```

```
Out [38] 5
```

Метод `sum` суммирует значения объекта `Series`:

```
In [39] numbers.sum()
```

```
Out [39] 15.0
```

Большинство математических методов по умолчанию игнорируют отсутствующие значения. Чтобы включить их в расчеты, можно передать аргумент `False` для параметра `skipna`.

Вот пример вызова метода `sum` именно с этим параметром. Pandas возвращает `nan`, поскольку не может прибавить к сумме нечисловое значение `nan`, расположенное по индексу 3:

```
In [40] numbers.sum(skipna = False)
```

```
Out [40] nan
```

Параметр `min_count` метода `sum` задает минимальное количество допустимых значений в `Series`, которое необходимо pandas для подсчета суммы объекта. Наш объект `Series numbers` содержит пять обычных значений и одно значение `nan`.

В примере ниже объект `Series` удовлетворяет заданному порогу в три значения, так что библиотека pandas возвращает сумму:

```
In [41] numbers.sum(min_count = 3)
```

```
Out [41] 15.0
```

И напротив, в следующем вызове для вычисления библиотекой pandas суммы необходимо наличие шести допустимых значений (у нас пять), так что библиотека pandas возвращает `nan`:

```
In [42] numbers.sum(min_count = 6)
```

```
Out [42] nan
```

СОВЕТ

Если вы захотите узнать, какие параметры есть у метода, можете нажать `Shift+Tab` между скобками метода, чтобы вывести в блокноте Jupyter документацию.

Метод `product` перемножает все значения объекта `Series`:

```
In [43] numbers.product()
```

```
Out [43] 120.0
```

Этот метод также принимает параметры `skipna` и `min_count`. Вот здесь мы просим pandas учесть при вычислениях значения `nan`:

```
In [44] numbers.product(skipna = False)
```

```
Out [44] nan
```

А в следующем примере мы запрашиваем вычисление произведения всех значений объекта `Series`, если их не меньше трех:

```
In [45] numbers.product(min_count = 3)
```

```
Out [45] 120.0
```

Метод `cumsum` (нарастающая сумма) возвращает новый объект `Series`, содержащий скользящую сумму значений. Соответствующая каждому индексу позиция содержит сумму значений, вплоть до значения по этому индексу (включительно). Нарастающая сумма помогает определять, какие значения вносят максимальный вклад в общую сумму:

```
In [46] numbers
```

```
Out [46] 0      1.0
         1      2.0
         2      3.0
         3      NaN
         4      4.0
         5      5.0
         dtype: float64
```

```
In [47] numbers.cumsum()
```

```
Out [47] 0      1.0
         1      3.0
         2      6.0
         3      NaN
         4     10.0
         5     15.0
         dtype: float64
```

Пройдемся по результатам расчета.

- Нарастающая сумма на позиции с индексом 0 равна 1.0 — первому значению из объекта `Series numbers`. Складывать с ним пока нечего.

- Нарастающая сумма на позиции с индексом 1 равна 3.0 — сумме 1.0 по индексу 0 и 2.0 с позиции с индексом 1.
- Нарастающая сумма по индексу 2 равна 6.0 — сумме 1.0, 2.0 и 3.0.
- Объект `numbers` на позиции с индексом 3 содержит `nan`. Библиотека `pandas` не может прибавить отсутствующее значение к нарастающей сумме, так что помещает `nan` на соответствующую позицию в возвращаемом объекте `Series`.
- Нарастающая сумма на позиции с индексом 4 равна 10.0. Библиотека `pandas` складывает предыдущую нарастающую сумму со значением, соответствующим текущему индексу ($1.0 + 2.0 + 3.0 + 4.0$).

Если передать в параметре `skipna` аргумент `False`, нарастающие суммы будут отображаться вплоть до индекса, по которому встречается первое отсутствующее значение, а для остальных значений будет выводиться `NaN`:

```
In [48] numbers.cumsum(skipna = False)
```

```
Out [48] 0    1.0
         1    3.0
         2    6.0
         3    NaN
         4    NaN
         5    NaN
         dtype: float64
```

Метод `pct_change` (процентное изменение) возвращает процентную разницу между последовательными значениями объекта `Series`. На позиции, соответствующей каждому из индексов, `pandas` складывает значение с предыдущей позиции и текущее, после чего делит сумму на предыдущее значение. `Pandas` вычисляет процентную разницу только при наличии на обеих позициях допустимых числовых значений.

По умолчанию метод `pct_change` реализует стратегию *заполнения предыдущим значением* (`forward-fill`) для отсутствующих значений. При такой стратегии `pandas` заменяет `nan` последним из встреченных допустимых значений. Вызовем этот метод и пройдем пошагово по выполняемым им вычислениям:

```
In [49] numbers
```

```
Out [49] 0    1.0
         1    2.0
         2    3.0
         3    NaN
         4    4.0
         5    5.0
         dtype: float64
```

```
In [50] numbers.pct_change()
```

```
Out [50] 0      NaN
         1    1.000000
         2    0.500000
         3    0.000000
         4    0.333333
         5    0.250000
         dtype: float64
```

Вот что сделала pandas.

- На позиции с индексом 0 еще нет никакого предыдущего значения, с которым pandas могла бы сравнить значение 1.0. Поэтому возвращаемый объект **Series** на позиции с индексом 0 содержит значение NaN.
- На позиции с индексом 1 pandas сравнивает значение по индексу 1, равное 2.0, со значением по индексу 0, равным 1.0. Разница между 2.0 и 1.0 в процентах составляет 100 (в два раза), что превращается в 1.000000 на позиции с индексом 1 в возвращаемом объекте **Series**.
- На позиции с индексом 2 pandas повторяет аналогичную операцию.
- На позиции с индексом 3 объект **Series numbers** содержит отсутствующее значение NaN. Pandas заменяет его на последнее встреченное допустимое значение (3.0 с индекса 2). Разница между 3.0 по индексу 3 и 3.0 по индексу 2 составляет 0 %.
- На позиции с индексом 4 pandas сравнивает значение по индексу 4, равное 4.0, со значением в предыдущей строке. И снова заменяет nan на последнее встреченное допустимое значение, 3.0. Разница между 4 и 3 составляет 0.333333 (рост на 33 %).

На рис. 2.3 демонстрируются вычисления процентного изменения с заполнением предыдущим значением. Отправная точка — объект **Series** слева. Объект **Series** посередине демонстрирует выполняемые библиотекой pandas промежуточные вычисления. Объект **Series** справа — итоговый результат.

0	1.0	0	NaN	0	NaN
1	2.0	1	$(2.0 - 1.0) / 1.0$	1	1.000000
2	3.0	2	$(3.0 - 2.0) / 2.0$	2	0.500000
3	NaN	3	$(4.0 - 3.0) / 3.0$	3	0.000000
4	4.0	4	$(5.0 - 4.0) / 4.0$	4	0.333333
5	5.0	5	$(3.0 - 3.0) / 3.0$	5	0.250000

Рис. 2.3. Пошаговый разбор вычисления значений методом `pct_change` с заполнением предыдущим значением

Параметр `fill_method` позволяет задавать способ замены значений `NaN` методом `pct_change`. Этот параметр присутствует во множестве методов, так что стоит потратить немного времени и познакомиться с ним поближе. Как уже упоминалось ранее, при стратегии заполнения предыдущим значением, принятой по умолчанию, библиотека `pandas` заменяет значение `nan` *последним* допустимым просмотренным значением. Тот же результат можно получить, передав параметр `fill_method` с указанным явным образом аргументом `"pad"` или `"ffill"`:

```
In [51] # Три строки ниже эквивалентны
        numbers.pct_change()
        numbers.pct_change(fill_method = "pad")
        numbers.pct_change(fill_method = "ffill")
```

```
Out [51] 0      NaN
         1      1.000000
         2      0.500000
         3      0.000000
         4      0.333333
         5      0.250000
         dtype: float64
```

Альтернативная стратегия обработки отсутствующих значений — *заполнение следующим значением* (`backfill`). При этом варианте `pandas` заменяет значение `nan` на следующее допустимое значение. Передадим параметр `fill_method` со значением `"bfill"` и рассмотрим пошагово, что получится:

```
In [52] # Две строки ниже эквивалентны
        numbers.pct_change(fill_method = "bfill")
        numbers.pct_change(fill_method = "backfill")
```

```
Out [52] 0      NaN
         1      1.000000
         2      0.500000
         3      0.333333
         4      0.000000
         5      0.250000
         dtype: float64
```

Обратите внимание, что значения на позициях с индексами 3 и 4 в вариантах заполнения следующим и предыдущим значением отличаются. Вот как `pandas` получила последний результат.

- На позиции с индексом 0 еще нет никакого предыдущего значения, с которым `pandas` могла бы сравнить значение `1.0`. Поэтому возвращаемый объект `Series` на позиции с индексом 0 содержит значение `NaN`.
- На позиции с индексом 3 `pandas` встречается с `NaN` в объекте `Series numbers`. `Pandas` заменяет его на следующее допустимое значение (`4.0` с индексом 4). Искомая разница между `4.0` по индексу 3 и `3.0` по индексу 2 составляет `0.33333`.

- На позиции с индексом 4 pandas сравнивает 4.0 со значением по индексу 3. И снова заменяет NaN по индексу 3 на 4.0 — следующее допустимое значение, присутствующее в объекте `Series numbers`. Разница между 4 и 4 равна 0.0.

На рис. 2.4 показаны вычисления процентных изменений с заполнением следующим значением. Отправная точка — объект `Series` слева. Объект `Series` посередине демонстрирует выполняемые библиотекой pandas промежуточные вычисления. Объект `Series` справа — итоговый результат.

0	1.0	0	NaN	0	NaN
1	2.0	1	$(2.0 - 1.0) / 1.0$	1	1.000000
2	3.0	2	$(3.0 - 2.0) / 2.0$	2	0.500000
3	NaN	3	$(4.0 - 3.0) / 3.0$	3	0.333333
4	4.0	4	$(4.0 - 4.0) / 4.0$	4	0.000000
5	5.0	5	$(5.0 - 4.0) / 4.0$	5	0.250000

Рис. 2.4. Пошаговый разбор вычисления значений методом `pct_change` с заполнением следующим значением

Метод `mean` возвращает среднее значений объекта `Series`. Среднее равно результату от деления суммы значений на их количество:

```
In [53] numbers.mean()
```

```
Out [53] 3.0
```

Метод `median` возвращает число, расположенное посередине в отсортированном списке значений `Series`. Половина значений объекта `Series` будет меньше медианного, а половина — больше:

```
In [54] numbers.median()
```

```
Out [54] 3.0
```

Метод `std` возвращает *стандартное отклонение* (standard deviation) — меру отклонения данных от среднего значения:

```
In [55] numbers.std()
```

```
Out [55] 1.5811388300841898
```

Методы `max` и `min` служат для извлечения максимального и минимального значений из объекта `Series`:

```
In [56] numbers.max()
```

```
Out [56] 5.0
```

```
In [57] numbers.min()
```

```
Out [57] 1.0
```

Библиотека `pandas` сортирует строковые объекты `Series` в алфавитном порядке. «Минимальной» строкой считается ближайшая к началу алфавита, а «максимальной» — ближайшая к его концу. Вот простой пример с небольшим объектом `Series`:

```
In [58] animals = pd.Series(["koala", "aardvark", "zebra"])
        animals
```

```
Out [58] 0      koala
         1  aardvark
         2    zebra
        dtype: object
```

```
In [59] animals.max()
```

```
Out [59] 'zebra'
```

```
In [60] animals.min()
```

```
Out [60] 'aardvark'
```

Получить всю сводную информацию об объекте `Series` сразу можно с помощью одного замечательного метода — `describe`. Он возвращает объект `Series` со статистическими показателями, включая количество, среднее значение (математическое ожидание) и стандартное отклонение:

```
In [61] numbers.describe()
```

```
Out [61] count      5.000000
        mean       3.000000
        std        1.581139
        min        1.000000
        25%        2.000000
        50%        3.000000
        75%        4.000000
        max        5.000000
        dtype: float64
```

Метод `sample` выбирает случайный набор значений из объекта `Series`. При этом порядок значений нового объекта может отличаться от порядка в исходном

объекте `Series`. В примере ниже обратите внимание, что благодаря отсутствию значений `NaN` в случайной выборке библиотека `pandas` возвращает `Series` целочисленных значений. Если бы хотя бы одно из выбранных значений было `NaN`, `pandas` вернула бы `Series` значений с плавающей точкой:

```
In [62] numbers.sample(3)
```

```
Out [62] 1      2
          3      4
          2      3
          dtype: int64
```

Метод `unique` возвращает NumPy-объект `ndarray`, содержащий неповторяющиеся значения из объекта `Series`. В следующем примере строковое значение "Orwell" встречается дважды в объекте `Series` `authors`, но лишь один раз — в возвращаемом объекте `ndarray`:

```
In [63] authors = pd.Series(
          ["Hemingway", "Orwell", "Dostoevsky", "Fitzgerald", "Orwell"]
        )
        authors.unique()
```

```
Out [63] array(['Hemingway', 'Orwell', 'Dostoevsky', 'Fitzgerald'],
               dtype=object)
```

Дополняющий его метод `nunique` возвращает количество уникальных значений в объекте `Series`:

```
In [64] authors.nunique()
```

```
Out [64] 4
```

Возвращаемое методом `nunique` значение равняется длине массива из метода `unique`.

2.5.2. Арифметические операции

В подразделе 2.5.1 мы пробовали вызывать различные математические методы наших объектов `Series`. Библиотека `pandas` предоставляет и другие способы выполнения арифметических вычислений с `Series`. Начнем с создания объекта `Series` с целочисленными значениями и одним отсутствующим значением:

```
In [65] s1 = pd.Series(data = [5, np.nan, 15], index = ["A", "B", "C"])
        s1
```

```
Out [65] A      5.0
          B      NaN
          C     15.0
          dtype: float64
```

Производить арифметические операции над `Series` в Python можно с помощью стандартных математических операторов:

- `+` для сложения;
- `-` для вычитания;
- `*` для умножения;
- `/` для деления.

Синтаксис понятен интуитивно: с объектом `Series` можно обращаться как с обычным операндом на одной из сторон математического оператора. Обратите внимание, что результатом любой математической операции с `nan` является `nan`. В следующем примере мы прибавляем 3 ко всем значениям из объекта `Series` `s1`:

```
In [66] s1 + 3
```

```
Out [66] A      8.0
         B      NaN
         C     18.0
         dtype: float64
```

Некоторые разработчики могут удивиться такому результату. Как можно прибавить целочисленное значение к структуре данных? Ведь типы явно несовместимы. Дело в том, что «за кулисами» операции библиотека `pandas` оказывается достаточно интеллектуальной, чтобы разобрать наш синтаксис и понять, что мы хотим прибавить целочисленное значение к каждому из значений в объекте `Series`, а не к самому объекту `Series`.

Если вам больше нравится подход на основе методов, тот же результат можно получить с помощью метода `add`:

```
In [67] s1.add(3)
```

```
Out [67] A      8.0
         B      NaN
         C     18.0
         dtype: float64
```

Приведу три примера, иллюстрирующих различные варианты синтаксиса для вычитания (`-`), умножения (`*`) и деления (`/`). Зачастую в `pandas` можно произвести одну и ту же операцию несколькими эквивалентными способами:

```
In [68] # Три строки ниже эквивалентны
         s1 - 5
         s1.sub(5)
         s1.subtract(5)
```

80 Часть I. Основы pandas

```
Out [68] A    0.0  
        B    NaN  
        C   10.0  
        dtype: float64
```

```
In [69] # Три строки ниже эквивалентны  
        s1 * 2  
        s1.mul(2)  
        s1.multiply(2)
```

```
Out [69] A   10.0  
        B    NaN  
        C   30.0  
        dtype: float64
```

```
In [70] # Три строки ниже эквивалентны  
        s1 / 2  
        s1.div(2)  
        s1.divide(2)
```

```
Out [70] A    2.5  
        B    NaN  
        C    7.5  
        dtype: float64
```

Оператор целочисленного деления с округлением вниз (`//`) производит деление, после чего удаляет из результата все цифры после десятичной точки. Например, при обычном делении 15 на 4 получается 3,75, а при целочисленном делении с округлением вниз — 3. Этот оператор также можно применять к объектам `Series`. Либо можно вызвать метод `floordiv`:

```
In [71] # Две строки ниже эквивалентны  
        s1 // 4  
        s1.floordiv(4)
```

```
Out [71] A    1.0  
        B    NaN  
        C    3.0  
        dtype: float64
```

Оператор деления по модулю (`%`) возвращает остаток от деления. Вот пример:

```
In [72] # Две строки ниже эквивалентны  
        s1 % 3  
        s1.mod(3)
```

```
Out [72] A    2.0  
        B    NaN  
        C    0.0  
        dtype: float64
```


В последнем примере:

- pandas делит значение `5.0`, соответствующее метке индекса A, на 3 и возвращает остаток `2.0`;
- pandas не может разделить `NaN`, соответствующий метке индекса B;
- pandas делит значение `15.0`, соответствующее метке индекса C, на 3 и возвращает остаток `0.0`.

2.5.3. Транслирование

Напомню, что «за кулисами» выполнения команд библиотека pandas хранит значения объектов `Series` в NumPy-объектах `ndarray`. При использовании синтаксиса наподобие `s1 + 3` или `s1 - 5` pandas поручает математические вычисления NumPy.

В документации NumPy вычисление массива значений на основе другого массива описывается при помощи термина «*транслирование*» (broadcasting). Если не углубляться слишком сильно в технические подробности (для эффективной работы с библиотекой pandas не обязательно понимать все нюансы функционирования NumPy), термин «*транслирование*» ведет свое начало от вышек радиотрансляции, передающих один и тот же сигнал на все слушающие его приемники. Синтаксис `s1 + 3` означает «Применить одну и ту же операцию (прибавление 3) к каждому из значений объекта `Series`». Все значения объекта `Series` получают одно сообщение, подобно тому как все слушатели одной радиостанции в одно время слышат одну и ту же песню.

Транслирование также описывает математические операции между несколькими объектами `Series`. Как показывает опыт, библиотека pandas выравнивает значения в различных структурах данных в соответствии с совпадением меток индекса. Продемонстрируем эту идею на примере. Создадим два объекта `Series` с одинаковым индексом из трех элементов:

```
In [73] s1 = pd.Series([1, 2, 3], index = ["A", "B", "C"])
        s2 = pd.Series([4, 5, 6], index = ["A", "B", "C"])
```

При использовании оператора `+` с этими двумя объектами `Series` в качестве операндов библиотека pandas складывает значения, расположенные на позициях с одинаковым индексом:

- на позиции с индексом A pandas складывает значения 1 и 4, в результате чего получает 5;
- на позиции с индексом B pandas складывает значения 2 и 5, в результате чего получает 7;

- на позиции с индексом C pandas складывает значения 3 и 6, в результате чего получает 9.

```
In [74] s1 + s2
```

```
Out [74] A    5
         B    7
         C    9
         dtype: int64
```

На рис. 2.5 приведена наглядная иллюстрация выравнивания двух объектов Series библиотекой pandas.

A	1		A	4		A	5
B	2	+	B	5	=	B	7
C	3		C	6		C	9

Рис. 2.5. Библиотека pandas выравнивает объекты Series при выполнении математических операций в соответствии с совпадением меток индекса

А вот еще один пример того, как библиотека pandas выравнивает данные по общим меткам индекса. Создадим еще два объекта Series с обычным числовым индексом. И добавим в каждый объект отсутствующее значение:

```
In [75] s1 = pd.Series(data = [3, 6, np.nan, 12])
         s2 = pd.Series(data = [2, 6, np.nan, 12])
```

Оператор равенства (==) языка Python проверяет два объекта на равенство. С его помощью можно сравнить значения в двух объектах Series, как показано в следующем примере. Обратите внимание, что библиотека pandas считает, что одно значение nan не равно другому значению nan: нельзя же предполагать, что одно отсутствующее значение всегда равно другому отсутствующему значению. Эквивалентный оператору равенства метод называется eq:

```
In [76] # Две строки ниже эквивалентны
         s1 == s2
         s1.eq(s2)
```

```
Out [76] 0    False
         1     True
         2    False
         3     True
         dtype: bool
```

Оператор неравенства (!=) помогает убедиться, что два значения не равны. Эквивалентный оператору неравенства метод называется ne:

```
In [77] # Две строки ниже эквивалентны
         s1 != s2
         s1.ne(s2)
```

```
Out [77] 0      True
         1     False
         2      True
         3     False
         dtype: bool
```

Операции сравнения объектов **Series** усложняются, если индексы неодинаковы. В одном из индексов может быть больше или меньше меток, либо сами метки могут не совпадать.

Вот пример создания двух объектов **Series**, у которых совпадают только две метки индекса, B и C:

```
In [78] s1 = pd.Series(
         data = [5, 10, 15], index = ["A", "B", "C"]
       )

         s2 = pd.Series(
         data = [4, 8, 12, 14], index = ["B", "C", "D", "E"]
       )
```

Что будет, если сложить **s1** и **s2**? Pandas складывает значения на позициях с метками B и C и возвращает значения **NaN** для оставшихся индексов (A, D и E). Напоминаю, что любая арифметическая операция с участием значения **NaN** возвращает **NaN**:

```
In [79] s1 + s2

Out [79] A      NaN
         B     14.0
         C     23.0
         D      NaN
         E      NaN
         dtype: float64
```

На рис. 2.6 показано, как библиотека **pandas** выравнивает объекты **Series s1** и **s2**, после чего складывает соответствующие значения.

A	5	+	B	4	=	A	NaN
B	10		C	8		B	14.0
C	15		D	12		C	23.0
			E	14		D	NaN
						E	NaN

Рис. 2.6. Библиотека **pandas** возвращает **NaN** во всех случаях, когда метки индекса объектов **Series** не совпадают

Резюмируем: библиотека **pandas** выравнивает данные в двух объектах **Series** по совпадению меток индекса, заменяя **NaN** везде, где требуется.

2.6. ПЕРЕДАЧА ОБЪЕКТОВ SERIES ВСТРОЕННЫМ ФУНКЦИЯМ ЯЗЫКА PYTHON

Сообщество разработчиков Python стремится вырабатывать единую точку зрения относительно определенных архитектурных принципов ради единообразия различных баз кода. Один из примеров — беспроblemная интеграция между объектами библиотек и встроенными функциями языка Python. Библиотека pandas не исключение. Можно передавать объекты `Series` любым встроенным функциям языка Python и получать вполне предсказуемые результаты. Создадим небольшой объект `Series`, содержащий названия городов США:

```
In [80] cities = pd.Series(
        data = ["San Francisco", "Los Angeles", "Las Vegas", np.nan]
    )
```

Функция `len` возвращает количество строк в объекте `Series`, включая отсутствующие значения (`NaN`):

```
In [81] len(cities)
```

```
Out [81] 4
```

Как мы видели ранее, функция `type` возвращает класс, к которому относится объект. Ее можно использовать, когда есть сомнения насчет того, с какой структурой данных мы имеем дело или к какой библиотеке она относится:

```
In [82] type(cities)
```

```
Out [82] pandas.core.series.Series
```

Функция `dir` возвращает список атрибутов и методов объекта в виде строковых значений. Обратите внимание, что в следующем примере приведена сокращенная версия выводимых данных:

```
In [83] dir(cities)
```

```
Out [83] ['T',
        '_AXIS_ALIASES',
        '_AXIS_IALIASES',
        '_AXIS_LEN',
        '_AXIS_NAMES',
        '_AXIS_NUMBERS',
        '_AXIS_ORDERS',
        '_AXIS_REVERSED',
        '_HANDLED_TYPES',
        '__abs__',
        '__add__',
        '__and__',
        '__annotations__',
        '__array__']
```

```
'__array_priority__',
#...
]
```

Значениями объекта **Series** можно заполнить нативную структуру данных Python. В примере ниже мы создаем из нашего объекта **cities** типа **Series** список с помощью функции **list** языка Python:

```
In [84] list(cities)
```

```
Out [84] ['San Francisco', 'Los Angeles', 'Las Vegas', nan]
```

Можно также передать объект **Series** встроенной функции **dict** языка Python, чтобы получить ассоциативный массив. Pandas отображает метки индекса и значения объекта **Series** в ключи и значения ассоциативного массива:

```
In [85] dict(cities)
```

```
Out [85] {0: 'San Francisco', 1: 'Los Angeles', 2: 'Las Vegas', 3: nan}
```

В Python для проверки вхождения используется ключевое слово **in**. В библиотеке **pandas** с помощью ключевого слова **in** можно проверять, существует ли определенное значение в индексе объекта **Series**. Напомню, как выглядит объект **cities**:

```
In [86] cities
```

```
Out [86] 0    San Francisco
         1    Los Angeles
         2    Las Vegas
         3             NaN
         dtype: object
```

В следующих двух примерах мы проверяем, входят ли "Las Vegas" и 2 в индекс нашего объекта **Series**:

```
In [87] "Las Vegas" in cities
```

```
Out [87] False
```

```
In [88] 2 in cities
```

```
Out [88] True
```

Чтобы проверить, входит ли что-то в число значений объекта **Series**, можно воспользоваться, помимо ключевого слова **in**, атрибутом **values**. Напомню, что через атрибут **values** можно обращаться к объекту **ndarray**, в котором содержатся сами данные:

```
In [89] "Las Vegas" in cities.values
```

```
Out [89] True
```

Для проверки на невхождение можно воспользоваться обратным оператором `not in`. Этот оператор возвращает `True`, если pandas не удастся найти соответствующее значение в объекте `Series`:

```
In [90] 100 not in cities
```

```
Out [90] True
```

```
In [91] "Paris" not in cities.values
```

```
Out [91] True
```

Объекты pandas часто интегрируются со встроенными функциями языка Python и предоставляют свои собственные атрибуты/методы для доступа к тем же данным с теми же целями. Можете использовать тот синтаксис, который вам удобнее.

2.7. УПРАЖНЕНИЯ

Пришло время предложить первый комплект упражнений этой книги! Их задача — помочь вам освоиться с понятиями, с которыми вы познакомились в этой главе. Решения приведены сразу же после упражнений. Удачи!

2.7.1. Задачи

Пусть даны две структуры данных:

```
In [92] superheroes = [
        "Batman",
        "Superman",
        "Spider-Man",
        "Iron Man",
        "Captain America",
        "Wonder Woman"
    ]
```

```
In [93] strength_levels = (100, 120, 90, 95, 110, 120)
```

А вот задачи этой главы.

1. Заполните новый объект `Series` значениями из списка супергероев.
2. Заполните новый объект `Series` значениями из кортежа уровней силы.
3. Создайте объект `Series` с супергероями в качестве меток индекса и уровнями силы в качестве значений. Присвойте этот объект `Series` переменной `heroes`.
4. Извлеките первые две строки из объекта `Series` `heroes`.

5. Извлеките последние четыре строки из объекта Series `heroes`.
6. Определите количество уникальных значений в `heroes`.
7. Вычислите среднюю силу супергероев в `heroes`.
8. Вычислите максимальную и минимальную силу в `heroes`.
9. Вычислите, каким будет уровень силы каждого из супергероев при удвоении.
10. Преобразуйте объект Series `heroes` в ассоциативный массив языка Python.

2.7.2. Решения

Взглянем на решения задач из подраздела 2.7.1.

1. Для создания нового объекта Series можно воспользоваться конструктором Series, доступным на верхнем уровне библиотеки pandas. В качестве первого его позиционного аргумента необходимо передать источник данных:

```
In [94] pd.Series(superheroes)
```

```
Out [94] 0          Batman
         1          Superman
         2      Spider-Man
         3          Iron Man
         4  Captain America
         5    Wonder Woman
         dtype: object
```

2. Решение этой задачи идентично предыдущей: необходимо просто передать наш кортеж с уровнями силы конструктору Series. На этот раз укажем ключевой параметр `data` явным образом:

```
In [95] pd.Series(data = strength_levels)
```

```
Out [95] 0    100
         1    120
         2     90
         3     95
         4    110
         5    120
         dtype: int64
```

3. Для создания объекта Series с пользовательским индексом необходимо передать конструктору параметр `index`. В данном случае роль значений Series будут играть уровни силы, а меток индекса — имена супергероев:

```
In [96] heroes = pd.Series(
         data = strength_levels, index = superheroes
       )
         heroes
```

```
Out [96] Batman          100
         Superman        120
         Spider-Man       90
         Iron Man         95
         Captain America 110
         Wonder Woman     120
         dtype: int64
```

4. Напомню, что *метод* — это действие над объектом или команда объекту. Можно воспользоваться методом `head` для извлечения строк из начала структуры данных библиотеки `pandas`. Единственный параметр этого метода, `n`, задает число извлекаемых строк. Метод `head` возвращает новый объект `Series`:

```
In [97] heroes.head(2)
```

```
Out [97] Batman          100
         Superman        120
         dtype: int64
```

5. Парный к нему метод `tail` извлекает строки из конца структуры данных библиотеки `pandas`. Для извлечения четырех последних строк необходимо передать ему аргумент `4`:

```
In [98] heroes.tail(4)
```

```
Out [98] Spider-Man       90
         Iron Man         95
         Captain America 110
         Wonder Woman     120
         dtype: int64
```

6. Чтобы определить число уникальных значений в объекте `Series`, можно воспользоваться методом `nunique`. Объект `Series` `heroes` содержит всего шесть значений, из них пять уникальных; значение `120` встречается дважды:

```
In [99] heroes.nunique()
```

```
Out [99] 5
```

7. Для вычисления среднего значения объекта `Series` можно вызвать метод `mean`:

```
In [100] heroes.mean()
```

```
Out [100] 105.83333333333333
```

8. Следующая задача — определить максимальное и минимальное значения в объекте `Series`. Для этой цели подойдут функции `max` и `min`:

```
In [101] heroes.max()
```

```
Out [101] 120
```



```
In [102] heroes.min()
```

```
Out [102] 90
```

9. Как удвоить уровень силы всех супергероев? Умножить все значения объекта `Series` на 2. В следующем решении используется оператор умножения, но можно также воспользоваться методом `mul` или `multiply`:

```
In [103] heroes * 2
```

```
Out [103] Batman          200
          Superman        240
          Spider-Man      180
          Iron Man        190
          Captain America 220
          Wonder Woman    240
          dtype: int64
```

10. Последняя задача: преобразовать объект `Series` `heroes` в ассоциативный массив Python. Решить эту задачу можно посредством передачи нашей структуры данных конструктору/функции `dict` языка Python. Библиотека `pandas` делает из меток индекса ключи ассоциативного массива, а из значений `Series` — значения ассоциативного массива:

```
In [104] dict(heroes)
```

```
Out [104] {'Batman': 100,
          'Superman': 120,
          'Spider-Man': 90,
          'Iron Man': 95,
          'Captain America': 110,
          'Wonder Woman': 120}
```

Поздравляю, вы выполнили свой первый набор упражнений!

РЕЗЮМЕ

- `Series` представляет собой одномерный однородный маркированный массив, содержащий значения и индекс.
- Значения объекта `Series` могут относиться к любому типу данных. Метки индекса могут относиться к любому неизменяемому типу данных.
- `Pandas` задает *позицию* индекса и *метку* индекса для каждого значения объекта `Series`.
- Заполнить объект `Series` можно данными из списка, ассоциативного массива, кортежа, массива `NumPy` и многих других источников данных.

- Метод `head` извлекает первые строки объекта `Series`.
- Метод `tail` извлекает последние строки объекта `Series`.
- Класс `Series` поддерживает распространенные статистические операции, такие как вычисление суммы, среднего значения, медианы и стандартного отклонения.
- Библиотека `pandas` позволяет применить арифметические операции над несколькими объектами `Series` в соответствии с совпадением меток индекса.
- Класс `Series` прекрасно работает со встроенными функциями языка Python, включая `dict`, `list` и `len`.

3

Методы класса Series

В этой главе

- ✓ Импорт наборов данных в формате CSV с помощью функции `read_csv`.
- ✓ Сортировка значений объектов `Series` в порядке возрастания и убывания.
- ✓ Извлечение самого большого и маленького значений в объекте `Series`.
- ✓ Подсчет количества уникальных значений в объекте `Series`.
- ✓ Вызов функции для всех значений в объекте `Series`.

В главе 2 вы начали изучать объект `Series` — одномерный маркированный массив однородных данных. Мы заполняли объекты `Series` данными из различных источников, включая списки, ассоциативные массивы и `ndarray` библиотеки NumPy. Вы видели, как библиотека `pandas` задает для каждого значения объекта `Series` метку и позицию индекса. Вы научились применять математические операции к объектам `Series`.

Теперь, после изучения основ, пора приступить к реальным наборам данных! В этой главе вы познакомитесь со множеством продвинутых операций класса `Series`, включая сортировку, подсчет значений и группировку по корзинам. Вы также увидите, как эти методы помогают извлекать полезную информацию из данных. Приступим!

3.1. ИМПОРТ НАБОРА ДАННЫХ С ПОМОЩЬЮ ФУНКЦИИ `READ_CSV`

Файлы CSV представляют собой открытый текст, в котором строки данных разделяются символом переноса строки, а значения внутри строк — запятыми. Первая строка файла содержит названия столбцов данных. В этой главе мы поэкспериментируем с тремя CSV-файлами.

- `pokemon.csv` — список более чем 800 покемонов, мультипликационных монстров из популярной медиафраншизы Nintendo. Каждому покемону соответствует один или несколько *типов* (type), например Fire (Огонь), Water (Вода) и Grass (Трава).
- `google_stock.csv` — набор ежедневных курсов акций в долларах США для технологической компании Google начиная с ее выхода на рынок в августе 2004 года до октября 2019-го.
- `revolutionary_war.csv` — список битв периода Войны за независимость США. Для каждого сражения указаны дата начала и штат США.

Начнем с импорта наборов данных. По мере обсуждения мы рассмотрим некоторые возможные виды оптимизации, с помощью которой можно упростить анализ.

Первый шаг — открытие нового блокнота Jupyter и импорт библиотеки pandas. Создайте блокнот в том же каталоге, что и CSV-файлы:

```
In [1] import pandas as pd
```

Библиотека pandas включает более десятка функций импорта для загрузки файлов в различных форматах. Эти функции доступны на верхнем уровне библиотеки, их названия начинаются с приставки `read`. В нашем случае для импорта CSV-файла необходима функция `read_csv`. В первый параметр этой функции, `filepath_or_buffer`, необходимо передать строковое значение с именем файла. Убедитесь, что строковое значение включает расширение `.csv` (например, `"pokemon.csv"`, а не просто `"pokemon"`). По умолчанию библиотека pandas ищет файл в том же каталоге, где находится блокнот:

```
In [2] # Две нижеприведенных строки эквивалентны
pd.read_csv(filepath_or_buffer = "pokemon.csv")
pd.read_csv("pokemon.csv")
```

```
Out [2]
```

	Pokemon	Type
0	Bulbasaur	Grass / Poison
1	Ivysaur	Grass / Poison
2	Venusaur	Grass / Poison

```

3      Charmander      Fire
4      Charmeleon     Fire
...
804    Stakataka      Rock / Steel
805    Blacephalon    Fire / Ghost
806    Zeraora        Electric
807    Meltan         Steel
808    Melmetal       Steel

```

```
809 rows × 2 columns
```

Вне зависимости от числа столбцов в наборе данных функция `read_csv` всегда импортирует данные в объект **DataFrame** — двумерную структуру данных **pandas**, поддерживающую хранение нескольких строк и столбцов. Вы познакомитесь с этим объектом ближе в главе 4. Ничего плохого в использовании **DataFrame** нет, но мы хотели бы поработать с объектом **Series**, так что сохраним данные из CSV-файла в этой меньшей по размеру структуре данных.

Первая проблема: набор данных включает два столбца (**Pokemon** и **Type**), но **Series** поддерживает только один столбец данных. Простейшее решение — сделать из одного из столбцов данных индекс **Series**. Для задания столбца индекса можно воспользоваться параметром `index_col`. Не забудьте о чувствительности к регистру: строковое значение должно совпадать с заголовком в наборе данных. Передадим строковое значение "Pokemon" в качестве аргумента параметра `index_col`:

```
In [3] pd.read_csv("pokemon.csv", index_col = "Pokemon")
```

```
Out [3]
```

Pokemon	Type
Bulbasaur	Grass / Poison
Ivysaur	Grass / Poison
Venusaur	Grass / Poison
Charmander	Fire
Charmeleon	Fire
...	...
Stakataka	Rock / Steel
Blacephalon	Fire / Ghost
Zeraora	Electric
Meltan	Steel
Melmetal	Steel

```
809 rows × 1 columns
```

Мы успешно сделали из столбца **Pokemon** индекс **Series**, но по умолчанию библиотека **pandas** все равно импортирует данные в **DataFrame**. В конце концов, контейнер, способный хранить несколько столбцов данных, вполне может хранить один столбец, оставаясь при этом двумерным **DataFrame**. Чтобы заставить

библиотеку pandas использовать `Series`, необходимо добавить еще один параметр — `squeeze`, указав для него аргумент `True`. Параметр `squeeze` приводит к использованию `Series` вместо `DataFrame`:

```
In [4] pd.read_csv("pokemon.csv", index_col = "Pokemon", squeeze = True)
```

```
Out [4] Pokemon
      Bulbasaur      Grass / Poison
      Ivysaur      Grass / Poison
      Venusaur      Grass / Poison
      Charmander      Fire
      Charmeleon      Fire
      ...
      Stakataka      Rock / Steel
      Blacephalon      Fire / Ghost
      Zeraora      Electric
      Meltan      Steel
      Melmetal      Steel
      Name: Type, Length: 809, dtype: object
```

Теперь мы получили объект `Series`. Ура! Метки индекса — названия покемонов, а значения — типы покемонов.

Под значениями в выводе приводится весьма важная информация:

- библиотека pandas присвоила объекту `Series` название `Type` — имя столбца из CSV-файла;
- объект `Series` содержит 809 значений;
- `dtype: object` сообщает нам, что `Series` содержит строковые значения. `object` — внутреннее обозначение библиотеки pandas для строковых значений и более сложных структур данных.

Последний этап — присваивание этого объекта `Series` переменной, для которой прекрасно подойдет название `pokemon`:

```
In [5] pokemon = pd.read_csv(
      "pokemon.csv", index_col = "Pokemon", squeeze = True
    )
```

Обработка двух оставшихся наборов данных требует рассмотрения некоторых дополнительных нюансов. Давайте взглянем на `google_stock.csv`:

```
In [6] pd.read_csv("google_stocks.csv").head()
```

```
Out [6]
```

	Date	Close
0	2004-08-19	49.98
1	2004-08-20	53.95

```

2  2004-08-23  54.50
3  2004-08-24  52.24
4  2004-08-25  52.80

```

При импорте набора данных библиотека `pandas` определяет самый подходящий тип данных для каждого столбца. Иногда библиотека `pandas` перестраховывается и старательно избегает предположений по умолчанию относительно наших данных. Файл `google_stock.csv`, например, включает столбец `Date` со значениями даты/времени в формате `YYYY-MM-DD` (например, `2010-08-04`). И если не дать библиотеке `pandas` конкретное указание обрабатывать эти значения как метки даты/времени, по умолчанию `pandas` импортирует их в виде строковых значений. Строковое значение — более универсальный и гибкий тип данных; в виде строки может быть выражено практически любое значение.

Давайте явным образом попросим библиотеку `pandas` преобразовать значения из столбца `Date` в метки даты/времени. И хотя мы обсудим метки даты/времени только в главе 11, здесь и в будущем рекомендуется хранить данные каждого из столбцов в наиболее подходящем типе данных. Если `pandas` знает, что речь идет о метках даты/времени, то дает возможность использовать дополнительные методы, недоступные для простых строковых значений, например, можно вызвать метод для вычисления дня недели даты.

Параметр `parse_dates` функции `read_csv` позволяет задать список строковых значений, отмечающих столбцы, текстовые значения из которых `pandas` должна преобразовать в метки даты/времени. В следующем примере в него передается список, содержащий значения `"Date"`:

```
In [7] pd.read_csv("google_stocks.csv", parse_dates = ["Date"]).head()
```

```
Out [7]
```

```

      Date  Close
-----
0  2004-08-19  49.98
1  2004-08-20  53.95
2  2004-08-23  54.50
3  2004-08-24  52.24
4  2004-08-25  52.80

```

Визуально никаких отличий в выводимых результатах нет, но «за кулисами», не показывая явно, библиотека `pandas` хранит столбец `Date` в формате другого типа данных. Сделаем столбец `Date` индексом объекта `Series` с помощью параметра `index_col`, `Series` прекрасно работает с метками даты/времени в роли индексов. Наконец, добавим параметр `squeeze`, чтобы получить объект `Series` вместо `DataFrame`:

```

In [8] pd.read_csv(
      "google_stocks.csv",
      parse_dates = ["Date"],

```

```

        index_col = "Date",
        squeeze = True
    ).head()

```

```

Out [8] Date
2004-08-19    49.98
2004-08-20    53.95
2004-08-23    54.50
2004-08-24    52.24
2004-08-25    52.80
Name: Close, dtype: float64

```

То что надо. Мы получили объект **Series** с метками индекса типа даты/времени и значениями с плавающей точкой. Сохраним этот объект **Series** в переменной **google**:

```

In [9] google = pd.read_csv(
    "google_stocks.csv",
    parse_dates = ["Date"],
    index_col = "Date",
    squeeze = True
)

```

Осталось импортировать еще один набор данных: битвы периода Войны за независимость США. На этот раз при осуществлении импорта просмотрим последние пять строк набора. Для этого мы присоединим цепочкой метод **tail** к объекту **DataFrame**, возвращаемому функцией **read_csv**:

```

In [10] pd.read_csv("revolutionary_war.csv").tail()

```

```

Out [10]

```

	Battle	Start Date	State
227	Siege of Fort Henry	9/11/1782	Virginia
228	Grand Assault on Gibraltar	9/13/1782	NaN
229	Action of 18 October 1782	10/18/1782	NaN
230	Action of 6 December 1782	12/6/1782	NaN
231	Action of 22 January 1783	1/22/1783	Virginia

Взгляните на столбец **State**. Упс... в нем пропущены некоторые значения. Напомню, что библиотека **pandas** отмечает отсутствующие значения при помощи **NaN** («не числовое значение»). **NaN** — объект **NumPy**, отражающий пустоту или отсутствие значения. Этот набор данных содержит пропущенные/отсутствующие значения для битв без четкой даты начала или проходивших за пределами территории США.

Сделаем столбец **Start Date** индексом. Снова воспользуемся для этой цели параметром **index_col** для указания столбца-индекса и параметром **parse_dates** для преобразования строковых значений столбца **Start Date** в значения даты/времени. **Pandas** распознает формат дат этого набора данных (M/D/YYYY):


```
In [11] pd.read_csv(
        "revolutionary_war.csv",
        index_col = "Start Date",
        parse_dates = ["Start Date"],
    ).tail()
```

Out [11]

Start Date	Battle	State
1782-09-11	Siege of Fort Henry	Virginia
1782-09-13	Grand Assault on Gibraltar	NaN
1782-10-18	Action of 18 October 1782	NaN
1782-12-06	Action of 6 December 1782	NaN
1783-01-22	Action of 22 January 1783	Virginia

По умолчанию функция `read_csv` импортирует все столбцы из CSV-файла. Нам придется ограничить импорт двумя столбцами, чтобы получить объект **Series**: один столбец для индекса и другой для значений. Параметр `squeeze` как таковой в этом сценарии необязателен, pandas проигнорирует его, если столбцов данных более одного.

С помощью параметра `usecols` функции `read_csv` можно указать список импортируемых библиотекой pandas столбцов. Включим в него только **Start Date** и **State**:

```
In [12] pd.read_csv(
        "revolutionary_war.csv",
        index_col = "Start Date",
        parse_dates = ["Start Date"],
        usecols = ["State", "Start Date"],
        squeeze = True
    ).tail()
```

```
Out [12] Start Date
1782-09-11    Virginia
1782-09-13         NaN
1782-10-18         NaN
1782-12-06         NaN
1783-01-22    Virginia
Name: State, dtype: object
```

Идеально! Мы получили объект **Series**, состоящий из индекса со значениями даты/времени и строковых значений. Присвоим его переменной **battles**:

```
In [13] battles = pd.read_csv(
        "revolutionary_war.csv",
        index_col = "Start Date",
        parse_dates = ["Start Date"],
        usecols = ["State", "Start Date"],
        squeeze = True
    )
```

Теперь, когда мы импортировали наборы данных в объекты **Series**, прикинем, что с ними можно сделать.

3.2. СОРТИРОВКА ОБЪЕКТОВ SERIES

Можно отсортировать объект¹ `Series` по значениям или индексу, в порядке возрастания или убывания.

3.2.1. Сортировка значений с помощью метода `sort_values`

Пусть нам нужно узнать минимальный и максимальный курс акций Google. Метод `sort_values` возвращает новый объект `Series` с отсортированными в порядке возрастания значениями. «В порядке возрастания» (ascending) означает увеличение значений — другими словами, от меньших значений к большим. Метки индекса при сортировке переносятся одновременно с соответствующими значениями:

```
In [14] google.sort_values()
```

```
Out [14] Date
2004-09-03      49.82
2004-09-01      49.94
2004-08-19      49.98
2004-09-02      50.57
2004-09-07      50.60
...
2019-04-23     1264.55
2019-10-25     1265.13
2018-07-26     1268.33
2019-04-26     1272.18
2019-04-29     1287.58
Name: Close, Length: 3824, dtype: float64
```

Строковые объекты `Series` библиотека pandas сортирует в алфавитном порядке. «В порядке возрастания» при этом означает «от начала алфавита к его концу»:

```
In [15] pokemon.sort_values()
```

```
Out [15] Pokemon
Illumise      Bug
Silcoon       Bug
Pinsir        Bug
Burmey        Bug
Wurmple       Bug
...
Tirtouga      Water / Rock
Relicanth     Water / Rock
Corsola       Water / Rock
```

¹ В отличие от сортировок в таблицах, где меняется порядок следования строк, сортировку объектов `Series` следует трактовать как сортировку структур «значение + индекс» внутри объекта. — *Примеч. пер.*

```

Carracosta      Water / Rock
Empoleon         Water / Steel
Name: Type, Length: 809, dtype: object

```

Pandas при сортировке ставит символы в верхнем регистре перед символами в нижнем. Таким образом, строка с заглавной буквы Z будет располагаться в очереди раньше строки со строчной буквы a. Обдумывая код ниже, обратите внимание, что строковое значение `adam` следует за `Ben`:

```
In [16] pd.Series(data = ["Adam", "adam", "Ben"]).sort_values()
```

```

Out [16] 0      Adam
         2       Ben
         1      adam
         dtype: object

```

Задавать порядок сортировки можно с помощью параметра `ascending`, по умолчанию равного `True`. Для сортировки значений объекта `Series` в порядке убывания (от больших к меньшим) передайте в этот параметр аргумент `False`:

```
In [17] google.sort_values(ascending = False).head()
```

```

Out [17] Date
2019-04-29      1287.58
2019-04-26      1272.18
2018-07-26      1268.33
2019-10-25      1265.13
2019-04-23      1264.55
Name: Close, dtype: float64

```

Сортировка в порядке убывания располагает строковые значения объекта `Series` в обратном алфавитном порядке. «В порядке убывания» (`descending`) означает «от конца алфавита к его началу»:

```
In [18] pokemon.sort_values(ascending = False).head()
```

```

Out [18] Pokemon
Empoleon      Water / Steel
Carracosta    Water / Rock
Corsola       Water / Rock
Relicanth     Water / Rock
Tirtouga      Water / Rock
Name: Type, dtype: object

```

Параметр `na_position` определяет размещение значений `NaN` в возвращаемом объекте `Series`, по умолчанию его аргумент равен `"last"`. То есть по умолчанию библиотека `pandas` помещает отсутствующие значения в конец отсортированного объекта `Series`:

```

In [19] # Две нижеприведенных строки эквивалентны
        battles.sort_values()
        battles.sort_values(na_position = "last")

```

```
Out [19] Start Date
1781-09-06    Connecticut
1779-07-05    Connecticut
1777-04-27    Connecticut
1777-09-03        Delaware
1777-05-17        Florida
...
1782-08-08        NaN
1782-08-25        NaN
1782-09-13        NaN
1782-10-18        NaN
1782-12-06        NaN
Name: State, Length: 232, dtype: object
```

Чтобы выводить сначала отсутствующие значения, передайте в параметр `na_position` аргумент `"first"`. В результате получится объект `Series`, в котором сначала будут идти `NaN`, а затем — отсортированные значения:

```
In [20] battles.sort_values(na_position = "first")

Out [20] Start Date
1775-09-17        NaN
1775-12-31        NaN
1776-03-03        NaN
1776-03-25        NaN
1776-05-18        NaN
...
1781-07-06    Virginia
1781-07-01    Virginia
1781-06-26    Virginia
1781-04-25    Virginia
1783-01-22    Virginia
Name: State, Length: 232, dtype: object
```

А если нужно убрать значения `NaN`? Метод `dropna` возвращает объект `Series`, из которого удалены все отсутствующие значения. Обратите внимание, что этот метод удаляет `NaN` только среди значений объекта `Series`, но не индекса. Следующий пример фильтрует список сражений, оставляя только те, у которых указано место прохождения:

```
In [21] battles.dropna().sort_values()

Out [21] Start Date
1781-09-06    Connecticut
1779-07-05    Connecticut
1777-04-27    Connecticut
1777-09-03        Delaware
1777-05-17        Florida
...
1782-08-19    Virginia
1781-03-16    Virginia
1781-04-25    Virginia
```

```

1778-09-07      Virginia
1783-01-22      Virginia
Name: State, Length: 162, dtype: object

```

Полученный таким образом объект `Series` ожидаемо короче объекта `battles`. Библиотека `pandas` удалила из `battles` 70 значений `NaN`.

3.2.2. Сортировка по индексу с помощью метода `sort_index`

Иногда больший интерес вызывает индекс, а не значения. К счастью, можно отсортировать объект `Series` и по индексу, используя для этого метод `sort_index`. При этом одновременно с соответствующими метками индекса переносятся и значения. Как и `sort_values`, метод `sort_index` принимает параметр `ascending`, аргумент которого по умолчанию также равен `True`:

```

In [22] # Две нижеприведенные строки эквивалентны
        pokemon.sort_index()
        pokemon.sort_index(ascending = True)

```

```

Out [22] Pokemon
Abomasnow      Grass / Ice
Abra           Psychic
Absol          Dark
Accelgor       Bug
Aegislash      Steel / Ghost
...
Zoroark        Dark
Zorua          Dark
Zubat          Poison / Flying
Zweilous       Dark / Dragon
Zygarde        Dragon / Ground
Name: Type, Length: 809, dtype: object

```

При сортировке набора меток даты/времени в порядке возрастания библиотека `pandas` сортирует от самой ранней даты к самой поздней. Продемонстрирую подобную сортировку на примере `battles`:

```

In [23] battles.sort_index()

```

```

Out [23] Start Date
1774-09-01      Massachusetts
1774-12-14      New Hampshire
1775-04-19      Massachusetts
1775-04-19      Massachusetts
1775-04-20      Virginia
...
1783-01-22      Virginia
NaT            New Jersey
NaT            Virginia

```

```

NaT          NaN
NaT          NaN
Name: State, Length: 232, dtype: object

```

Ближе к концу отсортированного объекта `Series` замечаем новый для нас тип значения. Библиотека pandas указывает вместо отсутствующих значений дат/времени еще один NumPy-объект — `NaT` (not a time — «не значение времени»). Объект `NaT` поддерживает целостность данных с индексом типа «дата/время».

Метод `sort_index` также содержит параметр `na_position`, позволяющий влиять на расположение значений `NaN`. В следующем примере сначала выводятся отсутствующие значения, а затем отсортированные метки даты/времени:

```
In [24] battles.sort_index(na_position = "first").head()
```

```

Out [24] Start Date
NaT          New Jersey
NaT          Virginia
NaT          NaN
NaT          NaN
1774-09-01    Massachusetts
Name: State, dtype: object

```

Для сортировки в порядке убывания нужно задать параметру `ascending` аргумент `False`. При сортировке в порядке убывания даты отображаются от последней к самой ранней:

```
In [25] battles.sort_index(ascending = False).head()
```

```

Out [25] Start Date
1783-01-22    Virginia
1782-12-06     NaN
1782-10-18     NaN
1782-09-13     NaN
1782-09-11    Virginia
Name: State, dtype: object

```

Самая поздняя битва из этого набора данных произошла 22 января 1783 года в штате Виргиния.

3.2.3. Получение минимального и максимального значений с помощью методов `nsmallest` и `nlargest`

Пусть нам нужно найти пять дат, когда акции Google котировались выше всего. Один из вариантов — отсортировать объект `Series` в порядке убывания, а затем ограничить результаты первыми пятью строками:

```
In [26] google.sort_values(ascending = False).head()
```

```
Out [26] Date
        2019-04-29    1287.58
        2019-04-26    1272.18
        2018-07-26    1268.33
        2019-10-25    1265.13
        2019-04-23    1264.55
        Name: Close, dtype: float64
```

Это достаточно распространенная операция, поэтому библиотека `pandas` предоставляет вспомогательный метод для небольшой экономии кода. Метод `nlargest` возвращает максимальные значения из объекта `Series`. Первый его параметр, `n`, задает число возвращаемых записей. Аргумент по умолчанию параметра `n` равен 5. Библиотека `pandas` сортирует значения в возвращаемом объекте `Series` в порядке убывания:

```
In [27] # Две нижеприведенные строки эквивалентны
        google.nlargest(n = 5)
        google.nlargest()
```

```
Out [27] Date
        2019-04-29    1287.58
        2019-04-26    1272.18
        2018-07-26    1268.33
        2019-10-25    1265.13
        2019-04-23    1264.55
        Name: Close, dtype: float64
```

Логично дополняет этот функционал метод `nsmallest`, он возвращает минимальные значения объекта `Series`, отсортированные в порядке возрастания. Аргумент по умолчанию его параметра `n` также равен 5:

```
In [28] # Две нижеприведенные строки эквивалентны
        google.nsmallest(n = 5)
        google.nsmallest(5)
```

```
Out [28] Date
        2004-09-03    49.82
        2004-09-01    49.94
        2004-08-19    49.98
        2004-09-02    50.57
        2004-09-07    50.60
        2004-08-30    50.81
        Name: Close, dtype: float64
```

Обратите внимание, что ни один из этих методов не работает для объектов `Series` со строковыми значениями.

3.3. ПЕРЕЗАПИСЬ ОБЪЕКТА SERIES С ПОМОЩЬЮ ПАРАМЕТРА INPLACE

Все вызывавшиеся в этой главе до сих пор методы возвращали в результате своей отработки новые объекты `Series`. Исходные объекты `Series`, на которые ссылаются переменные `pokemon`, `google` и `battles`, не подвергались никаким изменениям ни в одной из предыдущих операций. В качестве примера взглянем на `battles` до и после вызова метода; объект `Series` не меняется:

```
In [29] battles.head(3)
```

```
Out [29] Start Date
        1774-09-01    Massachusetts
        1774-12-14    New Hampshire
        1775-04-19    Massachusetts
        Name: State, dtype: object
```

```
In [30] battles.sort_values().head(3)
```

```
Out [30] Start Date
        1781-09-06    Connecticut
        1779-07-05    Connecticut
        1777-04-27    Connecticut
        Name: State, dtype: object
```

```
In [31] battles.head(3)
```

```
Out [31] Start Date
        1774-09-01    Massachusetts
        1774-12-14    New Hampshire
        1775-04-19    Massachusetts
        Name: State, dtype: object
```

Но как быть, если нам нужно модифицировать объект `Series battles`? Многие методы в библиотеке `pandas` включают параметр `inplace`, задание ему аргумента `True` позволяет модифицировать объект, для которого вызывается метод.

Сравните последний пример со следующим. Мы снова вызываем метод `sort_values`, но на этот раз передаем аргумент `True` для параметра `inplace`. При использовании параметра `inplace` метод возвращает `None` и в блокноте Jupyter ничего не выводится. Но если потом вывести объект `battles`, станет очевидно, что он изменился:

```
In [32] battles.head(3)
```

```
Out [32] Start Date
        1774-09-01    Massachusetts
        1774-12-14    New Hampshire
```



```
1775-04-19    Massachusetts
Name: State, dtype: object
```

```
In [33] battles.sort_values(inplace = True)
```

```
In [34] battles.head(3)
```

```
Out [34] Start Date
1781-09-06    Connecticut
1779-07-05    Connecticut
1777-04-27    Connecticut
Name: State, dtype: object
```

У параметра `inplace` имеется одна особенность, о которой следует сказать. Его название предполагает модификацию/изменение уже существующего объекта вместо создания копии. Разработчики иногда стремятся использовать `inplace`, чтобы сократить количество копий объектов, а значит, и использование памяти. Но необходимо осознать, что даже при использовании параметра `inplace` библиотека `pandas` создает копию объекта при каждом вызове метода. Библиотека *всегда* создает дубликат; параметр `inplace` просто размещает этот новый объект в уже существующую переменную. Таким образом, вопреки расхожему мнению, параметр `inplace` не дает никакого преимущества в смысле производительности. Следующие две строки фактически эквивалентны:

```
battles.sort_values(inplace = True)
battles = battles.sort_values()
```

Почему создатели библиотеки `pandas` выбрали такую реализацию? Каковы преимущества от постоянного создания копий? Подробные пояснения вы можете найти в Интернете, но если коротко: сохранение при работе неизменяемых структур данных обычно ведет к меньшему количеству ошибок. Напомню, что неизменяемый объект — такой, который не может меняться. Можно скопировать неизменяемый объект и производить операции над копией, но исходный объект менять нельзя. Хороший пример — строковое значение языка Python. Неизменяемый объект с меньшей вероятностью окажется в поврежденном или недопустимом состоянии, кроме того, его проще тестировать.

Команда создателей библиотеки `pandas` обсуждает возможность исключения параметра `inplace` из будущих версий библиотеки. Я бы рекомендовал по возможности воздержаться от его использования. В качестве альтернативного осознанного решения можно присваивать возвращаемое значение метода той же переменной или создавать отдельную переменную с более информативным названием. Например, можно присвоить возвращаемое методом `sort_values` значение переменной с названием наподобие `sorted_battles`.

3.4. ПОДСЧЕТ КОЛИЧЕСТВА ЗНАЧЕНИЙ С ПОМОЩЬЮ МЕТОДА VALUE_COUNTS

Напомню, что представляет собой объект `Series` `pokemon`:

```
In [35] pokemon.head()
```

```
Out [35] Pokemon
Bulbasaur      Grass / Poison
Ivysaur        Grass / Poison
Venusaur       Grass / Poison
Charmander      Fire
Charmeleon      Fire
Name: Type, dtype: object
```

Как выяснить, какие типы покемонов встречаются чаще всего? Необходимо сгруппировать значения по корзинам и подсчитать количество элементов в каждой. Для решения этой задачи идеально подходит метод `value_counts`, подсчитывающий количество вхождений каждого значения объекта `Series`:

```
In [36] pokemon.value_counts()
```

```
Out [36] Normal      65
Water      61
Grass      38
Psychic    35
Fire      30
..
Fire / Dragon      1
Dark / Ghost      1
Steel / Ground     1
Fire / Psychic     1
Dragon / Ice       1
Name: Type, Length: 159, dtype: int64
```

Метод `value_counts` возвращает новый объект `Series`. В роли меток индекса выступают значения объекта `Series` `pokemon`, а в роли значений — их количество в наборе. Шестьдесят пять покемонов относятся к типу `Normal`, 61 классифицированы как `Water` и т. д. Если вам интересно, «нормальные» покемоны отличаются способностями к физическим атакам.

Длина возвращаемого `value_counts` объекта `Series` равна числу уникальных значений в объекте `Series` `pokemon`. Напомню, что подобную информацию возвращает метод `nunique`:

```
In [37] len(pokemon.value_counts())
```

```
Out [37] 159
```

```
In [38] pokemon.nunique()
```

```
Out [38] 159
```

В подобных ситуациях первостепенное значение имеет целостность данных. Лишний пробел или отличающийся регистр символа — и библиотека pandas распознает два значения как не равные между собой и посчитает их по отдельности. Мы обсудим вопросы очистки данных в главе 6.

Аргумент по умолчанию параметра `ascending` метода `value_counts` равен `False`. Библиотека pandas сортирует значения в порядке убывания, от максимального количества вхождений к минимальному. Для сортировки значений в порядке возрастания надо передать этому параметру значение `True`:

```
In [39] pokemon.value_counts(ascending = True)
```

```
Out [39] Rock / Poison      1
         Ghost / Dark      1
         Ghost / Dragon    1
         Fighting / Steel  1
         Rock / Fighting   1
         ..
         Fire              30
         Psychic           35
         Grass             38
         Water             61
         Normal            65
```

Нам, наверное, интересно будет узнать соотношение количества покемонов конкретного типа к общему их количеству. Для получения частотности всех уникальных значений установите параметр `normalize` метода `value_counts` в `True`. Частота встречаемости значения соответствует тому, какую долю набора данных оно составляет:

```
In [40] pokemon.value_counts(normalize = True).head()
```

```
Out [40] Normal      0.080346
         Water       0.075402
         Grass       0.046972
         Psychic     0.043263
         Fire        0.037083
```

Для получения доли каждого типа покемонов в процентах можно умножить объект `Series` с частотностями на 100. Помните синтаксис, применявшийся в главе 2? Работая с объектами `Series`, можно использовать обычный математический оператор умножения. Библиотека pandas при этом применит соответствующую операцию ко всем значениям:

```
In [41] pokemon.value_counts(normalize = True).head() * 100
```

```
Out [41] Normal      8.034611
         Water       7.540173
         Grass       4.697157
         Psychic     4.326329
         Fire        3.708282
```

Покемоны типа Normal составляют 8,034611 % набора данных, покемоны типа Water — 7,540173 % и т. д. Любопытная информация, не правда ли?!

Пусть нам нужно ограничить точность представления значений в процентах. Можно округлить значения объекта `Series` с помощью метода `round`. Первый параметр этого метода, `decimals`, задает количество цифр после десятичной точки. В следующем примере мы округляем значения до двух цифр; во избежание синтаксической ошибки код арифметических действий из предыдущего примера заключен в круглые скобки, чтобы гарантировать, что pandas сначала умножит все значения на 100 и лишь потом вызовет метод `round` для полученного объекта `Series`:

```
In [42] (pokemon.value_counts(normalize = True) * 100).round(2)
```

```
Out [42] Normal          8.03
         Water          7.54
         Grass         4.70
         Psychic        4.33
         Fire           3.71
         ...
         Rock / Fighting 0.12
         Fighting / Steel 0.12
         Ghost / Dragon  0.12
         Ghost / Dark    0.12
         Rock / Poison   0.12
         Name: Type, Length: 159, dtype: float64
```

Метод `value_counts` ведет себя точно так же с числовыми объектами `Series`. Приведу пример подсчета количества вхождений каждого уникального курса акций в объекте `Series google`. Оказывается, ни один курс акций не встречается в наборе данных более трех раз:

```
In [43] google.value_counts().head()
```

```
Out [43] 237.04    3
         288.92    3
         287.68    3
         290.41    3
         194.27    3
```

Для выявления тенденций в числовых наборах данных полезно будет сгруппировать значения по заранее заданным диапазонам, а не подсчитывать отдельные значения. Начнем с определения разности между минимальным и максимальным значениями в объекте `Series google`. Для этого прекрасно подойдут методы `max` и `min` класса `Series`. Или можно передать объект `Series` во встроенные функции `max` и `min` языка Python:

```
In [44] google.max()
```

```
Out [44] 1287.58
```

```
In [45] google.min()
```

```
Out [45] 49.82
```

Итак, разница между минимальным и максимальным значениями составляет приблизительно 1250. Сгруппируем курсы акций по корзинам размером 200, начиная с 0 и до 1400. Можно описать эти интервалы группировки в виде значений в списке и передать его в параметре `bins` метода `value_counts`. Каждые два последовательных значения списка будут играть роль нижней и верхней границы очередного интервала:

```
In [46] buckets = [0, 200, 400, 600, 800, 1000, 1200, 1400]
        google.value_counts(bins = buckets)
```

```
Out [46] (200.0, 400.0]      1568
         (-0.001, 200.0]    595
         (400.0, 600.0]    575
         (1000.0, 1200.0]   406
         (600.0, 800.0]    380
         (800.0, 1000.0]   207
         (1200.0, 1400.0]   93
         Name: Close, dtype: int64
```

Согласно этим результатам, курс акций Google был между 200 и 400 долларов для 1568 значений из набора данных.

Обратите внимание, что `pandas` отсортировала предыдущий объект `Series` в порядке убывания по количеству значений в каждой корзине. А что, если мы захотим отсортировать результаты по интервалам? Для этого нужно воспользоваться сочетанием нескольких методов библиотеки `pandas`. Интервалы представляют собой метки индекса в возвращаемом объекте `Series`, так что можно воспользоваться методом `sort_index` для их сортировки. Методика последовательного вызова нескольких методов называется *цепочкой вызовов методов* (method chaining):

```
In [47] google.value_counts(bins = buckets).sort_index()
```

```
Out [47] (-0.001, 200.0]    595
         (200.0, 400.0]    1568
         (400.0, 600.0]    575
         (600.0, 800.0]    380
         (800.0, 1000.0]   207
         (1000.0, 1200.0]   406
         (1200.0, 1400.0]   93
         Name: Close, dtype: int64
```

Тот же результат можно получить, передав значение `False` в параметр `sort` метода `value_counts`:

```
In [48] google.value_counts(bins = buckets, sort = False)
```

```
Out [48] (-0.001, 200.0]      595
         (200.0, 400.0]      1568
         (400.0, 600.0]      575
         (600.0, 800.0]      380
         (800.0, 1000.0]     207
         (1000.0, 1200.0]    406
         (1200.0, 1400.0]     93
         Name: Close, dtype: int64
```

Обратите внимание, что первый интервал включает значение `-0.001` вместо `0`. При размещении значений объекта `Series` по корзинам библиотека pandas может расширить любой из диапазонов до 0,1 % в любом направлении. Символы в описании интервалов означают следующее:

- круглая скобка означает, что значение *не включается* в интервал;
- квадратная скобка означает, что значение *включается* в интервал.

Рассмотрим интервал `(-0.001, 200.0]`: `-0.001` не включается в него, а `200` — включается. Таким образом, в этот интервал входят все значения, большие `-0.001` и меньшие или равные `200`.

Замкнутый интервал (closed interval) включает обе граничные точки. Например: `[5, 10]` (больше или равно 5, меньше или равно 10).

Открытый интервал (open interval) не включает ни одной из граничных точек. Например: `(5, 10)` (больше 5, меньше 10).

Метод `value_counts` с параметром `bins` возвращает *полуоткрытые* интервалы, включающие одну из граничных точек и не включающие другую.

Параметр `bins` метода `value_counts` также принимает целочисленный аргумент. Библиотека pandas автоматически вычисляет разницу между максимальным и минимальным значениями объекта `Series` и разбивает диапазон на указанное число корзин. В примере ниже данные по курсам акций разбиваются по шести корзинам. Обратите внимание, что размеры корзин не обязательно в точности одинаковы (из-за возможного расширения любого интервала на 0,1 % в любую сторону), но близки по размеру:

```
In [49] google.value_counts(bins = 6, sort = False)
```

```
Out [49] (48.581, 256.113]    1204
         (256.113, 462.407]    1104
         (462.407, 668.7]      507
         (668.7, 874.993]      380
```

```
(874.993, 1081.287]    292
(1081.287, 1287.58]    337
Name: Close, dtype: int64
```

А как насчет нашего набора данных `battles`? Мы уже давно его не видели:

```
In [50] battles.head()
```

```
Out [50] Start Date
1781-09-06    Connecticut
1779-07-05    Connecticut
1777-04-27    Connecticut
1777-09-03         Delaware
1777-05-17         Florida
Name: State, dtype: object
```

Можно с помощью метода `value_counts` выяснить, в каких штатах проходило наибольшее число битв в ходе Войны за независимость США:

```
In [51] battles.value_counts().head()
```

```
Out [51] South Carolina    31
New York                  28
New Jersey                24
Virginia                  21
Massachusetts             11
Name: State, dtype: int64
```

Библиотека `pandas` исключила по умолчанию значения `NaN` из возвращаемого `value_counts` объекта `Series`. Передадим этому методу аргумент `False` для параметра `dropna`, чтобы подсчитать отдельно пустые значения:

```
In [52] battles.value_counts(dropna = False).head()
```

```
Out [52] NaN              70
South Carolina          31
New York                28
New Jersey              24
Virginia                21
Name: State, dtype: int64
```

Индексы объектов `Series` также поддерживают метод `value_counts`. Прежде чем вызывать его, необходимо обратиться к объекту индекса через атрибут `index`. Выясним, на какие даты пришлось максимальное количество битв во время Войны за независимость США:

```
In [53] battles.index
```

```
Out [53]
```

```
DatetimeIndex(['1774-09-01', '1774-12-14', '1775-04-19', '1775-04-19',
               '1775-04-20', '1775-05-10', '1775-05-27', '1775-06-11',
```

```
'1775-06-17', '1775-08-08',
...
'1782-08-08', '1782-08-15', '1782-08-19', '1782-08-26',
'1782-08-25', '1782-09-11', '1782-09-13', '1782-10-18',
'1782-12-06', '1783-01-22'],
dtype='datetime64[ns]', name='Start Date', length=232,
freq=None)
```

```
In [54] battles.index.value_counts()
```

```
Out [54] 1775-04-19      2
         1781-05-22      2
         1781-04-15      2
         1782-01-11      2
         1780-05-25      2
         ..
         1778-05-20      1
         1776-06-28      1
         1777-09-19      1
         1778-08-29      1
         1777-05-17      1
         Name: Start Date, Length: 217, dtype: int64
```

Похоже, что ни в одну из дат не происходило более двух битв одновременно.

3.5. ВЫЗОВ ФУНКЦИИ ДЛЯ КАЖДОГО ИЗ ЗНАЧЕНИЙ ОБЪЕКТА SERIES С ПОМОЩЬЮ МЕТОДА APPLY

Функции являются *объектами первого класса* (first-class object) в языке Python, то есть язык обрабатывает их как любой другой тип данных. Может показаться, что функция — более абстрактная сущность, но это такая же допустимая структура данных, как и любая другая.

Проще всего представлять себе объекты первого класса следующим образом. Все, что можно делать с числом, можно делать и с функцией. Например, следующее:

- хранить функцию в списке;
- присваивать ключу словаря функцию в качестве значения;
- передавать одну функцию в другую в качестве аргумента;
- возвращать одну функцию из другой.

Важно различать функцию и вызов функции. *Функция* — это последовательность инструкций, генерирующих какой-либо результат, «рецепт», пока еще не ставший готовым блюдом. А *вызов функции* (function invocation) — это само выполнение инструкций, приготовление блюда по рецепту.

Приведу код, в котором объявляется список `funcs` для хранения трех встроенных функций языка Python. Функции `len`, `max` и `min` не вызываются внутри этого списка, там лишь хранятся ссылки на функции:

```
In [55] funcs = [len, max, min]
```

В следующем примере мы проходим в цикле `for` по списку `funcs`. За три итерации цикла переменная-итератор `current_func` становится по очереди функциями `len`, `max` и `min`. На каждой итерации цикла производится динамический вызов функции `current_func` с передачей ей объекта `Series google`, после чего выводится возвращенное значение:

```
In [56] for current_func in funcs:
        print(current_func(google))
```

```
Out [56] 3824
        1287.58
        49.82
```

Выводимые результаты этого фрагмента кода включают возвращаемые в ходе последовательного вызова трех функций значения: длину объекта `Series`, максимальное значение объекта `Series` и минимальное значение объекта `Series`.

Подведем итоги рассмотрения этого примера: с функцией можно обращаться в Python так же, как и с любым другим объектом. Что это означает для библиотеки `pandas`? Пусть нам нужно округлить все значения с плавающей точкой в нашем объекте `Series google` до ближайшего целого числа. В языке Python есть удобная функция для этой цели — `round`. Она округляет значения больше 0,5 вверх, а меньше 0,5 — вниз¹:

```
In [57] round(99.2)
```

```
Out [57] 99
```

```
In [58] round(99.49)
```

```
Out [58] 99
```

```
In [59] round(99.5)
```

```
Out [59] 100
```

Хорошо было бы применить эту функцию ко всем значениям в нашем объекте `Series`, правда? И... нам повезло! У `Series` есть метод `apply`, вызывающий

¹ На самом деле алгоритм ее работы несколько сложнее. В частности, если два ближайших целых числа равноудалены от округляемого значения, округление производится в сторону четного числа (так, например, `round(0.5)` и `round(-0.5)` возвращают 0, а `round(1.5)` возвращает 2). — *Примеч. пер.*

заданную функцию однократно для каждого значения объекта `Series`. Этот метод возвращает новый объект `Series`, состоящий из возвращаемых значений этих вызовов функции. В качестве первого параметра метод `apply` ожидает вызываемую функцию. Приведу фрагмент кода, в котором мы передадим методу встроенную функцию `round` языка Python:

```
In [60] # Две нижеприведенные строки эквивалентны
        google.apply(func = round)
        google.apply(round)
```

```
Out [60] Date
        2004-08-19      50
        2004-08-20      54
        2004-08-23      54
        2004-08-24      52
        2004-08-25      53
        ...
        2019-10-21     1246
        2019-10-22     1243
        2019-10-23     1259
        2019-10-24     1261
        2019-10-25     1265
        Name: Close, Length: 3824, dtype: int64
```

Свершилось! Мы округлили все значения нашего объекта `Series`.

Опять же учтите, что методу `apply` был передан не вызов функции `round`. Мы передаем «рецепт». А где-то внутри библиотеки pandas метод `apply` вызывает ее для каждого значения объекта `Series`. Библиотека pandas абстрагирует сложность этой операции, скрывая задействованные механизмы.

Методу `apply` можно передавать и пользовательские функции. Достаточно описать функцию с одним параметром, возвращающую значение, которое pandas должна поместить в агрегированный объект `Series`.

Выясним, сколько наших покемонов относится к одному типу (например, `Fire`), а сколько — к двум или более типам. К каждому значению объекта `Series` необходимо применить одну и ту же логику категоризации покемонов. Функция — идеальный контейнер для инкапсуляции подобной логики. Опишем вспомогательную функцию `single_or_multi`, принимающую на входе тип покемона и определяющую, составной он или нет. В составных типах покемонов отдельные типы разделены косой чертой ("`Fire / Ghost`"). Для проверки наличия прямой косой черты в строковом значении аргумента воспользуемся оператором `in` языка Python. Оператор `if` выполняет блок кода, только если условие равно `True`. В нашем случае при наличии в строковом значении / функция возвращает строковое значение "`Multi`", а в противном случае возвращает "`Single`":

```
In [61] def single_or_multi(pokemon_type):
        if "/" in pokemon_type:
            return "Multi"
        return "Single"
```

Теперь можно передать функцию `single_or_multi` методу `apply`. Напомню, как выглядит содержимое объекта `pokemon`:

```
In [62] pokemon.head(4)
```

```
Out [62] Pokemon
Bulbasaur      Grass / Poison
Ivysaur        Grass / Poison
Venusaur       Grass / Poison
Charmander     Fire
Name: Type, dtype: object
```

Осуществим вызов метода `apply` с функцией `single_or_multi` в качестве аргумента. Библиотека `pandas` вызывает функцию `single_or_multi` для каждого из значений объекта `Series`:

```
In [63] pokemon.apply(single_or_multi)
```

```
Out [63] Pokemon
Bulbasaur      Multi
Ivysaur        Multi
Venusaur       Multi
Charmander     Single
Charmeleon     Single
...
Stakataka      Multi
Blacephalon    Multi
Zeraora        Single
Meltan         Single
Melmetal       Single
Name: Type, Length: 809, dtype: object
```

Первый наш покемон, `Bulbasaur`, классифицирован как покемон типа `Grass / Poison`, так что функция `single_or_multi` возвращает `"Multi"`. А четвертый, `Charmander`, — покемон типа `Fire`, так что наша функция возвращает `"Single"`. Та же логика применяется и к прочим значениям из объекта `pokemon`.

Мы получили новый объект `Series`! Давайте узнаем, сколько покемонов попало в каждую категорию, вызвав `value_counts`:

```
In [64] pokemon.apply(single_or_multi).value_counts()
```

```
Out [64] Multi      405
Single      404
Name: Type, dtype: int64
```

Оказывается, покемонов с одной и несколькими способностями примерно поровну. Надеюсь, что эта информация когда-нибудь пригодится вам в жизни.

3.6. УПРАЖНЕНИЕ

Попробуйте выполнить упражнение, требующее применения различных идей из этой главы и главы 2.

3.6.1. Задача

Представьте себе, что к вам обратился историк, который хочет выяснить, на какой день недели приходилось наибольшее число битв во время Войны за независимость США. Результат должен представлять собой объект `Series` с днями недели (воскресенье, понедельник и т. д.) в качестве меток индекса, а количество приходящихся на соответствующий день битв — в качестве значений. Начните с самого начала — импортируйте набор данных `revolutionary_war.csv` и выполните необходимые операции, чтобы получить в конечном итоге следующие результаты:

Saturday	39
Friday	39
Wednesday	32
Thursday	31
Sunday	31
Tuesday	29
Monday	27

Для решения этой задачи вам понадобится знание еще одной возможности языка Python. Метод `strftime`, будучи вызванным для отдельного объекта даты с аргументом `"%A"`, возвращает день недели, на который приходится эта дата (например, `Sunday`). Подробнее об этом в примере ниже, а также в более детальном обзоре объектов даты/времени в приложении Б:

```
In [65] import datetime as dt
        today = dt.datetime(2020, 12, 26)
        today.strftime("%A")
```

```
Out [65] 'Saturday'
```

ПОДСКАЗКА

Удобно будет объявить пользовательскую функцию для вычисления дня недели по дате.

Удачи!

3.6.2. Решение

Давайте заново импортируем набор данных `revolutionary_war.csv` и вспомним, какая изначально была форма у его данных:

```
In [66] pd.read_csv("revolutionary_war.csv").head()
```

```
Out [66]
```

	Battle	Start Date	State
0	Powder Alarm	9/1/1774	Massachusetts
1	Storming of Fort William and Mary	12/14/1774	New Hampshire
2	Battles of Lexington and Concord	4/19/1775	Massachusetts
3	Siege of Boston	4/19/1775	Massachusetts
4	Gunpowder Incident	4/20/1775	Virginia

Для текущей задачи столбцы **Battle** и **State** нам не нужны. Можете использовать любой из них в качестве индекса или оставить вариант по умолчанию.

Важнейший этап — преобразование строковых значений в столбце **Start Date** в метки даты/времени. Для меток даты/времени можно вызывать методы работы с датами, для обычных строк такой возможности нет. Выбираем столбец **Start Date** с помощью параметра `usecols` и преобразуем его значения в метки даты/времени с помощью параметра `parse_dates`. Наконец, не забываем передать `True` в параметр `squeeze`, чтобы получить объект **Series** вместо **DataFrame**:

```
In [67] days_of_war = pd.read_csv(
    "revolutionary_war.csv",
    usecols = ["Start Date"],
    parse_dates = ["Start Date"],
    squeeze = True,
)
```

```
days_of_war.head()
```

```
Out [67] 0    1774-09-01
         1    1774-12-14
         2    1775-04-19
         3    1775-04-19
         4    1775-04-20
         Name: Start Date, dtype: datetime64[ns]
```

Следующая задача — получить дни недели для всех дат. Одно из решений (использующее только те инструменты, которые мы уже изучили в этой книге) — передать каждое из значений **Series** в функцию, возвращающую день недели для даты. Давайте объявим эту функцию:

```
In [68] def day_of_week(date):
         return date.strftime("%A")
```

Как нам теперь вызвать функцию `day_of_week` для каждого значения объекта `Series`? Очень просто: передать функцию `day_of_week` в качестве аргумента методу `apply`. И вроде бы мы должны получить дни недели, но...

```
In [69] days_of_war.apply(day_of_week)
-----
ValueError                                Traceback (most recent call last)
<ipython-input-411-c133befd2940> in <module>
----> 1 days_of_war.apply(day_of_week)
ValueError: NaTType does not support strftime
```

Упс... В нашем столбце `Start Date` были пропущены некоторые значения. В отличие от объектов даты/времени у объектов `NaT` нет метода `strftime`, и библиотека `pandas` сталкивается с ошибкой при попытке передать такой объект в функцию `day_of_week`. Простейшее решение: отбросить все отсутствующие значения даты/времени из объекта `Series` перед вызовом метода `apply`. Сделать это можно с помощью метода `dropna`:

```
In [70] days_of_war.dropna().apply(day_of_week)

Out [70] 0      Thursday
         1      Wednesday
         2      Wednesday
         3      Wednesday
         4      Thursday
         ...
        227     Wednesday
        228       Friday
        229       Friday
        230       Friday
        231     Wednesday
         Name: Start Date, Length: 228, dtype: object
```

Вот теперь у нас что-то получилось! А подсчитать число вхождений каждого дня недели можно с помощью метода `value_counts`:

```
In [71] days_of_war.dropna().apply(day_of_week).value_counts()

Out [71] Saturday      39
         Friday       39
         Wednesday    32
         Thursday     31
         Sunday       31
         Tuesday     29
         Monday      27
         Name: Start Date, dtype: int64
```

Идеально! Ницья в поединке между пятницей и субботой. Поздравляем с завершением упражнений!

РЕЗЮМЕ

- Функция `read_csv` импортирует содержимое CSV-файла в структуру данных библиотеки `pandas`.
- Параметры функции `read_csv` позволяют настраивать импортируемые столбцы, индекс, типы данных и многое другое.
- Метод `sort_values` сортирует значения объекта `Series` в порядке возрастания или убывания.
- Метод `sort_index` сортирует индекс объекта `Series` в порядке возрастания или убывания.
- Для присвоения исходной переменной, в которой хранился объект изначально, его копии, возвращаемой методом в качестве результата, служит параметр `inplace`. Никаких преимуществ в смысле производительности у использования параметра `inplace` нет.
- Метод `value_counts` подсчитывает число вхождений каждого уникального значения в объекте `Series`.
- Метод `apply` вызывает заданную функцию для каждого значения в объекте `Series` и возвращает результаты в новом объекте `Series`.

4

Объект *DataFrame*

В этой главе

- ✓ Создание экземпляров `DataFrame` из ассоциативных массивов и NumPy-объектов `ndarray`.
- ✓ Импорт объектов `DataFrame` из CSV-файлов с помощью функции `read_csv`.
- ✓ Сортировка столбцов объектов `DataFrame`.
- ✓ Доступ к строкам и столбцам `DataFrame`.
- ✓ Задание и замена индекса `DataFrame`.
- ✓ Переименование столбцов и меток индекса объектов `DataFrame`.

Объект `DataFrame` библиотеки `pandas` представляет собой двумерную таблицу данных со строками и столбцами. Аналогично объектам `Series` `pandas` присваивает метку индекса и позицию индекса каждому столбцу. Мы называем `DataFrame` двумерным, поскольку для указания конкретного значения из набора данных необходимы две координаты — строка и столбец. На рис. 4.1 показан наглядный пример объекта `DataFrame` библиотеки `pandas`.

Объект `DataFrame` — «рабочая лошадка» библиотеки `pandas` и структура данных, с которой вам предстоит работать больше всего каждый день, так что дальнейшая часть этой книги будет посвящена изучению его обширных возможностей.

	Столбец А	Столбец В
Строка А		
Строка В		
Строка С		
Строка D		
Строка Е		

Рис. 4.1. Визуальное представление объекта DataFrame с пятью строками и двумя столбцами

4.1. ОБЗОР DATAFRAME

Как обычно, откроем новый блокнот Jupyter и произведем импорт библиотеки pandas. Кроме этого, нам понадобится библиотека NumPy, чтобы генерировать случайные данные в подразделе 4.1.2. Для NumPy обычно задается псевдоним np:

```
In [1] import pandas as pd
        import numpy as np
```

Конструктор класса `DataFrame` доступен на верхнем уровне библиотеки pandas. Синтаксис создания экземпляра класса `DataFrame` идентичен синтаксису создания экземпляра `Series`. Обращаемся к классу `DataFrame` и создаем его экземпляр с помощью пары круглых скобок: `pd.DataFrame()`.

4.1.1. Создание объекта DataFrame на основе ассоциативного массива

Первый параметр конструктора, `data`, должен содержать данные для наполнения объекта `DataFrame`. Одно из возможных входных значений для него — ассоциативный массив Python с названиями столбцов будущего объекта `DataFrame` в роли ключей и значениями столбцов в роли его значений. Итак, передадим ассоциативный массив со строковыми ключами и списками значений. Для этого служит код ниже. После его отработки библиотека pandas возвращает объект `DataFrame` с тремя столбцами. Каждый элемент списка становится значением в соответствующем столбце:

```
In [2] city_data = {
        "City": ["New York City", "Paris", "Barcelona", "Rome"],
        "Country": ["United States", "France", "Spain", "Italy"],
```

122 Часть I. Основы pandas

```
"Population": [8600000, 2141000, 5515000, 2873000]
}

cities = pd.DataFrame(city_data)
cities
```

Out [2]

	City	Country	Population
0	New York City	United States	8600000
1	Paris	France	2141000
2	Barcelona	Spain	5515000
3	Rome	Italy	2873000

Получен первый настоящий объект **DataFrame**! Обратите внимание, что эта структура данных визуализируется иначе, чем **Series**.

Объект **DataFrame** содержит индекс для меток строк. Мы не указали при вызове конструктора пользовательский индекс, и pandas распорядилась самостоятельно — сгенерировала числовой индекс, начинающийся с 0. Логика та же самая, что и с **Series**.

Объект **DataFrame** может содержать несколько столбцов данных. Удобно рассматривать заголовки столбцов как второй индекс. **City**, **Country** и **Population** — три метки индекса на оси столбцов; pandas ставит им в соответствие позиции индекса 0, 1 и 2 соответственно.

А что, если понадобится поменять заголовки столбцов и метки индекса местами? Сделать это можно двумя способами. Либо вызвать метод **transpose** объекта **DataFrame**, либо обратиться к его атрибуту **T**:

```
In [3] # Две строки ниже эквивалентны
        cities.transpose()
        cities.T
```

Out [3]

	0	1	2	3
City	New York City	Paris	Barcelona	Rome
Country	United States	France	Spain	Italy
Population	8600000	2141000	5515000	2873000

Предыдущий пример — напоминание о том, что библиотека pandas может хранить метки индекса различных типов. В выведенных результатах значения для меток индекса и позиций индекса совпадают. У строк же метки (**City**, **Country**, **Population**) и позиции (0, 1 и 2) различаются.

4.1.2. Создание объекта DataFrame на основе ndarray библиотеки NumPy

Приведу еще один пример. В параметр `data` конструктора `DataFrame` можно также передать объект `ndarray` библиотеки NumPy. Сгенерировать `ndarray` произвольного размера можно с помощью функции `randint` из модуля `random` библиотеки NumPy. В следующем примере мы создаем `ndarray` целых чисел в диапазоне от 1 (включительно) до 101 (не включительно) размером 3×5 :

```
In [4] random_data = np.random.randint(1, 101, [3, 5])
      random_data
```

```
Out [4] array([[25, 22, 80, 43, 42],
               [40, 89,  7, 21, 25],
               [89, 71, 32, 28, 39]])
```

Дополнительную информацию о генерации случайных данных в NumPy можно найти в приложении В.

Далее передадим полученный объект `ndarray` в конструктор `DataFrame`. Наш объект `ndarray` не содержит меток ни строк, ни столбцов. Следовательно, pandas создаст числовой индекс как для оси строк, так и для оси столбцов:

```
In [5] pd.DataFrame(data = random_data)
```

```
Out [5]
```

	0	1	2	3	4
0	25	22	80	43	42
1	40	89	7	21	25
2	89	71	32	28	39

Можно вручную задать метки строк с помощью параметра `index` конструктора `DataFrame`, принимающего любой итерируемый объект, включая списки, кортежи и `ndarray`. Обратите внимание, что длина этого итерируемого объекта должна совпадать с количеством строк набора данных. Мы передаем объект `ndarray` размером 3×5 , так что должны указать три метки строк:

```
In [6] row_labels = ["Morning", "Afternoon", "Evening"]
      temperatures = pd.DataFrame(
          data = random_data, index = row_labels
      )
      temperatures
```

```
Out [6]
```

	0	1	2	3	4
Morning	25	22	80	43	42
Afternoon	40	89	7	21	25
Evening	89	71	32	28	39

Задать названия столбцов можно с помощью параметра `columns` конструктора. Наш объект `ndarray` содержит пять столбцов, значит, необходимо передать итерируемый объект, содержащий пять элементов. В примере ниже передаем названия столбцов в виде кортежа:

```
In [7] row_labels = ["Morning", "Afternoon", "Evening"]
      column_labels = (
          "Monday",
          "Tuesday",
          "Wednesday",
          "Thursday",
          "Friday",
      )

      pd.DataFrame(
          data = random_data,
          index = row_labels,
          columns = column_labels,
      )
```

Out [7]

	Monday	Tuesday	Wednesday	Thursday	Friday
Morning	25	22	80	43	42
Afternoon	40	89	7	21	25
Evening	89	71	32	28	39

Библиотека `pandas` допускает дублирование индексов столбцов и строк. Например, `"Morning"` встречается в примере дважды в метках индекса для строк, а `"Tuesday"` — дважды в метках индекса для столбцов:

```
In [8] row_labels = ["Morning", "Afternoon", "Morning"]
      column_labels = [
          "Monday",
          "Tuesday",
          "Wednesday",
          "Tuesday",
          "Friday"
      ]

      pd.DataFrame(
          data = random_data,
          index = row_labels,
          columns = column_labels,
      )
```

Out [8]

	Monday	Tuesday	Wednesday	Tuesday	Friday
Morning	25	22	80	43	42
Afternoon	40	89	7	21	25
Evening	89	71	32	28	39

Как уже упоминалось в предыдущих главах, лучше, если индексы уникальны. Библиотеке pandas легче извлечь конкретную строку или столбец, если индексы не дублируются.

4.2. ОБЩИЕ ЧЕРТЫ SERIES И DATAFRAME

Многие атрибуты и методы класса `Series` присутствуют и в классе `DataFrame`. Их реализация, впрочем, может отличаться: библиотека pandas должна учитывать во втором случае наличие у объекта нескольких столбцов и двух отдельных осей координат.

4.2.1. Импорт объекта DataFrame с помощью функции read_csv

Набор данных `nba.csv` представляет собой список баскетболистов-профессионалов в НБА в сезоне 2019–2020 годов. Каждая строка включает имя игрока, команду, позицию на поле, день его рождения и зарплату. Благодаря комбинации различных типов данных этот набор данных замечательно подходит для изучения основ `DataFrame`.

Воспользуемся функцией `read_csv`, доступной на верхнем уровне библиотеки pandas, для импорта нашего файла (я познакомил вас с этой функцией в главе 3). В качестве первого аргумента эта функция принимает имя файла и возвращает по умолчанию объект `DataFrame`. Прежде чем выполнять следующий код, убедитесь, что файл с набором данных находится в том же каталоге, что и блокнот Jupyter:

In [9] `pd.read_csv("nba.csv")`

Out [9]

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	9/26/96	1445697
1	Christian Wood	Detroit Pistons	PF	9/27/95	1645357
2	PJ Washington	Charlotte Hornets	PF	8/23/98	3831840
3	Derrick Rose	Detroit Pistons	PG	10/4/88	7317074

```

4      Marial Shayok    Philadelphia 76ers          G    7/26/95      79568
...      ...          ...          ...      ...      ...
445    Austin Rivers    Houston Rockets          PG    8/1/92      2174310
446      Harry Giles    Sacramento Kings          PF    4/22/98      2578800
447      Robin Lopez    Milwaukee Bucks          C     4/1/88      4767000
448    Collin Sexton    Cleveland Cavaliers          PG    1/4/99      4764960
449      Ricky Rubio    Phoenix Suns          PG   10/21/90     16200000

```

450 rows × 5 columns

Внизу, в завершающей строке при выдаче этих результатов pandas сообщила, что набор данных содержит 450 строк и 5 столбцов.

Прежде чем присвоить полученный объект `DataFrame` переменной, проведем одну оптимизацию. Библиотека pandas импортирует столбец `Birthday` в виде строковых значений, а не меток даты/времени, что сильно ограничивает количество возможных операций над ними. Для преобразования значений в метки даты/времени можно воспользоваться параметром `parse_dates`:

```
In [10] pd.read_csv("nba.csv", parse_dates = ["Birthday"])
```

Out [10]

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
2	PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
3	Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
4	Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568
...
445	Austin Rivers	Houston Rockets	PG	1992-08-01	2174310
446	Harry Giles	Sacramento Kings	PF	1998-04-22	2578800
447	Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
448	Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
449	Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000

450 rows × 5 columns

Вот теперь гораздо лучше! Столбец состоит из меток даты/времени. Pandas отображает метки даты/времени в общепринятом формате ГГГГ-ММ-ДД. Импортированные данные отвечают нашим ожиданиям, так что можно присвоить полученный объект `DataFrame` переменной, например `nba`:

```
In [11] nba = pd.read_csv("nba.csv", parse_dates = ["Birthday"])
```

Объект `DataFrame` удобно рассматривать как набор объектов `Series` с общим индексом. В этом примере у пяти столбцов в `nba` (`Name`, `Team`, `Position`, `Birthday` и `Salary`) общий индекс для строк. Приступим к исследованию этого `DataFrame`.

4.2.2. Атрибуты Series и DataFrame: сходство и различие

Атрибуты и методы `Series` и `DataFrame` могут различаться как по названию, так и по реализации. Вот пример. У объектов `Series` есть атрибут `dtype`, содержащий тип данных их значений (см. главу 2). Обратите внимание, что он назван в единственном числе, поскольку объект `Series` может содержать данные только одного типа:

```
In [12] pd.Series([1, 2, 3]).dtype
```

```
Out [12] dtype('int64')
```

Объект `DataFrame`, в свою очередь, может содержать неоднородные данные. Слово «неоднородные» (heterogeneous) означает данные смешанных или различающихся типов. В одной строке могут содержаться целочисленные значения, а в другой — строковые. У объекта `DataFrame` есть свой особый атрибут `dtypes` (обратите внимание на множественное число этого названия). Этот атрибут возвращает объект `Series` со столбцами объекта `DataFrame` в качестве меток индекса и типами данных столбцов в качестве значений:

```
In [13] nba.dtypes
```

```
Out [13] Name          object
         Team          object
         Position      object
         Birthday      datetime64[ns]
         Salary        int64
         dtype: object
```

Для столбцов `Name`, `Team` и `Position` указан тип данных `object`. Тип данных `object` — внутреннее обозначение библиотеки `pandas` для сложных объектов, включая строковые значения. Таким образом, `DataFrame nba` содержит три строковых столбца, один столбец с датой/временем и один столбец с целочисленными значениями.

Можно вызвать метод `value_counts` этого объекта `Series`, чтобы подсчитать число столбцов с данными каждого типа:

```
In [14] nba.dtypes.value_counts()
```

```
Out [14] object          3
         datetime64[ns]  1
         int64           1
         dtype: int64
```

`dtype` и `dtypes` — один из примеров различий атрибутов `Series` и `DataFrame`. Но у этих структур данных есть и много общих атрибутов и методов.

Объект `DataFrame` состоит из нескольких меньших объектов: индекса с метками строк, индекса с метками столбцов и контейнера данных со значениями. Через атрибут `index` можно обращаться к индексу объекта `DataFrame`:

```
In [15] nba.index
```

```
Out [15] RangeIndex(start=0, stop=450, step=1)
```

Получаем объект `RangeIndex` — индекс, оптимизированный для хранения последовательности числовых значений. Объект `RangeIndex` включает три атрибута: `start` (нижняя граница, включительно), `stop` (верхняя граница, не включительно) и `step` (интервал/шаг между двумя соседними значениями). Из результатов выше видно, что индекс объекта `nba` начинается с 0 и доходит до 450, с приращением 1.

Для хранения столбцов `DataFrame` в библиотеке pandas служит отдельный объект для индекса, обращаться к которому можно через атрибут `columns`:

```
In [16] nba.columns
```

```
Out [16] Index(['Name', 'Team', 'Position', 'Birthday', 'Salary'],  
              dtype='object')
```

Это еще один тип объекта для индекса: `Index`, используемый pandas, когда индекс состоит из текстовых значений.

`index` — пример атрибута, существующего как у объектов `DataFrame`, так и объектов `Series`. `columns` — пример атрибута, который есть исключительно у объектов `DataFrame`, ведь у объектов `Series` нет понятия столбцов.

Атрибут `ndim` возвращает размерность объекта pandas. Размерность `DataFrame` — два:

```
In [17] nba.ndim
```

```
Out [17] 2
```

Атрибут `shape` возвращает размеры объекта `DataFrame` по измерениям в виде кортежа. Набор данных `nba` содержит 450 строк и 5 столбцов:

```
In [18] nba.shape
```

```
Out [18] (450, 5)
```

Атрибут `size` вычисляет общее количество значений в наборе данных. В это число включаются и отсутствующие (пустые) значения:

```
In [19] nba.size
```

```
Out [19] 2250
```


Чтобы исключить отсутствующие значения, можно воспользоваться методом `count`, который возвращает объект `Series` с количеством имеющихся значений по столбцам:

```
In [20] nba.count()
```

```
Out [20] Name      450
         Team      450
         Position  450
         Birthday  450
         Salary    450
         dtype: int64
```

Если сложить эти значения с помощью метода `sum()`, мы получим число непустых значений в объекте `DataFrame`. В наборе данных `nba` нет отсутствующих значений, так что атрибут `size` и метод `sum` возвращают одно и то же значение:

```
In [21] nba.count().sum()
```

```
Out [21] 2250
```

А вот пример, иллюстрирующий разницу между атрибутом `size` и методом `sum`. Создадим объект `DataFrame` с отсутствующим значением. При этом мы воспользуемся `nan` — атрибутом верхнего уровня пакета `NumPy`:

```
In [22] data = {
         "A": [1, np.nan],
         "B": [2, 3]
       }

         df = pd.DataFrame(data)
         df
```

```
Out [22]
```

```
      A  B
-----
0  1.0  2
1  NaN  3
```

Атрибут `size` возвращает 4, поскольку `DataFrame` содержит четыре ячейки:

```
In [23] df.size
```

```
Out [23] 4
```

А метод `sum` возвращает 3, поскольку непустых значений в этом объекте `DataFrame` только три:

```
In [24] df.count()
```

```
Out [24] A      1
```

```
B      2
dtype: int64
```

```
In [25] df.count().sum()
```

```
Out [25] 3
```

В столбце A содержится одно непустое значение, а в столбце B — два.

4.2.3. Общие методы Series и DataFrame

У объектов `DataFrame` и `Series` есть и общие методы. Например, для извлечения строк из начала объекта `DataFrame` можно воспользоваться методом `head`:

```
In [26] nba.head(2)
```

```
Out [26]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia	76ers	SG 1996-09-26	1445697
1	Christian Wood	Detroit	Pistons	PF 1995-09-27	1645357

Метод `tail` возвращает строки с конца объекта `DataFrame`:

```
In [27] nba.tail(n = 3)
```

```
Out [27]
```

	Name	Team	Position	Birthday	Salary
447	Robin Lopez	Milwaukee	Bucks	C 1988-04-01	4767000
448	Collin Sexton	Cleveland	Cavaliers	PG 1999-01-04	4764960
449	Ricky Rubio	Phoenix	Suns	PG 1990-10-21	16200000

По умолчанию при вызове без аргумента оба эти метода возвращают пять строк¹:

```
In [28] nba.tail()
```

```
Out [28]
```

	Name	Team	Position	Birthday	Salary
445	Austin Rivers	Houston	Rockets	PG 1992-08-01	2174310
446	Harry Giles	Sacramento	Kings	PF 1998-04-22	2578800
447	Robin Lopez	Milwaukee	Bucks	C 1988-04-01	4767000
448	Collin Sexton	Cleveland	Cavaliers	PG 1999-01-04	4764960
449	Ricky Rubio	Phoenix	Suns	PG 1990-10-21	16200000

¹ Как уже упоминалось в сноске к подразделу 1.3.1, это количество можно настраивать. — *Примеч. пер.*

Метод `sample` извлекает из объекта `DataFrame` случайные строки. Первый его параметр задает количество извлекаемых строк:

```
In [29] nba.sample(3)
```

```
Out [29]
```

Salary	Name	Team	Position	Birthdate
225	Tomas Satoransky	Chicago Bulls	PG	1991-10-30 10000000
201	Javonte Green	Boston Celtics	SF	1993-07-23 898310
310	Matthew Dellavedova	Cleveland Cavaliers	PG	1990-09-08 9607500

Предположим, мы хотим узнать, сколько команд, зарплат и позиций на поле в этом наборе данных. В главе 2 для подсчета количества уникальных значений в объекте `Series` мы использовали метод `nunique`. Тот же метод, вызванный для объекта `DataFrame`, возвращает объект `Series` с количеством уникальных значений в каждом столбце:

```
In [30] nba.nunique()
```

```
Out [30] Name      450
         Team       30
         Position    9
         Birthdate  430
         Salary     269
         dtype: int64
```

В `nba` 30 уникальных команд, 269 зарплат и 9 уникальных позиций на поле.

Наверное, вы помните также методы `max` и `min`. Метод `max`, вызванный для `DataFrame`, возвращает объект `Series` с максимальным значением для каждого столбца. Максимальное значение текстового столбца — строковое значение, ближайшее к концу алфавита. Максимальное значение столбца даты/времени — последняя дата в хронологическом порядке:

```
In [31] nba.max()
```

```
Out [31] Name      Zylan Cheatham
         Team      Washington Wizards
         Position      SG
         Birthdate  2000-12-23 00:00:00
         Salary      40231758
         dtype: object
```

Метод `min` возвращает объект `Series` с минимальным значением из каждого столбца (то есть наименьшим числом, ближайшим к началу алфавита строковым значением, самой ранней датой и т. д.):

```
In [32] nba.min()
```

```
Out [32] Name      Aaron Gordon
         Team      Atlanta Hawks
```

```

Position          C
Birthday    1977-01-26 00:00:00
Salary          79568
dtype: object

```

А как быть, если нужно найти несколько максимальных значений, например определить четырех самых высокооплачиваемых игроков в нашем наборе данных? На помощь приходит метод `nlargest`: он извлекает подмножество строк, значение заданного столбца в которых в объекте `DataFrame` максимально. В качестве аргумента (параметра `n`) передается число извлекаемых строк и столбец для сортировки в параметре `columns`. В следующем примере из объекта `DataFrame` извлекаются строки с четырьмя самыми высокими зарплатами в столбце `Salary`:

```
In [33] nba.nlargest(n = 4, columns = "Salary")
```

```
Out [33]
```

	Name	Team	Position	Birthday	Salary
205	Stephen Curry	Golden State Warriors	PG	1988-03-14	40231758
38	Chris Paul	Oklahoma City Thunder	PG	1985-05-06	38506482
219	Russell Westbrook	Houston Rockets	PG	1988-11-12	38506482
251	John Wall	Washington Wizards	PG	1990-09-06	38199000

Следующая задача: найти трех самых старших игроков в лиге. Для ее решения необходимо выделить три самые ранние даты рождения в столбце `Birthday`. В этом нам поможет метод `nsmallest`; он возвращает подмножество строк, значение заданного столбца в которых минимально в наборе данных. Минимальные значения даты/времени — самые ранние в хронологическом порядке. Обратите внимание, что методы `nlargest` и `nsmallest` можно вызывать только для числовых столбцов и столбцов с датами:

```
In [34] nba.nsmallest(n = 3, columns = ["Birthday"])
```

```
Out [34]
```

	Name	Team	Position	Birthday	Salary
98	Vince Carter	Atlanta Hawks	PF	1977-01-26	2564753
196	Udonis Haslem	Miami Heat	C	1980-06-09	2564753
262	Kyle Korver	Milwaukee Bucks	PF	1981-03-17	6004753

А если нужно вычислить сумму всех зарплат по НБА? Для этой цели `DataFrame` содержит метод `sum`:

```
In [35] nba.sum()
```

```

Out [35] Name      Shake MiltonChristian WoodPJ WashingtonDerrick...
         Team      Philadelphia 76ersDetroit PistonsCharlotte Hor...

```

```

Position    SGPFPFPGGPFSGSFCSFPGPGFCPGSGPFCCPFPGSGPFGSGSF...
Salary                                             3444112694
dtype: object

```

Нужный результат получен, но вид выводимой информации таков, что ничего нельзя понять, путаница полнейшая. А все потому, что по умолчанию библиотека pandas складывает значения в каждом из столбцов. В случае текстовых столбцов библиотека pandas производит конкатенацию всех строковых значений в одно, что и нашло отражение в выведенном результате. Чтобы ограничить суммирование только числовыми значениями, можно передать аргумент `True` для параметра `numeric_only` метода `sum`:

```
In [36] nba.sum(numeric_only = True)
```

```
Out [36] Salary    3444112694
dtype: int64
```

Совокупная зарплата всех 450 игроков НБА ошеломляющая: 3,4 миллиарда долларов. Подсчитать среднюю зарплату можно с помощью метода `mean`. Он также позволяет ограничиться только числовыми столбцами с помощью того же параметра `numeric_only`:

```
In [37] nba.mean(numeric_only = True)
```

```
Out [37] Salary    7.653584e+06
dtype: float64
```

`DataFrame` включает и методы для статистических расчетов, таких как вычисление медианы, моды и стандартного отклонения:

```
In [38] nba.median(numeric_only = True)
```

```
Out [38] Salary    3303074.5
dtype: float64
```

```
In [39] nba.mode(numeric_only = True)
```

```
Out [39]
Salary
-----
0    79568
```

```
In [40] nba.std(numeric_only = True)
```

```
Out [40] Salary    9.288810e+06
dtype: float64
```

Перечень и описание расширенных статистических методов можно найти в официальной документации `Series` (<http://mng.bz/myDa>).

4.3. СОРТИРОВКА ОБЪЕКТА DATAFRAME

Строки нашего набора данных поступили в него в беспорядочном, случайном порядке, но это не проблема! С помощью метода `sort_values` можно отсортировать объект `DataFrame` по одному или нескольким столбцам.

4.3.1. Сортировка по одному столбцу

Сначала отсортируем игроков по имени в алфавитном порядке. В первом параметре метода `sort_values` можно указать столбец, по которому библиотека pandas должна сортировать `DataFrame`. Передадим в него столбец `Name` в виде строкового значения:

```
In [41] # Две строки ниже эквивалентны
        nba.sort_values("Name")
        nba.sort_values(by = "Name")
```

Out [41]

	Name	Team	Position	Birthday	Salary

52	Aaron Gordon	Orlando Magic	PF	1995-09-16	19863636
101	Aaron Holiday	Indiana Pacers	PG	1996-09-30	2239200
437	Abdel Nader	Oklahoma City Thunder	SF	1993-09-25	1618520
81	Adam Mokoka	Chicago Bulls	G	1998-07-18	79568
399	Admiral Schofield	Washington Wizards	SF	1997-03-30	1000000
...
159	Zach LaVine	Chicago Bulls	PG	1995-03-10	19500000
302	Zach Norvell	Los Angeles Lakers	SG	1997-12-09	79568
312	Zhaire Smith	Philadelphia 76ers	SG	1999-06-04	3058800
137	Zion Williamson	New Orleans Pelicans	F	2000-07-06	9757440
248	Zylan Cheatham	New Orleans Pelicans	SF	1995-11-17	79568

450 rows × 5 columns

Параметр `ascending` метода `sort_values` определяет направление сортировки; по умолчанию он равен `True`, то есть по умолчанию pandas сортирует столбец чисел в порядке возрастания, столбец строковых значений — в алфавитном порядке, а столбец меток даты/времени — в хронологическом порядке.

Чтобы отсортировать названия в обратном алфавитном порядке, необходимо передать в параметр `ascending` аргумент `False`:

```
In [42] nba.sort_values("Name", ascending = False).head()
```

Out [42]

	Name	Team	Position	Birthday	Salary
248	Zylan Cheatham	New Orleans Pelicans	SF	1995-11-17	79568
137	Zion Williamson	New Orleans Pelicans	F	2000-07-06	9757440
312	Zhaire Smith	Philadelphia 76ers	SG	1999-06-04	3058800
302	Zach Norvell	Los Angeles Lakers	SG	1997-12-09	79568
159	Zach LaVine	Chicago Bulls	PG	1995-03-10	19500000

И еще один пример: как найти пять самых молодых игроков НБА без применения метода `nsmallest`? Да просто отсортировать столбец `Birthday` в обратном хронологическом порядке с помощью метода `sort_values` со значением `False` параметра `ascending`, после чего извлечь пять первых строк с помощью метода `head`:

```
In [43] nba.sort_values("Birthday", ascending = False).head()
```

```
Out [43]
```

	Name	Team	Position	Birthday	Salary
136	Sekou Doumbouya	Detroit Pistons	SF	2000-12-23	3285120
432	Talen Horton-Tucker	Los Angeles Lakers	GF	2000-11-25	898310
137	Zion Williamson	New Orleans Pelicans	F	2000-07-06	9757440
313	RJ Barrett	New York Knicks	SG	2000-06-14	7839960
392	Jalen Lecque	Phoenix Suns	G	2000-06-13	898310

Самый молодой из игроков НБА выводится первым, это Секу Думбуя, родившийся 23 декабря 2000 года.

4.3.2. Сортировка по нескольким столбцам

Можно сортировать несколько столбцов объекта `DataFrame`, передав в параметр `by` метода `sort_values` список этих столбцов. Библиотека `pandas` отсортирует столбцы объекта `DataFrame` последовательно, в том порядке, в котором они встречаются в списке. В следующем примере мы сортируем объект `DataFrame` `nba` сначала по столбцу `Team`, а потом по столбцу `Name`. Библиотека `pandas` по умолчанию сортирует все столбцы по возрастанию:

```
In [44] nba.sort_values(by = ["Team", "Name"])
```

```
Out [44]
```

	Name	Team	Position	Birthday	Salary
359	Alex Len	Atlanta Hawks	C	1993-06-16	4160000
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000

```

276 Brandon Goodwin      Atlanta Hawks      PG 1995-10-02      79568
438 Bruno Fernando       Atlanta Hawks      C 1998-08-15      1400000
194 Cam Reddish          Atlanta Hawks      SF 1999-09-01      4245720
...
418 Jordan McRae         Washington Wizards PG 1991-03-28      1645357
273 Justin Robinson     Washington Wizards PG 1997-10-12      898310
428 Moritz Wagner       Washington Wizards C 1997-04-26      2063520
21 Rui Hachimura         Washington Wizards PF 1998-02-08      4469160
36 Thomas Bryant        Washington Wizards C 1997-07-31      8000000

```

```
450 rows × 5 columns
```

Вот как следует читать результаты. При сортировке команд в алфавитном порядке первая команда в наборе данных — «Атланта Хокс». В команде «Атланта Хокс» первым по алфавиту идет имя Алекса Лены, за ним следуют Аллен Крэббе и Брэндон Гудвин. Эту логику сортировки библиотека pandas повторяет для всех оставшихся команд и имен внутри каждой команды.

Достаточно передать один булев аргумент для параметра `ascending`, чтобы применить соответствующий порядок сортировки ко всем столбцам. В следующем примере передадим `False`, поэтому библиотека pandas сортирует сначала столбец `Team` в порядке убывания, а потом столбец `Name` в порядке убывания:

```
In [45] nba.sort_values(["Team", "Name"], ascending = False)
```

```
Out [45]
```

	Name	Team	Position	Birthday	Salary
36	Thomas Bryant	Washington Wizards	C	1997-07-31	8000000
21	Rui Hachimura	Washington Wizards	PF	1998-02-08	4469160
428	Moritz Wagner	Washington Wizards	C	1997-04-26	2063520
273	Justin Robinson	Washington Wizards	PG	1997-10-12	898310
418	Jordan McRae	Washington Wizards	PG	1991-03-28	1645357
...
194	Cam Reddish	Atlanta Hawks	SF	1999-09-01	4245720
438	Bruno Fernando	Atlanta Hawks	C	1998-08-15	1400000
276	Brandon Goodwin	Atlanta Hawks	PG	1995-10-02	79568
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000
359	Alex Len	Atlanta Hawks	C	1993-06-16	4160000

```
450 rows × 5 columns
```

А что делать, если нужно отсортировать столбцы в разном порядке? Например, команды в порядке возрастания, а затем зарплаты в этих командах в порядке убывания. Для этого можно передать в параметр `ascending` список булевых значений. Длины передаваемых в параметры `by` и `ascending` списков должны совпадать. Библиотека pandas сопоставляет столбцы с порядком их сортировки по соответствию позиций индекса в последовательности. В следующем примере столбец `Team` в списке параметра `by` занимает позицию индекса 0; pandas ставит

ему в соответствии `True` на позиции индекса 0 в списке параметра `ascending`, так что сортирует этот столбец в порядке возрастания. А затем `pandas` применяет ту же логику к столбцу `Salary` и сортирует его в порядке убывания:

```
In [46] nba.sort_values(
        by = ["Team", "Salary"], ascending = [True, False]
    )
```

Out [46]

	Name	Team	Position	Birthday	Salary
111	Chandler Parsons	Atlanta Hawks	SF	1988-10-25	25102512
28	Evan Turner	Atlanta Hawks	PG	1988-10-27	18606556
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000
213	De'Andre Hunter	Atlanta Hawks	SF	1997-12-02	7068360
339	Jabari Parker	Atlanta Hawks	PF	1995-03-15	6500000
...
80	Isaac Bonga	Washington Wizards	PG	1999-11-08	1416852
399	Admiral Schofield	Washington Wizards	SF	1997-03-30	1000000
273	Justin Robinson	Washington Wizards	PG	1997-10-12	898310
283	Garrison Mathews	Washington Wizards	SG	1996-10-24	79568
353	Chris Chiozza	Washington Wizards	PG	1995-11-21	79568

450 rows × 5 columns

Такие данные нас устраивают, так что сохраним результаты этой сортировки. Метод `sort_values` поддерживает параметр `inplace`, но лучше давайте присвоим возвращаемый `DataFrame` той же переменной `nba` явным образом (см. обсуждение недостатков параметра `inplace` в главе 3):

```
In [47] nba = nba.sort_values(
        by = ["Team", "Salary"],
        ascending = [True, False]
    )
```

Ура! Объект `DataFrame` отсортирован по значениям столбцов `Team` и `Salary`. Теперь можно легко увидеть, какие игроки в каждой из команд — самые высокооплачиваемые.

4.4. СОРТИРОВКА ПО ИНДЕКСУ

С нашей новой сортировкой `DataFrame` выглядит иначе, чем сразу после импорта:

```
In [48] nba.head()
```

Out [48]

	Name	Team	Position	Birthday	Salary
111	Chandler Parsons	Atlanta Hawks	SF	1988-10-25	25102512
28	Evan Turner	Atlanta Hawks	PG	1988-10-27	18606556
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000
213	De'Andre Hunter	Atlanta Hawks	SF	1997-12-02	7068360
339	Jabari Parker	Atlanta Hawks	PF	1995-03-15	6500000

А как вернуть его в исходный вид?

4.4.1. Сортировка по индексу строк

В объекте `DataFrame` `nba` сохранился числовой индекс. Если отсортировать набор данных по позициям индекса, а не значений столбцов, он вернется в исходное состояние. Сделать это можно с помощью метода `sort_index`:

```
In [49] # Две строки ниже эквивалентны
        nba.sort_index().head()
        nba.sort_index(ascending = True).head()
```

Out [49]

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
2	PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
3	Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
4	Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568

Можно также отсортировать в обратном порядке, передав аргумент `False` для параметра `ascending` этого метода. В следующем примере сначала выводятся максимальные значения позиций индекса:

```
In [50] nba.sort_index(ascending = False).head()
```

Out [50]

	Name	Team	Position	Birthday	Salary
449	Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000
448	Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
447	Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
446	Harry Giles	Sacramento Kings	PF	1998-04-22	2578800
445	Austin Rivers	Houston Rockets	PG	1992-08-01	2174310

Мы вернулись к тому, с чего начинали, — к отсортированному по позициям индекса объекту `nba`. Присвоим полученный объект `DataFrame` обратно переменной `nba`:

```
In [51] nba = nba.sort_index()
```

А теперь посмотрим, как можно отсортировать наш `nba` по другим осям координат.

4.4.2. Сортировка по индексу столбцов

Объект `DataFrame` — двумерная структура данных. Ее можно отсортировать еще по одной оси координат — вертикальной.

Для сортировки столбцов объекта `DataFrame` по порядку мы снова воспользуемся методом `sort_index`. На этот раз, впрочем, необходимо добавить в его вызов параметр `axis` и передать ему аргумент `"columns"` или `1`. В следующем примере столбцы сортируются в порядке возрастания:

```
In [52] # Две строки ниже эквивалентны
        nba.sort_index(axis = "columns").head()
        nba.sort_index(axis = 1).head()

Out [52]
```

	Birthday	Name	Position	Salary	Team
0	1996-09-26	Shake Milton	SG	1445697	Philadelphia 76ers
1	1995-09-27	Christian Wood	PF	1645357	Detroit Pistons
2	1998-08-23	PJ Washington	PF	3831840	Charlotte Hornets
3	1988-10-04	Derrick Rose	PG	7317074	Detroit Pistons
4	1995-07-26	Marial Shayok	G	79568	Philadelphia 76ers

А как осуществить сортировку столбцов в обратном алфавитном порядке? Никаких проблем: просто передаем параметру `ascending` аргумент `False`. А вот и пример: ниже мы вызываем метод `sort_index` для сортировки столбцов (параметр `axis`) в порядке убывания (параметр `ascending`):

```
In [53] nba.sort_index(axis = "columns", ascending = False).head()

Out [53]
```

	Team	Salary	Position	Name	Birthday
0	Philadelphia 76ers	1445697	SG	Shake Milton	1996-09-26
1	Detroit Pistons	1645357	PF	Christian Wood	1995-09-27
2	Charlotte Hornets	3831840	PF	PJ Washington	1998-08-23
3	Detroit Pistons	7317074	PG	Derrick Rose	1988-10-04
4	Philadelphia 76ers	79568	G	Marial Shayok	1995-07-26

На секунду задумайтесь, насколько потрясающи возможности `pandas`. С помощью всего двух методов и нескольких параметров мы сумели отсортировать объект `DataFrame` по обеим осям координат, по одному столбцу, по нескольким столбцам, в порядке возрастания, в порядке убывания и даже

в различных порядках одновременно для разных столбцов. Гибкость библиотеки pandas удивительна. Нужно только взять нужный метод и задать нужные аргументы.

4.5. ЗАДАНИЕ НОВОГО ИНДЕКСА

Наш набор данных по своей сути представляет собой список игроков. Следовательно, логично будет воспользоваться значениями столбца `Name` в качестве меток индекса объекта `DataFrame`. К тому же `Name` — единственный столбец с уникальными значениями, что очень удобно.

Метод `set_index` возвращает новый объект `DataFrame` с заданным столбцом в качестве индекса. Название столбца задается в первом параметре метода, `keys`:

```
In [54] # Две строки ниже эквивалентны
        nba.set_index(keys = "Name")
        nba.set_index("Name")
```

Out [54]

	Team	Position	Birthday	Salary
Name				
Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568
...
Austin Rivers	Houston Rockets	PG	1992-08-01	2174310
Harry Giles	Sacramento Kings	PF	1998-04-22	2578800
Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000

450 rows × 4 columns

Сказано — сделано! Давайте перезапишем нашу переменную `nba`:

```
In [55] nba = nba.set_index(keys = "Name")
```

Замечу, что можно было задать индекс еще при импорте набора данных, достаточно было передать название столбца в виде строкового значения в параметр `index_col` функции `read_csv`. В результате выполнения следующего кода получается тот же `DataFrame`:

```
In [56] nba = pd.read_csv(
        "nba.csv", parse_dates = ["Birthday"], index_col = "Name"
    )
```

4.6. ИЗВЛЕЧЕНИЕ СТОЛБЦОВ ИЗ ОБЪЕКТОВ DATAFRAME

Объект `DataFrame` представляет собой набор объектов `Series` с общим индексом. Существует несколько вариантов синтаксиса извлечения одного или нескольких объектов `Series` из объекта `DataFrame`.

4.6.1. Извлечение одного столбца из объекта DataFrame

К любому столбцу `Series` можно обратиться как к атрибуту объекта `DataFrame`. Для доступа к атрибутам объекта применяется синтаксис с использованием точки. Например, можно извлечь столбец `Salary` с помощью оператора `nba.Salary`. Обратите внимание, что индекс переносится из объекта `DataFrame` в объект `Series`:

```
In [57] nba.Salary
```

```
Out [57] Name
      Shake Milton      1445697
      Christian Wood    1645357
      PJ Washington    3831840
      Derrick Rose     7317074
      Marial Shayok     79568
      ...
      Austin Rivers    2174310
      Harry Giles     2578800
      Robin Lopez     4767000
      Collin Sexton    4764960
      Ricky Rubio     16200000
      Name: Salary, Length: 450, dtype: int64
```

Можно также извлечь столбец путем передачи его названия в квадратных скобках после названия `DataFrame`:

```
In [58] nba["Position"]
```

```
Out [58] Name
      Shake Milton      SG
      Christian Wood    PF
      PJ Washington    PF
      Derrick Rose     PG
      Marial Shayok     G
      ..
      Austin Rivers    PG
      Harry Giles     PF
      Robin Lopez      C
      Collin Sexton    PG
      Ricky Rubio     PG
      Name: Position, Length: 450, dtype: object
```

Преимущества синтаксиса с квадратными скобками — поддержка названий столбцов, содержащих пробелы. Столбец с названием "Player Position", например, можно было бы извлечь только посредством квадратных скобок:

```
nba["Player Position"]
```

Название столбца, как вы могли заметить, представляет собой два слова, разделенных пробелом. Синтаксис с указанием атрибута привел бы к генерации исключения. Язык Python не знает, каков смысл этого пробела, он счел бы, что мы пытаемся обратиться к столбцу Player:

```
nba.Player Position
```

И хотя мнения по этому поводу разнятся, для извлечения данных я рекомендую синтаксис с квадратными скобками. Я отдаю предпочтение более универсальным однозначным решениям, которые работают во всех случаях, даже если при этом приходится написать несколько лишних символов.

4.6.2. Извлечение нескольких столбцов из объекта DataFrame

Для извлечения нескольких столбцов объекта `DataFrame` необходимо использовать список из названий столбцов, заключенный в квадратные скобки. В следующем примере мы извлекаем столбцы `Salary` и `Birthday`:

```
In [59] nba[["Salary", "Birthday"]]
```

```
Out [59]
```

Name	Salary	Birthday
Shake Milton	1445697	1996-09-26
Christian Wood	1645357	1995-09-27
PJ Washington	3831840	1998-08-23
Derrick Rose	7317074	1988-10-04
Marial Shayok	79568	1995-07-26

Библиотека `pandas` извлекает столбцы в порядке их следования в списке:

```
In [60] nba[["Birthday", "Salary"]].head()
```

```
Out [60]
```

Name	Birthday	Salary
Shake Milton	1996-09-26	1445697
Christian Wood	1995-09-27	1645357
PJ Washington	1998-08-23	3831840
Derrick Rose	1988-10-04	7317074
Marial Shayok	1995-07-26	79568

Для выбора столбцов по типам данных служит метод `select_dtypes`. Он принимает два параметра, `include` и `exclude`, аргументы которых могут представлять собой отдельные строковые значения или списки, указывающие типы столбцов, которые следует выбрать или отбросить. Напомню, что просмотреть типы данных столбцов `DataFrame` можно с помощью атрибута `dtypes`. В следующем примере мы извлекаем из `nba` только строковые столбцы:

```
In [61] nba.select_dtypes(include = "object")
```

```
Out [61]
```

Name	Team	Position
Shake Milton	Philadelphia 76ers	SG
Christian Wood	Detroit Pistons	PF
PJ Washington	Charlotte Hornets	PF
Derrick Rose	Detroit Pistons	PG
Marial Shayok	Philadelphia 76ers	G
...
Austin Rivers	Houston Rockets	PG
Harry Giles	Sacramento Kings	PF
Robin Lopez	Milwaukee Bucks	C
Collin Sexton	Cleveland Cavaliers	PG
Ricky Rubio	Phoenix Suns	PG

450 rows × 2 columns

А теперь извлечем все столбцы, кроме строковых и числовых:

```
In [62] nba.select_dtypes(exclude = ["object", "int64"])
```

```
Out [62]
```

Name	Birthday
Shake Milton	1996-09-26
Christian Wood	1995-09-27
PJ Washington	1998-08-23
Derrick Rose	1988-10-04
Marial Shayok	1995-07-26
...	...
Austin Rivers	1992-08-01
Harry Giles	1998-04-22
Robin Lopez	1988-04-01
Collin Sexton	1999-01-04
Ricky Rubio	1990-10-21

450 rows × 1 columns

Столбец `Birthday` — единственный в `nba`, данные в котором не относятся ни к строковым значениям, ни к целочисленным. Для включения в выборку или исключения из нее столбцов даты/времени можно передать в соответствующий параметр аргумент `"datetime"`.

4.7. ИЗВЛЕЧЕНИЕ СТРОК ИЗ ОБЪЕКТОВ DATAFRAME

Мы уже потренировались в извлечении столбцов, пора научиться извлекать строки объектов `DataFrame` по меткам или позициям индекса.

4.7.1. Извлечение строк по метке индекса

Атрибут `loc` служит для извлечения строки по метке. Атрибуты, подобные `loc`, называются *методами-получателями* (accessors)¹, поскольку они позволяют получить какой-либо элемент данных, сделать его доступным. Введите пару квадратных скобок сразу после `loc` и вставьте туда нужную метку индекса. В следующем примере мы извлекаем из `nba` строку с меткой индекса `"LeBron James"`. Библиотека `pandas` возвращает значения этой строки в виде объекта `Series`. Как всегда, не забывайте про чувствительность к регистру:

```
In [63] nba.loc["LeBron James"]

Out [63] Team           Los Angeles Lakers
         Position        PF
         Birthday    1984-12-30 00:00:00
         Salary           37436858
         Name: LeBron James, dtype: object
```

Для извлечения нескольких строк можно передать между квадратными скобками список. Результаты, включающие несколько записей, библиотека `pandas` сохраняет в объекте `DataFrame`:

```
In [64] nba.loc[["Kawhi Leonard", "Paul George"]]

Out [64]
```

Name	Team	Position	Birthday	Salary
Kawhi Leonard	Los Angeles Clippers	SF	1991-06-29	32742000
Paul George	Los Angeles Clippers	SF	1990-05-02	33005556

`Pandas` организует строки по порядку вхождения их меток индекса в список. В следующем примере мы меняем порядок строковых значений из предыдущего примера на обратный:

```
In [65] nba.loc[["Paul George", "Kawhi Leonard"]]

Out [65]
```

¹ В русскоязычной литературе также встречается название «метод доступа». — *Примеч. пер.*

Name	Team	Position	Birthday	Salary
Paul George	Los Angeles	Clippers	SF 1990-05-02	33005556
Kawhi Leonard	Los Angeles	Clippers	SF 1991-06-29	32742000

С помощью атрибута `loc` можно извлекать последовательность меток индекса. Синтаксис при этом напоминает синтаксис срезов списков языка Python. Указывается начальное значение, двоеточие и конечное значение. Для подобного извлечения данных я настоятельно рекомендую сначала отсортировать индекс, чтобы pandas быстрее находила значения.

Допустим, мы хотим извлечь данные обо всех игроках от Отто Портера до Патрика Беверли. Для этого можно отсортировать индекс `DataFrame`, чтобы получить имена игроков в алфавитном порядке, а затем передать имена двух вышеупомянутых игроков методу-получателю `loc`. "Otto Porter" — нижняя граница диапазона, а "Patrick Beverley" — верхняя:

```
In [66] nba.sort_index().loc["Otto Porter":"Patrick Beverley"]
```

```
Out [66]
```

Name	Team	Position	Birthday	Salary
Otto Porter	Chicago	Bulls	SF 1993-06-03	27250576
PJ Dozier	Denver	Nuggets	PG 1996-10-25	79568
PJ Washington	Charlotte	Hornets	PF 1998-08-23	3831840
Pascal Siakam	Toronto	Raptors	PF 1994-04-02	2351838
Pat Connaughton	Milwaukee	Bucks	SG 1993-01-06	1723050
Patrick Beverley	Los Angeles	Clippers	PG 1988-07-12	12345680

Обратите внимание, что метод-получатель `loc` библиотеки pandas все-таки немного отличается от срезов списков языка Python. В частности, метод-получатель `loc` включает метку верхней границы, в то время как срезы списков Python — нет.

Вот небольшой пример. В следующем фрагменте кода мы извлекаем с помощью синтаксиса срезов списков элементы с индекса 0 по индекс 2 из списка, состоящего из трех элементов. При этом индекс 2 ("PJ Washington") не включается в результат:

```
In [67] players = ["Otto Porter", "PJ Dozier", "PJ Washington"]
        players[0:2]
```

```
Out [67] ['Otto Porter', 'PJ Dozier']
```

С помощью `loc` можно также извлечь строки, начиная с середины объекта `DataFrame` и до его конца. Для этого необходимо указать внутри квадратных скобок начальную метку индекса и двоеточие:

```
In [68] nba.sort_index().loc["Zach Collins":]
```

```
Out [68]
```

Name	Team	Position	Birthday	Salary

Zach Collins	Portland Trail Blazers	C	1997-11-19	4240200
Zach LaVine	Chicago Bulls	PG	1995-03-10	19500000
Zach Norvell	Los Angeles Lakers	SG	1997-12-09	79568
Zhaire Smith	Philadelphia 76ers	SG	1999-06-04	3058800
Zion Williamson	New Orleans Pelicans	F	2000-07-06	9757440
Zylan Cheatham	New Orleans Pelicans	SF	1995-11-17	79568

И в другую сторону — при помощи срезов `loc` можно извлечь строки с начала объекта `DataFrame` и до конкретной метки индекса. Для этого необходимо указать сначала двоеточие, а затем конечную метку индекса. Следующий пример возвращает всех игроков с начала набора данных и до Эла Хорфорда:

```
In [69] nba.sort_index().loc[:"Al Horford"]
```

```
Out [69]
```

Name	Team	Position	Birthday	Salary

Aaron Gordon	Orlando Magic	PF	1995-09-16	19863636
Aaron Holiday	Indiana Pacers	PG	1996-09-30	2239200
Abdel Nader	Oklahoma City Thunder	SF	1993-09-25	1618520
Adam Mokoka	Chicago Bulls	G	1998-07-18	79568
Admiral Schofield	Washington Wizards	SF	1997-03-30	1000000
Al Horford	Philadelphia 76ers	C	1986-06-03	28000000

Если соответствующей метки индекса в объекте `DataFrame` нет, pandas сгенерирует исключение:

```
In [70] nba.loc["Bugs Bunny"]
```

```
-----
KeyError                                Traceback (most recent call last)
```

```
KeyError: 'Bugs Bunny'
```

Как ясно из названия, исключение `KeyError` сообщает, что в заданной структуре данных отсутствует ключ.

4.7.2. Извлечение строк по позиции индекса

Метод-получатель `iloc` (index location — «место индекса») извлекает строки по позиции индекса, что удобно, если позиция строк в наборе данных важна. Синтаксис аналогичен синтаксису `loc`. Введите пару квадратных скобок сразу после `iloc` и укажите целочисленное значение. Pandas извлечет строку с соответствующим индексом:

```
In [71] nba.iloc[300]
```

```
Out [71] Team           Denver Nuggets
         Position        PF
         Birthday      1999-04-03 00:00:00
         Salary         1416852
         Name: Jarred Vanderbilt, dtype: object
```

Для выборки нескольких записей методу-получателю `iloc` можно передать список позиций индекса. В следующем примере мы извлекаем игроков на позициях индекса 100, 200, 300 и 400:

```
In [72] nba.iloc[[100, 200, 300, 400]]
```

```
Out [72]
```

Name	Team	Position	Birthday	Salary
Brian Bowen	Indiana Pacers	SG	1998-10-02	79568
Marco Belinelli	San Antonio Spurs	SF	1986-03-25	5846154
Jarred Vanderbilt	Denver Nuggets	PF	1999-04-03	1416852
Louis King	Detroit Pistons	F	1999-04-06	79568

С методом-получателем `iloc` можно использовать синтаксис срезов списков. Впрочем, обратите внимание, что `pandas` не включает в результат позицию индекса, указанную после двоеточия. В следующем примере мы передаем в метод срез `400:404`. `Pandas` включает строки на позициях с индексами 400, 401, 402 и 403, но не включает строку с индексом 404:

```
In [73] nba.iloc[400:404]
```

```
Out [73]
```

Name	Team	Position	Birthday	Salary
Louis King	Detroit Pistons	F	1999-04-06	79568
Kostas Antetokounmpo	Los Angeles Lakers	PF	1997-11-20	79568
Rodions Kurucs	Brooklyn Nets	PF	1998-02-05	1699236
Spencer Dinwiddie	Brooklyn Nets	PG	1993-04-06	10605600

Если не указывать число перед двоеточием, будут извлечены строки с самого начала объекта `DataFrame`. Так в примере ниже мы извлекаем строки с начала `nba` и до позиции с индексом 2, не включая ее:

```
In [74] nba.iloc[:2]
```

```
Out [74]
```

Name	Team	Position	Birthday	Salary
Shake Milton	Philadelphia	76ers	SG 1996-09-26	1445697
Christian Wood	Detroit	Pistons	PF 1995-09-27	1645357

Аналогично можно убрать число после двоеточия, чтобы извлечь строки вплоть до конца объекта `DataFrame`. В следующем примере мы извлекаем строки, начиная с позиции с индексом 447 и до конца `nba`:

```
In [75] nba.iloc[447:]
```

```
Out [75]
```

Name	Team	Position	Birthday	Salary
Robin Lopez	Milwaukee	Bucks	C 1988-04-01	4767000
Collin Sexton	Cleveland	Cavaliers	PG 1999-01-04	4764960
Ricky Rubio	Phoenix	Suns	PG 1990-10-21	16200000

Можно также указывать отрицательные значения — одно или оба. Например, извлечем строки от десятой с конца до шестой с конца, не включая последнюю:

```
In [76] nba.iloc[-10:-6]
```

```
Out [76]
```

Name	Team	Position	Birthday	Salary
Jared Dudley	Los Angeles	Lakers	PF 1985-07-10	2564753
Max Strus	Chicago	Bulls	SG 1996-03-28	79568
Kevon Looney	Golden State	Warriors	C 1996-02-06	4464286
Willy Hernangomez	Charlotte	Hornets	C 1994-05-27	1557250

Можно указать и третье число внутри квадратных скобок, оно служит для задания целочисленной константы, на которую будет изменяться позиция индекса при извлечении каждой следующей строки. Приведу пример извлечения десяти первых строк `nba` с шагом 2. Полученный объект `DataFrame` включает строки с позициями индекса 0, 2, 4, 6 и 8:

```
In [77] nba.iloc[0:10:2]
```

```
Out [77]
```

Name	Team	Position	Birthday	Salary
Shake Milton	Philadelphia	76ers	SG 1996-09-26	1445697
PJ Washington	Charlotte	Hornets	PF 1998-08-23	3831840
Marial Shayok	Philadelphia	76ers	G 1995-07-26	79568
Kendrick Nunn	Miami	Heat	SG 1995-08-03	1416852
Brook Lopez	Milwaukee	Bucks	C 1988-04-01	12093024

Такая методика срезов особенно удобна, когда нужно извлечь каждую вторую строку.

4.7.3. Извлечение значений из конкретных столбцов

Как у атрибута `loc`, так и у `iloc` есть второй параметр, задающий извлекаемый столбец (столбцы). При использовании `loc` необходимо указать название столбца. При использовании же `iloc` — позицию столбца. В следующем примере извлечем с помощью `loc` значение, расположенное на пересечении строки "Giannis Antetokounmpo" и столбца `Team`:

```
In [78] nba.loc["Giannis Antetokounmpo", "Team"]
```

```
Out [78] 'Milwaukee Bucks'
```

При необходимости можно задать несколько значений, передав список в качестве одного или обоих аргументов метода-получателя `loc`. Код ниже извлекает значения из столбцов `Position` и `Birthday` в строке с меткой индекса "James Harden" и возвращает объект `Series`:

```
In [79] nba.loc["James Harden", ["Position", "Birthday"]]
```

```
Out [79] Position          PG
        Birthday    1989-08-26 00:00:00
        Name: James Harden, dtype: object
```

А так передаются несколько меток строк и столбцов:

```
In [80] nba.loc[
        ["Russell Westbrook", "Anthony Davis"],
        ["Team", "Salary"]
    ]
```

```
Out [80]
```

	Team	Salary
Name		

Russell Westbrook	Houston Rockets	38506482
Anthony Davis	Los Angeles Lakers	27093019

Имеется возможность извлекать несколько столбцов посредством синтаксиса срезов без явного указания их названий. В нашем наборе данных четыре столбца (`Team`, `Position`, `Birthday` и `Salary`). Извлечем все столбцы от `Position` до `Salary`. `Pandas` включает обе конечные точки в срез `loc`:

```
In [81] nba.loc["Joel Embiid", "Position":"Salary"]
```

```
Out [81] Position          C
        Birthday    1994-03-16 00:00:00
        Salary          27504630
        Name: Joel Embiid, dtype: object
```

Указывать названия столбцов необходимо в порядке их появления в объекте `DataFrame`. Результат следующего примера пуст, поскольку в исходном наборе столбец `Salary` идет за столбцом `Position`, то есть в команде задан ошибочный порядок. Обработывая синтаксис команды, pandas не может определить, какие столбцы нужно извлечь:

```
In [82] nba.loc["Joel Embiid", "Salary":"Position"]
```

```
Out [82] Series([], Name: Joel Embiid, dtype: object)
```

Предположим, нам нужно указать столбцы по порядку, а не по названиям. Напомню, что в библиотеке pandas каждому столбцу объекта `DataFrame` соответствует позиция индекса. В `nba` индекс столбца `Team` равен 0, столбца `Position` — 1 и т. д. Индекс столбца можно передать в качестве второго аргумента метода `iloc`. Например, извлечем значение, расположенное на пересечении строки с индексом 57 и столбца с индексом 3 (`Salary`):

```
In [83] nba.iloc[57, 3]
```

```
Out [83] 796806
```

Pandas поддерживает также использование и синтаксиса срезов списков. В следующем примере извлекаются все столбцы от начала до столбца (не включая его) с индексом 3 (`Salary`) в строках, начиная от позиции индекса 100 до (не включая его) позиции индекса 104:

```
In [84] nba.iloc[100:104, :3]
```

```
Out [84]
```

Name	Team	Position	Birthday
Brian Bowen	Indiana Pacers	SG	1998-10-02
Aaron Holiday	Indiana Pacers	PG	1996-09-30
Troy Daniels	Los Angeles Lakers	SG	1991-07-15
Buddy Hield	Sacramento Kings	SG	1992-12-17

Методы-получатели `iloc` и `loc` исключительно гибки. Им можно передавать в квадратных скобках отдельное значение, список значений, срез списка и многое другое. Отрицательная сторона подобной гибкости — дополнительные накладные расходы: pandas необходимо определять, какой именно тип входных данных получает `iloc` или `loc`.

Если нужно извлечь из объекта `DataFrame` отдельное значение, можно воспользоваться двумя альтернативными атрибутами, `at` и `iat`. Они работают быстрее,

поскольку библиотека pandas оптимизирует алгоритм для поиска отдельного значения.

Синтаксис этих атрибутов аналогичен. Атрибут `at` принимает метки строки и столбца:

```
In [85] nba.at["Austin Rivers", "Birthday"]
```

```
Out [85] Timestamp('1992-08-01 00:00:00')
```

Атрибут `iat` принимает индексы строки и столбца:

```
In [86] nba.iat[263, 1]
```

```
Out [86] 'PF'
```

Блокноты Jupyter включают несколько «магических» методов для облегчения труда разработчика. «Магические» методы объявляются с префиксом `%` и вводятся вместе с обычным кодом Python. Один из примеров — «магический» метод `%timeit`, выполняющий код в ячейке и вычисляющий среднее время его выполнения. `%timeit` порой выполняет ячейку до 100 000 раз! В следующем примере сравним с помощью «магического» метода быстродействие встречавшихся нам до сих пор методов-получателей:

```
In [87] %timeit
        nba.at["Austin Rivers", "Birthday"]
```

```
6.38 µs ± 53.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [88] %timeit
        nba.loc["Austin Rivers", "Birthday"]
```

```
9.12 µs ± 53.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [89] %timeit
        nba.iat[263, 1]
```

```
4.7 µs ± 27.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [90] %timeit
        nba.iloc[263, 1]
```

```
7.41 µs ± 39.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Результаты на разных компьютерах могут слегка различаться, но они демонстрируют очевидное более высокое быстродействие методов `at` и `iat` по сравнению с `loc` и `iloc`.

4.8. ИЗВЛЕЧЕНИЕ ЗНАЧЕНИЙ ИЗ ОБЪЕКТОВ SERIES

Методы-получатели `loc`, `iloc`, `at` и `iat` есть и в классе `Series`. Попробуем их в действии на объекте `Series`, извлеченном из нашего `DataFrame`, например `Salary`:

```
In [91] nba["Salary"].loc["Damian Lillard"]
```

```
Out [91] 29802321
```

```
In [92] nba["Salary"].at["Damian Lillard"]
```

```
Out [92] 29802321
```

```
In [93] nba["Salary"].iloc[234]
```

```
Out [93] 2033160
```

```
In [94] nba["Salary"].iat[234]
```

```
Out [94] 2033160
```

Теперь можете смело использовать те методы-получатели, которые вам покажутся более удобными.

4.9. ПЕРЕИМЕНОВАНИЕ СТОЛБЦОВ И СТРОК

Помните атрибут `columns`? Через него можно обращаться к объекту `Index`, в котором хранятся названия столбцов объекта `DataFrame`:

```
In [95] nba.columns
```

```
Out [95] Index(['Team', 'Position', 'Birthday', 'Salary'], dtype='object')
```

Можем переименовать один или несколько столбцов объекта `DataFrame`, присвоив этому атрибуту список новых названий. В следующем примере поменяем название столбца `Salary` на `Pay`:

```
In [96] nba.columns = ["Team", "Position", "Date of Birth", "Pay"]
        nba.head(1)
```

```
Out [96]
```

Name	Team	Position	Date of Birth	Pay
Shake Milton	Philadelphia	76ers	SG	1996-09-26 1445697

То же самое делает метод `rename`. Его параметру `columns` можно передать ассоциативный массив, ключами которого служат уже существующие названия столбцов, а значениями — их новые названия. Итак, приведем пример: поменяем название столбца


```
In [97] nba.rename(columns = { "Date of Birth": "Birthday" })
```

```
Out [97]
```

Name	Team	Position	Birthday	Pay
Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568
...
Austin Rivers	Houston Rockets	PG	1992-08-01	2174310
Harry Giles	Sacramento Kings	PF	1998-04-22	2578800
Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000

```
450 rows x 4 columns
```

Зафиксируем результат этой операции, присвоив возвращенный объект `DataFrame` переменной `nba`:

```
In [98] nba = nba.rename(columns = { "Date of Birth": "Birthday" })
```

Есть возможность переименовать и метки индекса, передав соответствующий ассоциативный массив в параметр `index` этого метода. Логика работы та же: ключи — старые метки, а значения — новые. В следующем примере имя "Giannis Antetokounmpo" меняется на известное прозвище этого игрока "Greek Freak":

```
In [99] nba.loc["Giannis Antetokounmpo"]
```

```
Out [99] Team          Milwaukee Bucks
         Position      PF
         Birthday      1994-12-06 00:00:00
         Pay           25842697
         Name: Giannis Antetokounmpo, dtype: object
```

```
In [100] nba = nba.rename(
            index = { "Giannis Antetokounmpo": "Greek Freak" }
        )
```

Найдем соответствующую строку по ее новой метке:

```
In [101] nba.loc["Greek Freak"]
```

```
Out [101] Team          Milwaukee Bucks
         Position      PF
         Birthday      1994-12-06 00:00:00
         Pay           25842697
         Name: Greek Freak, dtype: object
```

О чудо! Замена метки строки прошла удачно!

4.10. ЗАМЕНА ИНДЕКСА

Иногда необходимо изменить индекс объекта `DataFrame`, передав полномочия другому столбцу. Допустим, нам нужно сделать столбец `Team` индексом объекта `nba`. Можно вызвать метод `set_index`, с которым вы познакомились ранее в этой главе, передав ему название нужного столбца, но при этом мы потеряем текущий индекс, состоящий из имен игроков. Взгляните на следующий пример:

```
In [102] nba.set_index("Team").head()
```

```
Out [102]
```

Team	Position	Birthday	Salary
Philadelphia 76ers	SG	1996-09-26	1445697
Detroit Pistons	PF	1995-09-27	1645357
Charlotte Hornets	PF	1998-08-23	3831840
Detroit Pistons	PG	1988-10-04	7317074
Philadelphia 76ers	G	1995-07-26	79568

Чтобы сохранить имена игроков, необходимо сначала превратить уже существующий индекс в обычный столбец объекта `DataFrame`. Метод `reset_index` сначала делает из текущего индекса столбец `DataFrame` и лишь затем заменяет предыдущий индекс на числовой индекс pandas:

```
In [103] nba.reset_index().head()
```

```
Out [103]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
2	PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
3	Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
4	Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568

Теперь можно сделать из столбца `Team` индекс с помощью метода `set_index`, не рискуя потерять данные:

```
In [104] nba.reset_index().set_index("Team").head()
```

```
Out [104]
```

	Name	Position	Birthday	Salary
Team				
Philadelphia 76ers	Shake Milton	SG	1996-09-26	1445697
Detroit Pistons	Christian Wood	PF	1995-09-27	1645357
Charlotte Hornets	PJ Washington	PF	1998-08-23	3831840
Detroit Pistons	Derrick Rose	PG	1988-10-04	7317074
Philadelphia 76ers	Marial Shayok	G	1995-07-26	79568

Одно из преимуществ отказа от использования параметра `inplace` — возможность вне его связать цепочкой несколько вызовов методов. Свяжем цепочкой вызовы методов `reset_index` и `set_index` и запишем результат снова в переменную `nba`:

```
In [105] nba = nba.reset_index().set_index("Team")
```

Вот и все, ваше знакомство с объектом `DataFrame`, основной «рабочей лошадкой» библиотеки `pandas`, состоялось.

4.11. УПРАЖНЕНИЯ

Теперь, когда мы изучили финансовые показатели НБА, давайте применим концепции главы к другой спортивной лиге.

4.11.1. Задачи

Файл `nfl.csv` содержит список игроков Национальной футбольной лиги с уже знакомыми нам столбцами `Name`, `Team`, `Position`, `Birthday` и `Salary`. Попробуйте ответить на следующие вопросы.

1. Как импортировать файл `nfl.csv`? Как эффективнее всего преобразовать значения из столбца `Birthday` в метки даты/времени?
2. Какими двумя способами можно задать индекс объекта `DataFrame` для хранения имен игроков?
3. Как подсчитать количество игроков по командам в этом наборе данных?
4. Какие пять игроков наиболее высокооплачиваемые?
5. Как отсортировать этот набор данных сначала по командам в алфавитном порядке, а затем по зарплатам в порядке убывания?
6. Кто самый возрастной игрок в списке команды «Нью-Йорк Джетс» и когда он родился?

4.11.2. Решения

Решим эти задачи шаг за шагом.

1. Импортировать CSV-файл можно с помощью функции `read_csv`. Чтобы сохранить значения столбца `Birthday` в виде меток даты/времени, передадим этот столбец в параметр `parse_dates` в виде списка:

```
In [106] nfl = pd.read_csv("nfl.csv", parse_dates = ["Birthday"])
          nfl
```

```
Out [106]
```

	Name	Team	Position	Birthday	Salary
0	Tremon Smith	Philadelphia Eagles	RB	1996-07-20	570000
1	Shawn Williams	Cincinnati Bengals	SS	1991-05-13	3500000
2	Adam Butler	New England Patriots	DT	1994-04-12	645000
3	Derek Wolfe	Denver Broncos	DE	1990-02-24	8000000
4	Jake Ryan	Jacksonville Jaguars	OLB	1992-02-27	1000000
...
1650	Bashaud Breeland	Kansas City Chiefs	CB	1992-01-30	805000
1651	Craig James	Philadelphia Eagles	CB	1996-04-29	570000
1652	Jonotthan Harrison	New York Jets	C	1991-08-25	1500000
1653	Chuma Edogu	New York Jets	OT	1997-05-25	495000
1654	Tajae Sharpe	Tennessee Titans	WR	1994-12-23	2025000

1655 rows x 5 columns

- Наша следующая задача — сделать имена игроков метками индекса. Один из вариантов — вызвать метод `set_index` и присвоить новый объект `DataFrame` переменной `nfl`:

```
In [107] nfl = nfl.set_index("Name")
```

Другой вариант — передать параметр `index_col` функции `read_csv` при импорте набора данных:

```
In [108] nfl = pd.read_csv(
    "nfl.csv", index_col = "Name", parse_dates = ["Birthday"]
)
```

Результаты для обоих случаев будут одинаковы:

```
In [109] nfl.head()
```

```
Out [109]
```

Name	Team	Position	Birthday	Salary
Tremon Smith	Philadelphia Eagles	RB	1996-07-20	570000
Shawn Williams	Cincinnati Bengals	SS	1991-05-13	3500000
Adam Butler	New England Patriots	DT	1994-04-12	645000
Derek Wolfe	Denver Broncos	DE	1990-02-24	8000000
Jake Ryan	Jacksonville Jaguars	OLB	1992-02-27	1000000

- Для подсчета количества игроков в каждой команде можно вызвать метод `value_counts` для столбца `Team`. Но прежде необходимо извлечь объект `Series` для столбца `Team` посредством синтаксиса с использованием точки или квадратных скобок:

```
In [110] # Две строки ниже эквивалентны
nfl.Team.value_counts().head()
nfl["Team"].value_counts().head()
```

```
Out [110] New York Jets          58
          Washington Redskins    56
          Kansas City Chiefs     56
          San Francisco 49Ers    55
          New Orleans Saints     55
```

4. Чтобы найти пять наиболее высокооплачиваемых игроков, можно отсортировать столбец `Salary` с помощью метода `sort_values`. А чтобы pandas сортировала его именно в порядке убывания, необходимо передать параметру `ascending` аргумент `False`. Или можно воспользоваться методом `nlargest`:

```
In [111] nfl.sort_values("Salary", ascending = False).head()
```

```
Out [111]
Name                                     Team Position  Birthday  Salary
-----
Kirk Cousins      Minnesota Vikings      QB 1988-08-19  27500000
Jameis Winston    Tampa Bay Buccaneers      QB 1994-01-06  20922000
Marcus Mariota     Tennessee Titans      QB 1993-10-30  20922000
Derek Carr         Oakland Raiders      QB 1991-03-28  19900000
Jimmy Garoppolo    San Francisco 49Ers      QB 1991-11-02  17200000
```

5. Для сортировки по нескольким столбцам необходимо передать аргументы обоим параметрам `by` и `ascending` метода `sort_values`. Следующий код сортирует столбец `Team` в порядке возрастания, а затем внутри каждой команды (`Team`) столбец `Salary` в порядке убывания:

```
In [112] nfl.sort_values(
          by = ["Team", "Salary"],
          ascending = [True, False]
        )
```

```
Out [112]
Name                                     Team Position  Birthday  Salary
-----
Chandler Jones     Arizona Cardinals      OLB 1990-02-27  16500000
Patrick Peterson   Arizona Cardinals      CB 1990-07-11  11000000
Larry Fitzgerald   Arizona Cardinals      WR 1983-08-31  11000000
David Johnson      Arizona Cardinals      RB 1991-12-16   5700000
Justin Pugh        Arizona Cardinals      G 1990-08-15   5000000
...
Ross Pierschbacher Washington Redskins      C 1995-05-05   495000
Kelvin Harmon      Washington Redskins      WR 1996-12-15   495000
Wes Martin         Washington Redskins      G 1996-05-09   495000
Jimmy Moreland     Washington Redskins      CB 1995-08-26   495000
Jeremy Reaves      Washington Redskins      SS 1996-08-29   495000
```

```
1655 rows x 4 columns
```

6. Последняя задача, предупреждаю, с подвохом: необходимо найти самого возрастного игрока в списке команды «Нью-Йорк Джетс». Из арсенала

инструментов, доступных нам на этой стадии изучения pandas, воспользуемся изменением индекса: сделаем столбец `Team` индексом объекта `DataFrame`, чтобы упростить последующее извлечение всех игроков команды «Джетс». А чтобы сохранить имена игроков, которые сейчас играют роль индекса, предварительно воспользуемся методом `reset_index` для переноса их обратно в объект `DataFrame` в качестве обычного столбца:

```
In [113] nfl = nfl.reset_index().set_index(keys = "Team")
          nfl.head(3)
```

Out [113]

Name	Name	Position	Birthday	Salary
Philadelphia Eagles	Tremont Smith	RB	1996-07-20	570000
Cincinnati Bengals	Shawn Williams	SS	1991-05-13	3500000
New England Patriots	Adam Butler	DT	1994-04-12	645000

Теперь можно воспользоваться атрибутом `loc` для выделения всех игроков команды «Нью-Йорк Джетс»:

```
In [114] nfl.loc["New York Jets"].head()
```

Out [114]

Team	Name	Position	Birthday	Salary
New York Jets	Bronson Kaufusi	DE	1991-07-06	645000
New York Jets	Darryl Roberts	CB	1990-11-26	1000000
New York Jets	Jordan Willis	DE	1995-05-02	754750
New York Jets	Quinnen Williams	DE	1997-12-21	495000
New York Jets	Sam Ficken	K	1992-12-14	495000

Последний шаг — сортировка столбца `Birthday` и извлечение первой записи. Отсортировать его можно только потому, что мы заблаговременно преобразовали его значения в метки даты/времени:

```
In [115] nfl.loc["New York Jets"].sort_values("Birthday").head(1)
```

Out [115]

Team	Name	Position	Birthday	Salary
New York Jets	Ryan Kalil	C	1985-03-29	2400000

Самым возрастным игроком «Нью-Йорк Джетс» в этом наборе данных оказался Райан Кэлил. Он родился 29 марта 1985 года.

Поздравляю с завершением выполнения упражнений!

РЕЗЮМЕ

- **DataFrame** — двумерная структура данных, состоящая из строк и столбцов.
- Некоторые атрибуты и методы объектов **DataFrame** совпадают с атрибутами и методами объектов **Series**. Но многие их атрибуты и методы работают по-разному вследствие различной размерности этих объектов.
- Метод **sort_values** сортирует один или несколько столбцов объекта **DataFrame**. При этом можно задать различный порядок сортировки (в порядке убывания или возрастания) для различных столбцов.
- Атрибут **loc** позволяет извлекать строки или столбцы по меткам индекса. Атрибут **at** — удобный его вариант для извлечения лишь одного значения.
- Атрибут **iloc** позволяет извлекать строки или столбцы по позициям индекса. Атрибут **iat** — удобный его вариант для извлечения лишь одного значения.
- Метод **reset_index** преобразует индекс обратно в обычный столбец **DataFrame**.
- Метод **rename** позволяет задавать другие названия для одного (-ой) или нескольких столбцов или строк.

5

Фильтрация объектов *DataFrame*

В этой главе

- ✓ Уменьшение памяти, используемой объектом `DataFrame`.
- ✓ Извлечение строк объекта `DataFrame` по одному или нескольким условиям.
- ✓ Фильтрация строк объекта `DataFrame`, включающих или не включающих пустые значения.
- ✓ Выбор значений столбцов, попадающих в определенный диапазон.
- ✓ Удаление дубликатов и пустых значений из объекта `DataFrame`.

В главе 4 вы научились извлекать строки, столбцы и значения ячеек из объектов `DataFrame` с помощью методов-получателей `loc` и `iloc`. Эти методы-получатели хорошо подходят, если мы знаем точно, какие нам нужны метки и позиции индекса строк/столбцов. Но иногда бывает нужно выбрать строки не по идентификатору, а по какому-либо условию или критерию. Например, извлечь подмножество строк, в котором столбец содержит конкретное значение.

В этой главе вы научитесь описывать логические условия для включения/исключения строк из объекта `DataFrame`, а также сочетать различные условия с помощью операторов логического И и ИЛИ. Наконец, вы познакомитесь со вспомогательными методами библиотеки `pandas`, упрощающими процесс фильтрации. Впереди много интересного, так что по машинам!

5.1. ОПТИМИЗАЦИЯ ПАМЯТИ, ИСПОЛЬЗУЕМОЙ НАБОРОМ ДАННЫХ

Прежде чем перейти к фильтрации, вкратце обсудим вопрос сокращения объема используемой оперативной памяти в pandas. При импорте набора данных важно обеспечить хранение данных во всех столбцах в наиболее оптимальных типах. «Наилучший» тип данных — тип, потребляющий меньше всего памяти или обеспечивающий наибольшие возможности на практике. Например, целочисленные значения на большинстве компьютеров занимают меньше памяти, чем значения с плавающей точкой, так что, если набор данных содержит только целые числа, лучше импортировать их в виде целочисленных значений, а не значений с плавающей точкой. Еще один пример: если данные содержат даты, лучше всего импортировать их в виде меток даты/времени, а не строковых значений, чтобы можно было производить над ними соответствующие операции. В этом разделе вас ждет рассмотрение некоторых приемов, позволяющих сократить объем потребляемой памяти за счет преобразования данных столбцов в различные типы, а соответственно, и ускорить их фильтрацию в дальнейшем. Начнем с привычного импорта нашей любимой библиотеки анализа данных:

```
In [1] import pandas as pd
```

Набор данных для этой главы `employees.csv` представляет собой вымышленный список работников компании. Каждая запись включает имя сотрудника, его пол, дату начала работы в компании, зарплату; сведения о том, занимает ли он руководящую позицию (`True` или `False`), и подразделение, в котором он работает. Взглянем на этот набор данных с помощью функции `read_csv`:

```
In [2] pd.read_csv("employees.csv")
```

```
Out [2]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	8/6/93	NaN	True	Marketing
1	Thomas	Male	3/31/96	61933.0	True	NaN
2	Maria	Female	NaN	130590.0	False	Finance
3	Jerry	NaN	3/4/05	138705.0	True	Finance
4	Larry	Male	1/24/98	101004.0	True	IT
...
996	Phillip	Male	1/31/84	42392.0	False	Finance
997	Russell	Male	5/20/13	96914.0	False	Product
998	Larry	Male	4/20/13	60500.0	False	Business Dev
999	Albert	Male	5/15/12	129949.0	True	Sales
1000	NaN	NaN	NaN	NaN	NaN	NaN

```
1001 rows x 6 columns
```

Обратите внимание на разбросанные по этим результатам значения `NaN`. В каждом столбце есть отсутствующие значения. На самом деле последняя строка вообще состоит из одних `NaN`. Наборы данных могут попадать к нам с незаполненными строками, столбцами и т. д.

Как же повысить возможности и эффективность применения нашего набора данных? Первая оптимизация напрашивается прямо сейчас: преобразовать текстовые значения в столбце `Start Date` в метки даты/времени с помощью параметра `parse_dates`:

```
In [3] pd.read_csv("employees.csv", parse_dates = ["Start Date"]).head()
```

```
Out [3]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
3	Jerry	NaN	2005-03-04	138705.0	True	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT

Мы осуществили импорт CSV-файла успешно, с учетом особенностей типов данных, так что присвоим объект `DataFrame` переменной с информативным названием, например `employees`:

```
In [4] employees = pd.read_csv(
        "employees.csv", parse_dates = ["Start Date"]
    )
```

Существует несколько возможностей ускорения операций над нашим объектом `DataFrame` и повышения их эффективности. Во-первых, выведем общую информацию о нашем наборе данных в его текущем виде. Для просмотра списка столбцов, их типов данных, количества отсутствующих значений и общего объема занимаемой объектом `DataFrame` оперативной памяти можно вызвать метод `info`:

```
In [5] employees.info()
```

```
Out [5]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001 entries, 0 to 1000
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  -
0   First Name  933 non-null   object
1   Gender      854 non-null   object
2   Start Date  999 non-null   datetime64[ns]
3   Salary      999 non-null   float64
```

```

4  Mgmt          933 non-null    object
5  Team          957 non-null    object
dtypes: datetime64[ns](1), float64(1), object(4)
message usage: 47.0+ KB

```

Теперь пройдем по этим результатам сверху вниз. Наш объект `DataFrame` состоит из 1001 строки, начиная с индекса 0 и до индекса 1000. В нем четыре строковых столбца, один столбец с датами и один столбец со значениями с плавающей точкой. В каждом из этих шести столбцов есть пропущенные данные.

Текущий объем занимаемой памяти — 47 Кбайт — невелик для мощностей современных компьютеров, но давайте все-таки попробуем немного его сократить. При чтении следующих примеров обращайтесь основное внимание на сокращение расхода памяти, выраженное в процентах по отношению к изначальному, а не на конкретные числовые показатели сокращения. Чем больше наборы данных, тем более значительным окажется фактическое сокращение в физических единицах и отсюда — вырастет производительность.

5.1.1. Преобразование типов данных с помощью метода `astype`

Обратили ли вы внимание, что `pandas` импортировала значения столбца `Mgmt` в строковом виде? А он между тем содержит только два значения: `True` и `False`. Можно сократить объем занимаемой памяти за счет преобразования значений в более экономный булев тип данных.

Метод `astype` преобразует значения объекта `Series` в другой тип данных. Он принимает один аргумент — новый тип данных, причем передать ему можно как тип данных, так и строковое значение с его названием.

В следующем примере мы извлекаем объект `Series Mgmt` из `employees` и вызываем его метод `astype` с аргументом `bool`. `Pandas` возвращает новый объект `Series`, содержащий булевы значения. Обратите внимание, что `pandas` преобразует `NaN` в значения `True`. Мы обсудим вопрос удаления отсутствующих значений в подразделе 5.5.4.

```
In [6] employees["Mgmt"].astype(bool)
```

```

Out [6] 0      True
        1      True
        2     False
        3      True
        4      True
        ...
       996    False

```

```

997      False
998      False
999       True
1000      True
Name: Mgmt, Length: 1001, dtype: bool

```

Отлично! Мы оценили предварительно, как будет выглядеть наш объект `Series`, и теперь можем перезаписать столбец `Mgmt` в `employees`. Обновление столбца объекта `DataFrame` происходит аналогично заданию пары «ключ/значение» в ассоциативном массиве. Если столбец с указанным названием найден, библиотека `pandas` записывает на его место новый объект `Series`. Если же такого столбца не существует, `pandas` создает новый объект `Series` и присоединяет его справа к объекту `DataFrame`. `Pandas` сопоставляет строки в объектах `Series` и `DataFrame` по соответствию меток индекса.

В следующем примере кода перезаписывается столбец `Mgmt` новым объектом `Series` с булевыми значениями. Напомню, что язык `Python` вычисляет сначала правую сторону оператора присваивания (`=`). То есть сначала создается новый временный объект `Series`, а затем перезаписывается существующий столбец `Mgmt`:

```
In [7] employees["Mgmt"] = employees["Mgmt"].astype(bool)
```

Операция присваивания столбца не возвращает никакого значения, так что этот код не выводит ничего в блокноте `Jupyter`. А вот в объекте `DataFrame` произошли изменения. Давайте взглянем на него снова, чтобы увидеть, что получилось:

```
In [8] employees.tail()
```

```
Out [8]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales
1000	NaN	NaN	NaT	NaN	True	NaN

За исключением `True` в последней строке из пропущенных значений, объект `DataFrame` выглядит точно так же. А как насчет потребления памяти? Давайте снова вызовем метод `info`, чтобы увидеть разницу:

```
In [9] employees.info()
```

```
Out [9]
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001 entries, 0 to 1000
Data columns (total 6 columns):

```

```

#   Column      Non-Null Count  Dtype
---  -
0   First Name  933 non-null      object
1   Gender      854 non-null      object
2   Start Date  999 non-null      datetime64[ns]
3   Salary      999 non-null      float64
4   Mgmt        1001 non-null     bool
5   Team        957 non-null      object
dtypes: bool(1), datetime64[ns](1), float64(1), object(3)
memory usage: 40.2+ KB

```

Мы сократили объем занимаемой памяти почти на 15 %, с 47 до 40,2 Кбайт. Для начала весьма неплохо!

Перейдем теперь к столбцу Salary. Если открыть исходный CSV-файл, можно увидеть, что его значения представляют собой целые числа:

```

First Name,Gender,Start Date,Salary,Mgmt,Team
Douglas,Male,8/6/93,,True,Marketing
Thomas,Male,3/31/96,61933,True,
Maria,Female,,130590,False,Finance
Jerry,,3/4/05,138705,True,Finance

```

В `employees`, однако, `pandas` хранит значения столбца Salary в виде чисел с плавающей точкой. Для поддержки содержащихся в этом столбце NaN `pandas` преобразует целочисленные значения в числа с плавающей точкой. Это техническое требование библиотеки, с которым мы уже сталкивались в предыдущих главах.

Следуя стратегии нашего предыдущего примера с булевыми значениями, мы могли бы попытаться преобразовать значения этого столбца в целочисленные с помощью метода `astype`. К сожалению, `pandas` генерирует при этом исключение `ValueError`:

```

In [10] employees["Salary"].astype(int)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-99-b148c8b8be90> in <module>
----> 1 employees["Salary"].astype(int)

ValueError: Cannot convert non-finite values (NA or inf) to integer

```

Загвоздка в пустых значениях. Библиотека `pandas` не может преобразовать значения NaN в целочисленные. Можно решить эту проблему, заменив все значения NaN на какое-либо конкретное значение. Метод `fillna` как раз и заменяет пустые значения в объекте `Series` на переданный ему аргумент. В следующем примере используется значение-заполнитель 0. Обратите внимание, что иногда при таком подходе можно ненароком исказить данные; значение 0 выбрано исключительно ради примера.

Как мы знаем, в исходном столбце `Salary` было пропущено значение в последней строке. Давайте взглянем на эту строку после вызова метода `fillna`:

```
In [11] employees["Salary"].fillna(0).tail()
```

```
Out [11] 996      42392.0
          997      96914.0
          998      60500.0
          999     129949.0
         1000         0.0
          Name: Salary, dtype: float64
```

Прекрасно. Теперь, когда в столбце `Salary` больше нет пропущенных значений, можно преобразовать его значения в целочисленные с помощью метода `astype`:

```
In [12] employees["Salary"].fillna(0).astype(int).tail()
```

```
Out [12] 996      42392
          997      96914
          998      60500
          999     129949
         1000         0
          Name: Salary, dtype: int64
```

Следующим шагом можно перезаписать объект `Series Salary` в объекте `employees`:

```
In [13] employees["Salary"] = employees["Salary"].fillna(0).astype(int)
```

Возможно оптимизировать еще кое-что. В библиотеке pandas есть особый тип данных — *категория* (`category`). Он идеально подходит для столбцов, состоящих из небольшого числа уникальных значений (то есть набор значений невелик по сравнению с общим количеством записей в столбце). Несколько распространенных примеров точек данных с ограниченным количеством значений: пол, дни недели, группы крови, планеты и группы населения по уровню доходов. «За кулисами» pandas хранит только одну копию каждого категориального значения, а не все дублирующиеся в разных строках значения.

Подсчитать число уникальных значений в каждом столбце объекта `DataFrame` можно с помощью метода `nunique`. Отмечу, что по умолчанию он не считает отсутствующие значения (`NaN`):

```
In [14] employees.nunique()
```

```
Out [14] First Name      200
          Gender         2
          Start Date     971
          Salary        995
          Mgmt          2
          Team         10
          dtype: int64
```

Столбцы `Gender` и `Team`, похоже, неплохо подходят для хранения категориальных значений. На 1001 строку данных в столбце `Gender` только два уникальных значения, а в `Team` — только десять.

Воспользуемся снова методом `astype`. Во-первых, мы преобразуем значения столбца `Gender` в категории, передав в этот метод аргумент `"category"`:

```
In [15] employees["Gender"].astype("category")

Out [15] 0      Male
         1      Male
         2    Female
         3      NaN
         4      Male
         ...
        996    Male
        997    Male
        998    Male
        999    Male
       1000    NaN
        Name: Gender, Length: 1001, dtype: category
        Categories (2, object): [Female, Male]
```

Pandas нашла тут две уникальные категории: `"Female"` и `"Male"`. Можно пере-записывать столбец `Gender`:

```
In [16] employees["Gender"] = employees["Gender"].astype("category")
```

Проверим, как обстоят дела с объемом занимаемой памяти, с помощью метода `info`. И снова объем занимаемой памяти резко сократился, поскольку pandas приходится отслеживать лишь два значения вместо 1001:

```
In [17] employees.info()
```

```
Out [17]

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001 entries, 0 to 1000
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  -
0   First Name  933 non-null   object
1   Gender      854 non-null   category
2   Start Date  999 non-null   datetime64[ns]
3   Salary      1001 non-null  int64
4   Mgmt        1001 non-null  bool
5   Team        957 non-null   object
dtypes: bool(1), category(1), datetime64[ns](1), int64(1), object(2)
memory usage: 33.5+ KB
```

Повторим этот процесс для столбца `Team`, в котором только десять уникальных значений:

```
In [18] employees["Team"] = employees["Team"].astype("category")
```

```
In [19] employees.info()
```

```
Out [19]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001 entries, 0 to 1000
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   First Name  933 non-null   object
1   Gender      854 non-null   category
2   Start Date  999 non-null   datetime64[ns]
3   Salary      1001 non-null  int64
4   Mgmt        1001 non-null  bool
5   Team        957 non-null   category
dtypes: bool(1), category(2)
memory usage: 27.0+ KB
```

С помощью менее чем десятка строк кода мы сократили объем занимаемой объектом `DataFrame` памяти более чем на 40 %. Представьте себе эффект для наборов данных, содержащих миллионы строк!

5.2. ФИЛЬТРАЦИЯ ПО ОДНОМУ УСЛОВИЮ

Извлечение подмножества — вероятно, самая распространенная операция в анализе данных. *Подмножество* (subset) — это часть более крупного набора данных, удовлетворяющих какому-либо условию.

Пусть нам нужно сгенерировать список всех сотрудников по имени `Maria`. Для этого необходимо отфильтровать набор данных о сотрудниках по значениям из столбца `First Name`. Список сотрудников по имени `Maria` — подмножество набора всех сотрудников.

Во-первых, напомним коротко, как работает операция сравнения на равенство в языке Python. Оператор равенства (`==`) сравнивает два объекта Python на равенство, возвращая `True`, если они равны, и `False` в противном случае (см. подробные пояснения в приложении Б). Вот простой пример:

```
In [20] "Maria" == "Maria"
```

```
Out [20] True
```

```
In [21] "Maria" == "Taylor"
```

```
Out [21] False
```


Для сравнения всех записей объекта `Series` с константой необходимо указать объект `Series` с одной стороны оператора равенства, а константу — с другой:

```
Series == value
```

Может показаться, что такой синтаксис приведет к ошибке, но библиотека `pandas` достаточно «умна», чтобы понять: мы хотим сравнить с указанным строковым значением каждое из значений объекта `Series`, а не сам объект `Series`. Мы обсуждали подобные идеи в главе 2, когда рассматривали сочетание объектов `Series` с математическими операторами, например со знаком сложения.

При сочетании объекта `Series` с оператором равенства библиотека `pandas` возвращает объект `Series`, содержащий булевы значения. В следующем примере мы сравниваем каждое из значений столбца `First Name` с `"Maria"`. Значение `True` показывает, что по соответствующему индексу действительно располагается значение `"Maria"`, а `False` — что нет. Следующий результат демонстрирует, что на позиции с индексом 2 хранится значение `"Maria"`:

```
In [22] employees["First Name"] == "Maria"

Out [22] 0      False
         1      False
         2       True
         3      False
         4      False
         ...
        996      False
        997      False
        998      False
        999      False
       1000      False
         Name: First Name, Length: 1001, dtype: bool
```

Чтобы получить все записи `"Maria"` из нашего набора данных, необходимо извлечь лишь строки со значением `True` в вышеприведенных результатах. К счастью, библиотека `pandas` предоставляет удобный синтаксис для извлечения строк с помощью булевых объектов `Series`. Для фильтрации строк необходимо указать булев объект `Series` между квадратными скобками после объекта `DataFrame`:

```
In [23] employees[employees["First Name"] == "Maria"]

Out [23]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
2	Maria	Female	NaT	130590	False	Finance
198	Maria	Female	1990-12-27	36067	True	Product
815	Maria	NaN	1986-01-18	106562	False	HR

844	Maria	NaN	1985-06-19	148857	False	Legal
936	Maria	Female	2003-03-14	96250	False	Business Dev
984	Maria	Female	2011-10-15	43455	False	Engineering

Полный успех! Мы выбрали строки со значением "Maria" в столбце `First Name` с помощью булева объекта `Series`.

Если подобное количество вложенных квадратных скобок, как в кодовой строке выше, путает вас, можете присвоить булев объект `Series` переменной с информативным названием и указать ее в квадратных скобках. Следующий код дает в результате то же самое подмножество строк, что и предыдущий:

```
In [24] marias = employees["First Name"] == "Maria"
        employees[marias]
```

Out [24]

	First Name	Gender	Start Date	Salary	Mgmt	Team
2	Maria	Female	NaT	130590	False	Finance
198	Maria	Female	1990-12-27	36067	True	Product
815	Maria	NaN	1986-01-18	106562	False	HR
844	Maria	NaN	1985-06-19	148857	False	Legal
936	Maria	Female	2003-03-14	96250	False	Business Dev
984	Maria	Female	2011-10-15	43455	False	Engineering

Самая распространенная ошибка новичков при сравнении значений — использование одного знака равенства вместо двух. Помните, что один знак равенства служит для присваивания объекта переменной, а два — для проверки равенства объектов. Если бы мы случайно указали в этом примере только один знак равенства, то перезаписали бы на место всех значений столбца `First Name` значение "Maria". А это нам бы, конечно, не подошло.

Давайте взглянем еще на один пример. Пусть нам нужно извлечь подмножество сотрудников, не входящих в группу `Finance`. Алгоритм действий остается тем же, но с небольшим изменением. Необходимо сгенерировать булев объект `Series`, проверяющий, какие из значений столбца `Team` не равны "Finance". А затем можно будет отфильтровать `employees` на основе этого булева объекта `Series`. Оператор «не равно» языка Python возвращает `True`, если два значения не равны, и `False`, если равны:

```
In [25] "Finance" != "Engineering"
```

Out [25] True

Оператор «не равно» с тем же успехом работает с объектами `Series`, что и оператор равенства. В следующем примере значения из столбца `Team` сравниваются со строковым значением "Finance". `True` означает, что на соответствующей

позиции в столбце `Team` не содержится значение `"Finance"`, а `False` — что это значение равно `"Finance"`:

```
In [26] employees["Team"] != "Finance"

Out [26] 0      True
         1      True
         2     False
         3     False
         4      True
         ...
        996    False
        997     True
        998     True
        999     True
       1000     True
        Name: Team, Length: 1001, dtype: bool
```

Теперь, получив нужный булев объект `Series`, мы можем передать его между квадратными скобками, чтобы извлечь строки объекта `DataFrame` со значением `True`. Из следующих результатов видно, что библиотека `pandas` исключила строки с индексами 2 и 3, поскольку в столбце `Team` для них содержится значение `"Finance"`:

```
In [27] employees[employees["Team"] != "Finance"]
```

```
Out [27]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
1	Thomas	Male	1996-03-31	61933	True	NaN
4	Larry	Male	1998-01-24	101004	True	IT
5	Dennis	Male	1987-04-18	115163	False	Legal
6	Ruby	Female	1987-08-17	65476	True	Product
...
995	Henry	NaN	2014-11-23	132483	False	Distribution
997	Russell	Male	2013-05-20	96914	False	Product
998	Larry	Male	2013-04-20	60500	False	Business Dev
999	Albert	Male	2012-05-15	129949	True	Sales
1000	NaN	NaN	NaT	0	True	NaN

```
899 rows x 6 columns
```

Обратите внимание, что эти результаты включают строки с пропущенными значениями. Пример можно увидеть в строке с индексом 1000. В этом сценарии `pandas` считает, что `NaN` не равно строковому значению `"Finance"`.

А если мы хотим извлечь всех начальников в компании? Их можно определить по значению `True` в столбце `Mgmt`. Можно воспользоваться оператором

`employees["Mgmt"] == True`, но это не нужно, ведь `Mgmt` уже сам по себе представляет собой булев объект `Series`. Значения `True` и `False` в нем указывают, следует ли библиотеке `pandas` извлечь строку или отбросить ее. Следовательно, стоит просто указать сам столбец `Mgmt` внутри квадратных скобок:

```
In [28] employees[employees["Mgmt"]].head()
```

```
Out [28]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
1	Thomas	Male	1996-03-31	61933	True	NaN
3	Jerry	NaN	2005-03-04	138705	True	Finance
4	Larry	Male	1998-01-24	101004	True	IT
6	Ruby	Female	1987-08-17	65476	True	Product

Можно также использовать арифметические операторы для фильтрации столбцов на основе математических условий. В следующем примере мы генерируем булев объект `Series` для значений столбца `Salary`, превышающих 100 000 долларов (см. подробности синтаксиса в главе 2):

```
In [29] high_earners = employees["Salary"] > 100000
        high_earners.head()
```

```
Out [29] 0    False
         1    False
         2     True
         3     True
         4     True
        Name: Salary, dtype: bool
```

Взглянем теперь, какие сотрудники зарабатывают более 100 000 долларов:

```
In [30] employees[high_earners].head()
```

```
Out [30]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
2	Maria	Female	NaT	130590	False	Finance
3	Jerry	NaN	2005-03-04	138705	True	Finance
4	Larry	Male	1998-01-24	101004	True	IT
5	Dennis	Male	1987-04-18	115163	False	Legal
9	Frances	Female	2002-08-08	139852	True	Business Dev

Поэкспериментируйте с этим синтаксисом на каких-нибудь других столбцах объекта `employees`. Достаточно передать библиотеке `pandas` булев объект `Series`, и она отфильтрует `DataFrame`.

5.3. ФИЛЬТРАЦИЯ ПО НЕСКОЛЬКИМ УСЛОВИЯМ

Фильтровать объекты `DataFrame` можно и по нескольким условиям, это можно сделать посредством создания двух отдельных булевых объектов `Series` и объявления логического условия, которое `pandas` должна применить к ним.

5.3.1. Условие И

Пусть нам надо найти всех сотрудников женского пола, работающих в команде развития бизнеса. Теперь библиотеке `pandas` нужно выбирать строки по двум условиям: значению `"Female"` в столбце `Gender` и значению `"Business Dev"` в столбце `Team`. Эти критерии не зависят друг от друга, но искомые строки должны соответствовать обоим. Напомним вкратце логику работы операции И с двумя условиями (табл. 5.1).

Таблица 5.1

Условие 1	Условие 2	Результат
True	True	True
True	False	False
False	True	False
False	False	False

Создадим нужные объекты `Series` по очереди. Начнем с выделения значений `"Female"` в столбце `Gender`:

```
In [31] is_female = employees["Gender"] == "Female"
```

Затем выберем всех сотрудников группы `"Business Dev"`:

```
In [32] in_biz_dev = employees["Team"] == "Business Dev"
```

Наконец, необходимо вычислить пересечение этих двух объектов `Series`, выбрав строки, в которых значения и `is_female`, и `in_biz_dev` равны `True`. Передайте оба объекта `Series` в квадратных скобках, поставив между ними символ амперсанда (`&`). Амперсанд задает логический критерий И. То есть объект `Series is_female` должен равняться `True`, и одновременно объект `Series in_biz_dev` должен равняться `True`:

```
In [33] employees[is_female & in_biz_dev].head()
```

```
Out [33]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
9	Frances	Female	2002-08-08	139852	True	Business Dev
33	Jean	Female	1993-12-18	119082	False	Business Dev
36	Rachel	Female	2009-02-16	142032	False	Business Dev
38	Stephanie	Female	1986-09-13	36844	True	Business Dev
61	Denise	Female	2001-11-06	106862	False	Business Dev

Можно указать в квадратных скобках сколько угодно объектов `Series`, если любые два последовательных объекта разделены символом `&`. В следующем примере прибавляется третий критерий для выбора женщин-начальников в группе развития бизнеса:

```
In [34] is_manager = employees["Mgmt"]
        employees[is_female & in_biz_dev & is_manager].head()
```

Out [34]

	First Name	Gender	Start Date	Salary	Mgmt	Team
9	Frances	Female	2002-08-08	139852	True	Business Dev
38	Stephanie	Female	1986-09-13	36844	True	Business Dev
66	Nancy	Female	2012-12-15	125250	True	Business Dev
92	Linda	Female	2000-05-25	119009	True	Business Dev
111	Bonnie	Female	1999-12-17	42153	True	Business Dev

Подытожим: символ `&` служит для выбора строк, удовлетворяющих всем заданным условиям. Объясните два или более булевых объекта `Series` и свяжите их все амперсандами.

5.3.2. Условие ИЛИ

Можно также извлечь строки, удовлетворяющие хотя бы одному из нескольких условий. Скажем, не все условия должны выполняться, достаточно одного. Напомним вкратце логику работы операции ИЛИ с двумя условиями (табл. 5.2).

Таблица 5.2

Условие 1	Условие 2	Результат
True	True	True
True	False	True
False	True	True
False	False	False

Допустим, нам нужно найти всех сотрудников с зарплатой меньше 40 000 долларов или со значением `Start Date` после 1 января 2015 года. Получить два

отдельных булевых объекта `Series`, соответствующих этим условиям, можно с помощью математических операторов, в частности `<` и `>`:

```
In [35] earning_below_40k = employees["Salary"] < 40000
        started_after_2015 = employees["Start Date"] > "2015-01-01"
```

Критерий **ИЛИ** объявляется путем указания символа вертикальной черты (`|`) между булевыми объектами `Series`. В следующем примере выводятся строки, в которых в любом из двух булевых объектов `Series` содержится значение `True`:

```
In [36] employees[earning_below_40k | started_after_2015].tail()
```

```
Out [36]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
958	Gloria	Female	1987-10-24	39833	False	Engineering
964	Bruce	Male	1980-05-07	35802	True	Sales
967	Thomas	Male	2016-03-12	105681	False	Engineering
989	Justin	NaN	1991-02-10	38344	False	Legal
1000	NaN	NaN	NaT	0	True	NaN

Строки на позициях с индексами 958, 964, 989 и 1000 удовлетворяют условию по `Salary`, а строка на позиции с индексом 967 удовлетворяет условию по `Start Date`. Библиотека `pandas` включит в результат также строки, удовлетворяющие обоим этим условиям.

5.3.3. Логическое отрицание (~)

Символ тильды (`~`) обращает значения в булевом объекте `Series`. Все значения `True` превращаются в `False`, а все `False` становятся `True`. Вот простой пример с небольшим объектом `Series`:

```
In [37] my_series = pd.Series([True, False, True])
        my_series
```

```
Out [37] 0      True
         1     False
         2      True
        dtype: bool
```

```
In [38] ~my_series
```

```
Out [38] 0     False
         1      True
         2     False
        dtype: bool
```

Логическое отрицание удобно, когда нужно поменять условие на обратное. Допустим, нам надо найти сотрудников с зарплатой меньше 100 000 долларов. Возможны два подхода, первый из которых — написать `employees["Salary"] < 100000`:

```
In [39] employees[employees["Salary"] < 100000].head()
```

```
Out [39]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
1	Thomas	Male	1996-03-31	61933	True	NaN
6	Ruby	Female	1987-08-17	65476	True	Product
7	NaN	Female	2015-07-20	45906	True	Finance
8	Angela	Female	2005-11-22	95570	True	Engineering

Или можно найти отрицание набора сотрудников, зарабатывающих более (или ровно) 100 000 долларов. Полученные в результате этих подходов объекты `DataFrame` будут идентичны друг другу. В следующем примере мы помещаем операцию сравнения «больше» в круглые скобки. Такой синтаксис гарантирует генерацию библиотекой pandas булева объекта `Series` перед обращением его значений. И вообще, стоит принять на заметку: следует использовать круглые скобки во всех случаях, когда возможны сомнения в порядке выполнения операций:

```
In [40] employees[~(employees["Salary"] >= 100000)].head()
```

```
Out [40]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
1	Thomas	Male	1996-03-31	61933	True	NaN
6	Ruby	Female	1987-08-17	65476	True	Product
7	NaN	Female	2015-07-20	45906	True	Finance
8	Angela	Female	2005-11-22	95570	True	Engineering

СОВЕТ

В сложных случаях наподобие этого лучше присвоить булев объект `Series` переменной с информативным названием.

5.3.4. Методы для работы с булевыми значениями

Библиотека pandas предоставляет и другой синтаксис, предназначенный преимущественно для аналитиков, предпочитающих методы операциям. В табл. 5.3 приведены соответствия методов для равенства, неравенства и прочих арифметических операций:

Таблица 5.3

Операция	Арифметический синтаксис	Синтаксис методов
Равно	<code>employees["Team"] == "Marketing"</code>	<code>employees["Team"].eq("Marketing")</code>
Не равно	<code>employees["Team"] != "Marketing"</code>	<code>employees["Team"].ne("Marketing")</code>
Меньше чем	<code>employees["Salary"] < 100000</code>	<code>employees["Salary"].lt(100000)</code>
Меньше чем или равно	<code>employees["Salary"] <= 100000</code>	<code>employees["Salary"].le(100000)</code>
Больше чем	<code>employees["Salary"] > 100000</code>	<code>employees["Salary"].gt(100000)</code>
Больше чем или равно	<code>employees["Salary"] >= 100000</code>	<code>employees["Salary"].ge(100000)</code>

При этом применимы уже описанные правила относительно использования символов `&` и `|` для логических операций И/ИЛИ.

5.4. ФИЛЬТРАЦИЯ ПО УСЛОВИЮ

Существуют и более сложные операции, чем простая проверка на равенство или неравенство. Библиотека `pandas` включает множество удобных и разнообразных методов для генерации объектов `Series` в подобных сценариях выборки данных.

5.4.1. Метод `isin`

Что, если требуется выбрать сотрудников, относящихся к одной из групп `Sales`, `Legal` или `Marketing`? Можно указать три отдельных булевых объекта `Series` в квадратных скобках и добавить между ними символы `|` логической операции ИЛИ:

```
In [41] sales = employees["Team"] == "Sales"
        legal = employees["Team"] == "Legal"
        mktg = employees["Team"] == "Marketing"
        employees[sales | legal | mktg].head()
```

Out [41]

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
5	Dennis	Male	1987-04-18	115163	False	Legal
11	Julie	Female	1997-10-26	102508	True	Legal
13	Gary	Male	2008-01-27	109831	False	Sales
20	Lois	NaN	1995-04-22	64714	True	Legal

Такое решение работает, но плохо масштабируется. Ведь может случиться, что в следующем отчете понадобится вывести сотрудников, относящихся к 15 группам

вместо трех. Придется «перепахивать» и существенно расширять весь синтаксис? Объявлять объект `Series` для каждого из 15 условий — весьма утомительная задача.

Лучше будет воспользоваться методом `isin`, который принимает на входе итерируемый объект с элементами (список, кортеж, объект `Series` и т. д.) и возвращает булев объект `Series`. `True` в его записи означает, что pandas нашла значение из строки среди значений итерируемого объекта, а `False` — что нет. А затем можно отфильтровать объект `DataFrame` как обычно, на основе полученного объекта `Series`. В следующем примере мы получаем тот же результат, что и раньше:

```
In [42] all_star_teams = ["Sales", "Legal", "Marketing"]
        on_all_star_teams = employees["Team"].isin(all_star_teams)
        employees[on_all_star_teams].head()
```

Out [42]

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
5	Dennis	Male	1987-04-18	115163	False	Legal
11	Julie	Female	1997-10-26	102508	True	Legal
13	Gary	Male	2008-01-27	109831	False	Sales
20	Lois	NaN	1995-04-22	64714	True	Legal

Оптимальный сценарий использования метода `isin` — когда набор сравниваемых данных заранее не известен и, например, генерируется динамически.

5.4.2. Метод `between`

При работе с числами или датами часто приходится извлекать значения, входящие в определенный диапазон. Пусть нам нужно найти всех сотрудников с зарплатой от 80 000 до 90 000 долларов. Можно создать два объекта `Series`: один для нижней границы диапазона, а второй — для верхней. А затем воспользоваться оператором `&`, чтобы обеспечить истинность обоих условий:

```
In [43] higher_than_80 = employees["Salary"] >= 80000
        lower_than_90 = employees["Salary"] < 90000
        employees[higher_than_80 & lower_than_90].head()
```

Out [43]

	First Name	Gender	Start Date	Salary	Mgmt	Team
19	Donna	Female	2010-07-22	81014	False	Product
31	Joyce	NaN	2005-02-20	88657	False	Product
35	Theresa	Female	2006-10-10	85182	False	Sales
45	Roger	Male	1980-04-17	88010	True	Sales
54	Sara	Female	2007-08-15	83677	False	Engineering

Чуть более изящное решение — воспользоваться методом `between`, принимающим в качестве параметров нижнюю и верхнюю границы диапазона и возвращающим булев объект `Series`, в котором `True` означает, что значение входит в указанный диапазон. Обратите внимание, что первый аргумент, нижняя граница, включается в диапазон, а второй аргумент, верхняя граница, — не включается¹. Следующий код возвращает тот же объект `DataFrame`, что и предыдущий, выбирая зарплаты от 80 000 до 90 000 долларов:

```
In [44] between_80k_and_90k = employees["Salary"].between(80000, 90000)
        employees[between_80k_and_90k].head()
```

Out [44]

	First Name	Gender	Start Date	Salary	Mgmt	Team

19	Donna	Female	2010-07-22	81014	False	Product
31	Joyce	NaN	2005-02-20	88657	False	Product
35	Theresa	Female	2006-10-10	85182	False	Sales
45	Roger	Male	1980-04-17	88010	True	Sales
54	Sara	Female	2007-08-15	83677	False	Engineering

Метод `between` можно использовать и для столбцов других типов данных. Для фильтрации меток даты/времени можно передавать строковые значения в качестве начальной и конечной дат диапазона. Названия параметров для первого и второго аргументов метода — `left` и `right`. Код ниже выбирает всех сотрудников, работающих в компании с 1980-х:

```
In [45] eighties_folk = employees["Start Date"].between(
        left = "1980-01-01",
        right = "1990-01-01"
    )
    employees[eighties_folk].head()
```

Out [45]

	First Name	Gender	Start Date	Salary	Mgmt	Team

5	Dennis	Male	1987-04-18	115163	False	Legal
6	Ruby	Female	1987-08-17	65476	True	Product
10	Louise	Female	1980-08-12	63241	True	NaN
12	Brandon	Male	1980-12-01	112807	True	HR
17	Shawn	Male	1986-12-07	111737	False	Product

¹ Начиная с версии 1.3.0 библиотеки `pandas`, по умолчанию включаются обе границы. Но это поведение можно изменить, задав соответствующий аргумент для параметра `inclusive`. Возможные аргументы: `"both"` (включаются обе границы), `"neither"` (не включается ни одна), `"left"` (включается только левая граница), `"right"` (включается только правая граница). — *Примеч. пер.*

Метод `between` работает и со строковыми столбцами. Извлечем данные всех сотрудников, имя которых начинается с буквы `R`. Наш диапазон простирается от заглавной буквы `R` в качестве верхней границы (включительно) и до заглавной буквы `S` в качестве нижней границы (граница не включается):

```
In [46] name_starts_with_r = employees["First Name"].between("R", "S")
        employees[name_starts_with_r].head()
```

Out [46]

	First Name	Gender	Start Date	Salary	Mgmt	Team
6	Ruby	Female	1987-08-17	65476	True	Product
36	Rachel	Female	2009-02-16	142032	False	Business Dev
45	Roger	Male	1980-04-17	88010	True	Sales
67	Rachel	Female	1999-08-16	51178	True	Finance
78	Robin	Female	1983-06-04	114797	True	Sales

Как обычно, не забывайте о чувствительности к регистру при работе с символами и строковыми значениями.

5.4.3. Методы `isnull` и `notnull`

Набор данных по сотрудникам включает множество пропущенных значений. Некоторые из них видны уже в первых пяти строках вывода:

```
In [47] employees.head()
```

Out [47]

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	0	True	Marketing
1	Thomas	Male	1996-03-31	61933	True	NaN
2	Maria	Female	NaT	130590	False	Finance
3	Jerry	NaN	2005-03-04	138705	True	Finance
4	Larry	Male	1998-01-24	101004	True	IT

Библиотека `pandas` использует для отсутствующих текстовых и числовых значений обозначение `NaN` («нечисловое значение»), а для отсутствующих значений даты/времени — `NaT` («не значение времени»). Пример последнего можно видеть в столбце `Start Date` на позиции с индексом 2.

Для отбора строк с пустыми или непустыми значениями в заданном столбце можно использовать несколько методов библиотеки `pandas`. Метод `isnull` возвращает булев объект `Series`, в котором `True` означает, что значение для соответствующей строки отсутствует:

```
In [48] employees["Team"].isnull().head()
```

```
Out [48] 0    False
         1     True
         2    False
         3    False
         4    False
         Name: Team, dtype: bool
```

Библиотека pandas рассматривает как пустые также значения `NaT` и `None`. В следующем примере мы вызываем метод `isnull` для столбца `Start Date`:

```
In [49] employees["Start Date"].isnull().head()
```

```
Out [49] 0    False
         1    False
         2     True
         3    False
         4    False
         Name: Start Date, dtype: bool
```

Метод `notnull` возвращает обратный этому объект `Series`, в котором `True` означает наличие значения для соответствующей строки. Судя по следующим результатам, значения в позициях с индексами 0, 2, 3 и 4 не отсутствуют:

```
In [50] employees["Team"].notnull().head()
```

```
Out [50] 0     True
         1    False
         2     True
         3     True
         4     True
         Name: Team, dtype: bool
```

Получить тот же результат можно путем логического отрицания объекта `Series`, возвращаемого методом `isnull`. Напомню, что для логического отрицания булева объекта `Series` используется символ тильды (`~`):

```
In [51] (~employees["Team"].isnull()).head()
```

```
Out [51] 0     True
         1    False
         2     True
         3     True
         4     True
         Name: Team, dtype: bool
```

Оба эти подхода работают, но `notnull` выглядит информативнее, а потому лучше использовать именно его.

Как обычно, эти булевы объекты `Series` можно использовать для извлечения конкретных столбцов объекта `DataFrame`. Например, можно извлечь всех сотрудников, у которых отсутствует значение в столбце `Team`:

```
In [52] no_team = employees["Team"].isnull()
        employees[no_team].head()
```

Out [52]

	First Name	Gender	Start Date	Salary	Mgmt	Team
1	Thomas	Male	1996-03-31	61933	True	NaN
10	Louise	Female	1980-08-12	63241	True	NaN
23	NaN	Male	2012-06-14	125792	True	NaN
32	NaN	Male	1998-08-21	122340	True	NaN
91	James	NaN	2005-01-26	128771	False	NaN

В следующем примере мы извлекаем сотрудников, у которых указано имя (в столбце `First Name`):

```
In [53] has_name = employees["First Name"].notnull()
        employees[has_name].tail()
```

Out [53]

	First Name	Gender	Start Date	Salary	Mgmt	Team
995	Henry	NaN	2014-11-23	132483	False	Distribution
996	Phillip	Male	1984-01-31	42392	False	Finance
997	Russell	Male	2013-05-20	96914	False	Product
998	Larry	Male	2013-04-20	60500	False	Business Dev
999	Albert	Male	2012-05-15	129949	True	Sales

Использование методов `isnull` и `notnull` — оптимальный способ быстрой фильтрации на предмет присутствующих или отсутствующих значений в одной или нескольких строках.

5.4.4. Обработка пустых значений

Пока мы не ушли от темы пустых значений, обсудим, что с ними можно сделать. В разделе 5.2 рассказывалось, как заменять `NaN` константой с помощью метода `fillna`. Также их можно удалять.

Начнем этот раздел со сброса наших данных обратно в исходный вид. Импортируем CSV-файл заново с помощью функции `read_csv`:

```
In [54] employees = pd.read_csv(
        "employees.csv", parse_dates = ["Start Date"]
    )
```

Напомним, как выглядит его содержимое:

```
In [55] employees
```

```
Out [55]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
3	Jerry	NaN	2005-03-04	138705.0	True	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT
...
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales
1000	NaN	NaN	NaT	NaN	NaN	NaN

```
1001 rows x 6 columns
```

Метод `dropna` служит для удаления строк объекта `DataFrame`, содержащих какие-либо значения `NaN`. Неважно, сколько значений отсутствует в строке; этот метод удалит строку даже при наличии в ней одного-единственного значения `NaN`. В объекте `DataFrame` с данными по сотрудникам отсутствует значение по индексу 0 в столбце `Salary`, индексу 1 в столбце `Team`, индексу 2 в столбце `Start Date` и индексу 3 в столбце `Gender`. Как видно из следующих результатов, библиотека `pandas` удалила все эти строки:

```
In [56] employees.dropna()
```

```
Out [56]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
4	Larry	Male	1998-01-24	101004.0	True	IT
5	Dennis	Male	1987-04-18	115163.0	False	Legal
6	Ruby	Female	1987-08-17	65476.0	True	Product
8	Angela	Female	2005-11-22	95570.0	True	Engineering
9	Frances	Female	2002-08-08	139852.0	True	Business Dev
...
994	George	Male	2013-06-21	98874.0	True	Marketing
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales

```
761 rows x 6 columns
```

Для удаления только строк, в которых отсутствуют все значения, можно передать методу `dropna` параметр `how` с аргументом `"all"`. Только одна строка в наборе данных, последняя, удовлетворяет этому условию:

```
In [57] employees.dropna(how = "all").tail()
```

```
Out [57]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
995	Henry	NaN	2014-11-23	132483.0	False	Distribution
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales

Для удаления строк, в которых отсутствует значение в конкретном столбце, служит параметр `subset`. В следующем примере мы удаляем строки, в которых отсутствует значение в столбце `Gender`:

```
In [59] employees.dropna(subset = ["Gender"]).tail()
```

```
Out [59]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
994	George	Male	2013-06-21	98874.0	True	Marketing
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales

В параметр `subset` можно передать также список столбцов. Библиотека `pandas` при этом удалит строку, в которой отсутствует значение в любом из указанных столбцов. В следующем примере удаляются строки, в которых отсутствует значение в столбце `Start Date`, столбце `Salary` или и в том и в другом одновременно:

```
In [60] employees.dropna(subset = ["Start Date", "Salary"]).head()
```

```
Out [60]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
1	Thomas	Male	1996-03-31	61933.0	True	NaN
3	Jerry	NaN	2005-03-04	138705.0	True	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT
5	Dennis	Male	1987-04-18	115163.0	False	Legal
6	Ruby	Female	1987-08-17	65476.0	True	Product

Параметр `thresh` позволяет задавать минимальное количество непустых значений в строке, при котором библиотека `pandas` не будет ее удалять:

```
In [61] employees.dropna(how = "any", thresh = 4).head()
```

```
Out [61]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
3	Jerry	NaN	2005-03-04	138705.0	True	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT

Параметр `thresh` очень удобен, когда при определенном количестве отсутствующих значений строка становится бесполезной для анализа.

5.5. РЕШЕНИЕ ПРОБЛЕМЫ ДУБЛИКАТОВ

Отсутствующие значения часто встречаются в запутанных наборах данных, но столь же часто встречаются и дублирующиеся значения. К счастью, библиотека `pandas` предоставляет несколько методов для обнаружения и удаления дубликатов.

5.5.1. Метод `duplicated`

Во-первых, напомним значения столбца `Team` в первых пяти строках. Обратите внимание, что значение `"Finance"` встречается на позициях с индексами 2 и 3:

```
In [62] employees["Team"].head()
```

```
Out [62] 0    Marketing
         1         NaN
         2    Finance
         3    Finance
         4         IT
         Name: Team, dtype: object
```

Метод `duplicated` возвращает булев объект `Series`, указывающий на дублирующиеся значения в столбце. Библиотека `pandas` возвращает `True` при обнаружении значения, которое ранее уже встречалось ей в этом столбце. Рассмотрим следующий пример. Метод `duplicated` отмечает первое вхождение `"Finance"` в столбце `Team` значением `False` как недублирующее. А все последующие вхождения

"Finance" отмечает как дублирующие (True). Та же логика применяется и ко всем прочим значениям столбца Team:

```
In [63] employees["Team"].duplicated().head()
```

```
Out [63] 0    False
         1    False
         2    False
         3     True
         4    False
         Name: Team, dtype: bool
```

Параметр `keep` метода `duplicated` указывает библиотеке pandas, какое из дублирующих значений оставлять. Аргумент по умолчанию, "first", позволяет оставить первое вхождение каждого из дублирующих значений. Следующий код эквивалентен приведенному выше:

```
In [64] employees["Team"].duplicated(keep = "first").head()
```

```
Out [64] 0    False
         1    False
         2    False
         3     True
         4    False
         Name: Team, dtype: bool
```

Можно также попросить библиотеку pandas пометить последнее из вхождений значения в столбце как недублирующее. Для этого необходимо задать аргумент "last" для параметра `keep`:

```
In [65] employees["Team"].duplicated(keep = "last")
```

```
Out [65] 0         True
         1         True
         2         True
         3         True
         4         True
         ...
        996        False
        997        False
        998        False
        999        False
       1000        False
         Name: Team, Length: 1001, dtype: bool
```

Предположим, нам нужно извлечь по одному сотруднику из каждой группы. Одна из возможных стратегий — извлечь первую строку для каждой уникальной группы из столбца Team. Метод `duplicated` возвращает булев объект Series; True указывает на дублирующие значения, начиная со второго вхождения. Если

вычислить логическое отрицание этого объекта `Series`, получится объект `Series`, в котором `True` отмечает первое встреченное библиотекой `pandas` вхождение значения:

```
In [66] (~employees["Team"].duplicated()).head()
```

```
Out [66] 0      True
         1      True
         2      True
         3     False
         4      True
         Name: Team, dtype: bool
```

Теперь можно извлечь по одному сотруднику из команды, передав булев объект `Series` в квадратных скобках. Библиотека `pandas` включит в результат строки с первыми вхождениями значений в столбце `Team`. Обратите внимание, что библиотека `pandas` считает все `NaN` уникальными значениями:

```
In [67] first_one_in_team = ~employees["Team"].duplicated()
        employees[first_one_in_team]
```

```
Out [67]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT
5	Dennis	Male	1987-04-18	115163.0	False	Legal
6	Ruby	Female	1987-08-17	65476.0	True	Product
8	Angela	Female	2005-11-22	95570.0	True	Engineering
9	Frances	Female	2002-08-08	139852.0	True	Business Dev
12	Brandon	Male	1980-12-01	112807.0	True	HR
13	Gary	Male	2008-01-27	109831.0	False	Sales
40	Michael	Male	2008-10-10	99283.0	True	Distribution

Из этих результатов ясно, что Дуглас — первый сотрудник группы `Marketing` в нашем наборе данных, Томас — первый в команде без названия, Мария — первая в команде `Finance` и т. д.

5.5.2. Метод `drop_duplicates`

Метод `drop_duplicates` объектов `DataFrame` — удобное сокращенное написание для операции из подраздела 5.5.1. По умолчанию этот метод удаляет строки, в которых все значения совпадают с уже встречавшейся строкой. Сотрудников,

у которых совпадают все шесть значений строк, у нас нет, так что при стандартной форме вызова метод ничего не делает:

```
In [68] employees.drop_duplicates()
```

```
Out [68]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
3	Jerry	NaN	2005-03-04	138705.0	True	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT
...
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales
1000	NaN	NaN	NaT	NaN	NaN	NaN

```
1001 rows x 6 columns
```

Но этому методу можно передать параметр `subset` со списком столбцов, которые библиотека pandas должна учитывать при определении уникальности строки. В следующем примере мы находим первое вхождение каждого из уникальных значений в столбце `Team`. Другими словами, библиотека pandas оставляет строку только в том случае, если она содержит первое вхождение значения столбца `Team` (например, "Marketing"). Она исключает все строки с дублирующими первое вхождение значениями столбца `Team`:

```
In [69] employees.drop_duplicates(subset = ["Team"])
```

```
Out [69]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT
5	Dennis	Male	1987-04-18	115163.0	False	Legal
6	Ruby	Female	1987-08-17	65476.0	True	Product
8	Angela	Female	2005-11-22	95570.0	True	Engineering
9	Frances	Female	2002-08-08	139852.0	True	Business Dev
12	Brandon	Male	1980-12-01	112807.0	True	HR
13	Gary	Male	2008-01-27	109831.0	False	Sales
40	Michael	Male	2008-10-10	99283.0	True	Distribution

Метод `drop_duplicates` также принимает параметр `keep`. При передаче этому параметру аргумента `"last"` библиотека `pandas` оставит строки с последними вхождениями каждого из дублирующих значений. Вероятно, эти строки будут располагаться ближе к концу набора данных. В примере ниже Элис — последний сотрудник в наборе данных из группы HR, Джастин — последний сотрудник из группы Legal и т. д.:

```
In [70] employees.drop_duplicates(subset = ["Team"], keep = "last")
```

```
Out [70]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
988	Alice	Female	2004-10-05	47638.0	False	HR
989	Justin	NaN	1991-02-10	38344.0	False	Legal
990	Robin	Female	1987-07-24	100765.0	True	IT
993	Tina	Female	1997-05-15	56450.0	True	Engineering
994	George	Male	2013-06-21	98874.0	True	Marketing
995	Henry	NaN	2014-11-23	132483.0	False	Distribution
996	Phillip	Male	1984-01-31	42392.0	False	Finance
997	Russell	Male	2013-05-20	96914.0	False	Product
998	Larry	Male	2013-04-20	60500.0	False	Business Dev
999	Albert	Male	2012-05-15	129949.0	True	Sales
1000	NaN	NaN	NaT	NaN	NaN	NaN

Существует еще одна опция параметра `keep`. Можно передать в него аргумент `False`, чтобы исключить все строки с дублирующими значениями. Библиотека `pandas` будет отбрасывать любую строку, если существуют какие-либо другие строки с тем же значением. В следующем примере мы фильтруем `employees` на предмет строк с уникальным значением в столбце `First Name`. Другими словами, нижеприведенные имена встречаются в нашем объекте `DataFrame` только один раз:

```
In [71] employees.drop_duplicates(subset = ["First Name"], keep = False)
```

```
Out [71]
```

	First Name	Gender	Start Date	Salary	Mgmt	Team
5	Dennis	Male	1987-04-18	115163.0	False	Legal
8	Angela	Female	2005-11-22	95570.0	True	Engineering
33	Jean	Female	1993-12-18	119082.0	False	Business Dev
190	Carol	Female	1996-03-19	57783.0	False	Finance
291	Tammy	Female	1984-11-11	132839.0	True	IT
495	Eugene	Male	1984-05-24	81077.0	False	Sales
688	Brian	Male	2007-04-07	93901.0	True	Legal
832	Keith	Male	2003-02-12	120672.0	False	Legal
887	David	Male	2009-12-05	92242.0	False	Legal

Допустим, нам нужно идентифицировать дубликаты по сочетанию значений из различных столбцов. Например, первое вхождение каждого сотрудника с уникальной комбинацией имени и пола в наборе данных. Ниже показано подмножество всех сотрудников со значением "Douglas" в столбце **First Name** и "Male" в столбце **Gender**, нам предстоит еще чуть позже обратиться к этим результатам:

```
In [72] name_is_douglas = employees["First Name"] == "Douglas"
        is_male = employees["Gender"] == "Male"
        employees[name_is_douglas & is_male]
```

Out [72]

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
217	Douglas	Male	1999-09-03	83341.0	True	IT
322	Douglas	Male	2002-01-08	41428.0	False	Product
835	Douglas	Male	2007-08-04	132175.0	False	Engineering

В параметр `subset` метода `drop_duplicates` можно передать список столбцов. Библиотека pandas будет определять наличие дублирующих значений по этим столбцам. В следующем примере мы находим дубликаты по сочетанию значений столбцов **Gender** и **Team**:

```
In [73] employees.drop_duplicates(subset = ["Gender", "Team"]).head()
```

Out [73]

	First Name	Gender	Start Date	Salary	Mgmt	Team
0	Douglas	Male	1993-08-06	NaN	True	Marketing
1	Thomas	Male	1996-03-31	61933.0	True	NaN
2	Maria	Female	NaT	130590.0	False	Finance
3	Jerry	NaN	2005-03-04	138705.0	True	Finance
4	Larry	Male	1998-01-24	101004.0	True	IT

Давайте проанализируем эти результаты строка за строкой. Строка с индексом 0 содержит первое вхождение имени "Douglas" и пола "Male" в наборе данных сотрудников. Библиотека pandas исключит из итогового набора данных все прочие строки с этими двумя значениями в соответствующих столбцах. Уточним, что pandas по-прежнему будет включать строку с именем "Douglas", но полом, не равным "Male". Аналогично будут включены строки с полом "Male", но не равным "Douglas" именем. Очевидно, что библиотека pandas распознает дубликаты по сочетанию значений этих двух столбцов.

5.6. УПРАЖНЕНИЯ

На горизонте показался ваш шанс попрактиковаться в использовании изложенных в этой главе идей, не упустите его!

5.6.1. Задачи

Набор данных `netflix.csv` представляет собой коллекцию из почти 6000 фильмов и сериалов, доступных в ноябре 2019 года для просмотра на сервисе потокового видео Netflix. Он включает четыре столбца: название видео, режиссера, дату его добавления и тип/категорию. Некоторые значения в столбцах `director` и `date_added` отсутствуют, что видно на позициях с индексами 0, 2 и 5836:

```
In [74] pd.read_csv("netflix.csv")
```

```
Out [74]
```

	title	director	date_added	type
0	Alias Grace	NaN	3-Nov-17	TV Show
1	A Patch of Fog	Michael Lennox	15-Apr-17	Movie
2	Lunatics	NaN	19-Apr-19	TV Show
3	Uriyadi 2	Vijay Kumar	2-Aug-19	Movie
4	Shrek the Musical	Jason Moore	29-Dec-13	Movie
...
5832	The Pursuit	John Papola	7-Aug-19	Movie
5833	Hurricane Bianca	Matt Kugelman	1-Jan-17	Movie
5834	Amar's Hands	Khaled Youssef	26-Apr-19	Movie
5835	Bill Nye: Science Guy	Jason Sussberg	25-Apr-18	Movie
5836	Age of Glory	NaN	NaN	TV Show

```
5837 rows x 4 columns
```

Используя полученные в этой главе навыки, решите следующие задачи.

1. Добейтесь оптимального использования этим набором данных памяти и максимального удобства его применения.
2. Найдите все строки с названием `Limitless`.
3. Найдите все строки с режиссером `Robert Rodriguez` и типом `Movie`.
4. Найдите все строки с датой добавления `2019-07-31` или режиссером `Robert Altman`.
5. Найдите все строки с режиссером `Orson Welles`, `Aditya Kripalani` или `Sam Raimi`.

6. Найдите все строки со значением столбца `date_added` между 1 мая и 1 июня 2019 года.
7. Удалите все строки с `NaN` в столбце `director`.
8. Найдите все дни, в которые Netflix добавил в каталог только одну картину.

5.6.2. Решения

Приступим к решению задач!

1. Первый шаг на пути оптимизации набора данных — преобразование значений столбца `date_added` в метки даты/времени. Произвести это преобразование можно во время импорта набора данных с помощью параметра `parse_dates` функции `read_csv`:

```
In [75] netflix = pd.read_csv("netflix.csv", parse_dates = ["date_added"])
```

Иницилируем анализ расходования ресурсов, оценим, сколько памяти сейчас занимает набор:

```
In [76] netflix.info()
```

```
Out [76]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5837 entries, 0 to 5836
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   title       5837 non-null   object
1   director    3936 non-null   object
2   date_added  5195 non-null   datetime64[ns]
3   type        5837 non-null   object
dtypes: datetime64[ns](1), object(3)
memory usage: 182.5+ KB
```

Можно ли преобразовать значения еще какого-нибудь столбца в другой тип данных? Как насчет категориальных значений? Воспользуемся методом `nunique` и подсчитаем количество уникальных значений в каждом из столбцов:

```
In [77] netflix.nunique()
```

```
Out [77] title       5780
         director    3024
         date_added   1092
         type         2
         dtype: int64
```

Столбец `type` идеально подходит для категориальных значений. В наборе данных из 5837 строк он содержит только два уникальных значения: "Movie"

и "TV Show". Преобразовать его значения можно с помощью метода `astype`. Перепишем исходный объект `Series`:

```
In [78] netflix["type"] = netflix["type"].astype("category")
```

На сколько это преобразование сократило объем занимаемой памяти? На целых 22 %:

```
In [79] netflix.info()
```

```
Out [79]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5837 entries, 0 to 5836
Data columns (total 4 columns):
#   Column          Non-Null Count  Dtype
---  -
0   title            5837 non-null   object
1   director         3936 non-null   object
2   date_added       5195 non-null   datetime64[ns]
3   type             5837 non-null   category
dtypes: category(1), datetime64[ns](1), object(2)
memory usage: 142.8+ KB
```

- Для сравнения значений из столбца `title` со строковым значением "Limitless" мы воспользуемся оператором равенства. А затем извлечем на основе полученного булевого объекта `Series` строки объекта `netflix`, для которых оператор равенства вернул `True`:

```
In [80] netflix[netflix["title"] == "Limitless"]
```

```
Out [80]
```

	title	director	date_added	type
1559	Limitless	Neil Burger	2019-05-16	Movie
2564	Limitless	NaN	2016-07-01	TV Show
4579	Limitless	Vrinda Samarth	2019-10-01	Movie

- Для извлечения фильмов режиссера Роберта Родригеса нам понадобится два булевых объекта `Series`: один для сравнения значений столбца `director` с "Robert Rodriguez", а второй — для сравнения значений столбца `type` с "Movie". Операцию логического И к этим двум булевым объектам `Series` мы применяем с помощью символа `&`:

```
In [81] directed_by_robert_rodriguez = (
        netflix["director"] == "Robert Rodriguez"
    )
    is_movie = netflix["type"] == "Movie"
    netflix[directed_by_robert_rodriguez & is_movie]
```

```
Out [81]
```

		title	director	date_added	type

1384	Spy Kids: All the Time in the ...	Robert Rodriguez	2019-02-19	Movie	
1416	Spy Kids 3: Game...	Robert Rodriguez	2019-04-01	Movie	
1460	Spy Kids 2: The Island of Lost D...	Robert Rodriguez	2019-03-08	Movie	
2890	Sin City	Robert Rodriguez	2019-10-01	Movie	
3836	Shorts	Robert Rodriguez	2019-07-01	Movie	
3883	Spy Kids	Robert Rodriguez	2019-04-01	Movie	

4. Следующая задача: найти все картины со значением "2019-07-31" в столбце `date_added` или "Robert Altman" в столбце `director`. Задача аналогична предыдущей, только теперь для операции логического ИЛИ понадобится символ `|`:

```
In [82] added_on_july_31 = netflix["date_added"] == "2019-07-31"
        directed_by_altman = netflix["director"] == "Robert Altman"
        netflix[added_on_july_31 | directed_by_altman]
```

Out [82]

		title	director	date_added	type

611	Popeye	Robert Altman	2019-11-24	Movie	
1028	The Red Sea Diving Resort	Gideon Raff	2019-07-31	Movie	
1092	Gosford Park	Robert Altman	2019-11-01	Movie	
3473	Bangkok Love Stories: Innocence	NaN	2019-07-31	TV Show	
5117	Ramen Shop	Eric Khoo	2019-07-31	Movie	

5. В следующей задаче нужно найти записи, в которых режиссер — Orson Welles, Aditya Kripalani или Sam Raimi. Один из вариантов ее решения — создать три булевых объекта `Series`, по одному для каждого из режиссеров, а затем воспользоваться оператором `|`. Но более лаконичный и лучше масштабируемый подход к генерации булева объекта `Series` — вызвать метод `isin` для столбца `director`, передав ему список режиссеров:

```
In [83] directors = ["Orson Welles", "Aditya Kripalani", "Sam Raimi"]
        target_directors = netflix["director"].isin(directors)
        netflix[target_directors]
```

Out [83]

		title	director	date_added	type

946	The Stranger	Orson Welles	2018-07-19	Movie	
1870	The Gift	Sam Raimi	2019-11-20	Movie	
3706	Spider-Man 3	Sam Raimi	2019-11-01	Movie	
4243	Tikli and Laxmi Bomb	Aditya Kripalani	2018-08-01	Movie	
4475	The Other Side of the Wind	Orson Welles	2018-11-02	Movie	
5115	Tottaa Pataaka Item Maal	Aditya Kripalani	2019-06-25	Movie	

6. Наиболее лаконичный способ найти все строки с датой добавления между 1 мая и 1 июня 2019 года — воспользоваться методом `between`, указав вышеупомянутые две даты в качестве верхней и нижней границ диапазона. При таком подходе нам не нужны два отдельных булевых объекта `Series`:

```
In [84] may_movies = netflix["date_added"].between(
        "2019-05-01", "2019-06-01"
    )
    netflix[may_movies].head()
```

Out [84]

	title	director	date_added	type
29	Chopsticks	Sachin Yardi	2019-05-31	Movie
60	Away From Home	NaN	2019-05-08	TV Show
82	III Smoking Barrels	Sanjib Dey	2019-06-01	Movie
108	Jailbirds	NaN	2019-05-10	TV Show
124	Pegasus	Han Han	2019-05-31	Movie

7. Для удаления строк объекта `DataFrame` с отсутствующими значениями служит метод `dropna`. Нам придется воспользоваться для него параметром `subset`, чтобы ограничить набор столбцов, в которых библиотека `pandas` будет искать пустые значения. В этой задаче нас интересуют значения `NaN` в столбце `director`:

```
In [85] netflix.dropna(subset = ["director"]).head()
```

Out [85]

	title	director	date_added	type
1	A Patch of Fog	Michael Lennox	2017-04-15	Movie
3	Uriyadi 2	Vijay Kumar	2019-08-02	Movie
4	Shrek the Musical	Jason Moore	2013-12-29	Movie
5	Schubert In Love	Lars Büchel	2018-03-01	Movie
6	We Have Always Lived in the Castle	Stacie Passon	2019-09-14	Movie

8. Последняя задача — найти дни, в которые Netflix добавлял в каталог только один фильм. Одно из решений — проверить наличие дублирующих значений в столбце `date_added` для добавленных в один день картин. Для этого можно вызвать метод `drop_duplicates` с аргументом `date_added` для параметра `subset` и значением `False` параметра `keep`. В результате библиотека `pandas` удалит все строки с дублирующими значениями в столбце `date_added`. Полученный в результате объект `DataFrame` будет содержать только те картины, которые были единственными добавленными в соответствующие дни:

```
In [86] netflix.drop_duplicates(subset = ["date_added"], keep = False)
```

Out [86]

	title	director	date_added	type
4	Shrek the Musical	Jason Moore	2013-12-29	Movie
12	Without Gorky	Cosima Spender	2017-05-31	Movie
30	Anjelah Johnson: Not Fancy	Jay Karas	2015-10-02	Movie
38	One Last Thing	Tim Rouhana	2019-08-25	Movie
70	Marvel's Iron Man & Hulk: Heroes ...	Leo Riley	2014-02-16	Movie
...
5748	Menorca	John Barnard	2017-08-27	Movie
5749	Green Room	Jeremy Saulnier	2018-11-12	Movie
5788	Chris Brown: Welcome to My Life	Andrew Sandler	2017-10-07	Movie
5789	A Very Murray Christmas	Sofia Coppola	2015-12-04	Movie
5812	Little Singham in London	Prakash Satam	2019-04-22	Movie

391 rows x 4 columns

Поздравляю с успешным выполнением упражнений!

РЕЗЮМЕ

- Метод `astype` преобразует значения объекта `Series` в другой тип данных.
- Тип данных `category` идеально подходит для столбцов объектов `Series`, содержащих лишь небольшое количество уникальных значений.
- Библиотека `pandas` может извлекать подмножества данных из объектов `DataFrame` по одному или нескольким условиям.
- Для извлечения подмножества объекта `DataFrame` необходимо указать после его названия булев объект `Series` в квадратных скобках.
- Для сравнения каждой из записей объекта `Series` с константой можно использовать операторы «равно», «не равно» и прочие математические операторы.
- Символ `&` требует выполнения всех условий для извлечения строки.
- Символ `|` требует выполнения хотя бы одного из условий для извлечения строки.
- Вспомогательные методы `isnull`, `notnull`, `between` и `duplicated` возвращают булевы объекты `Series`, на основе которых можно затем фильтровать данные.
- Метод `fillna` заменяет `NaN` константой.
- Метод `dropna` удаляет строки с пустыми значениями. Варьируя значения его параметров, можно искать пустые значения в одном или нескольких столбцах.

Часть II

Библиотека *pandas* на практике

В части I был заложен фундамент вашего владения *pandas*. Теперь, когда вы уже освоились с объектами *Series* и *DataFrame*, можно расширить горизонты и научиться решать основные задачи анализа данных. В главе 6 мы займемся обработкой зашумленных текстовых данных, включая удаление пробельных символов и исправление несогласованного регистра символов. В главе 7 вы научитесь использовать замечательный класс *MultiIndex* для хранения и извлечения иерархических данных. Главы 8 и 9 посвящены агрегированию: созданию сводных таблиц для объектов *DataFrame*, группировке данных по корзинам, обобщению данных и многому другому. В главе 10 рассмотрим объединение нескольких наборов данных в один с помощью различных видов соединений. И сразу после этого, в главе 11, вы изучите все нюансы работы с другим распространенным типом данных — метками даты/времени. В главе 12 обсуждается импорт и экспорт наборов данных в библиотеку *pandas* и из нее. Глава 13 охватывает вопросы настройки параметров библиотеки *pandas*. И наконец, в главе 14 вы найдете руководство по созданию визуализаций данных на основе объектов *DataFrame*.

А по ходу дела мы опробуем различные возможности библиотеки *pandas* на практике, на более чем 30 наборах данных, охватывающих все, от имен детей до сухих завтраков, от компаний из списка Fortune 1000 до лауреатов Нобелевской премии. Вы можете читать главы последовательно или изучать выборочно те вопросы, которые вас особо заинтересовали. Можете считать каждую главу новым инструментом в копилке ваших знаний библиотеки *pandas*. Удачи!



Работа с текстовыми данными

В этой главе

- ✓ Удаление пробелов из строковых значений.
- ✓ Приведение строковых значений к нижнему и верхнему регистру.
- ✓ Поиск и замена символов в строковых значениях.
- ✓ Срез строкового значения по позициям индекса символов.
- ✓ Разбиение строковых значений по разделителю.

Текстовые данные нередко оказываются сильно зашумленными. Встречающиеся на практике наборы данных переполнены неправильными символами, буквами не в том регистре, лишними пробелами и т. д. Процесс очистки данных называется «выпасом» (wrangling, munging). Зачастую именно ему посвящена большая часть анализа данных. Даже если практически сразу известно, какую именно информацию мы хотим извлечь из данных, сложность часто заключается в приведении данных в подходящую для этой операции форму. К счастью, одна из основных идей pandas как раз и состоит в упрощении очистки неправильно отформатированных текстовых значений. Библиотека pandas проверена на практике и очень гибка. В этой главе вы научитесь исправлять с ее помощью самые разнообразные изъяны в текстовых наборах данных. Объем материала очень велик, так что приступим не откладывая в долгий ящик.

6.1. РЕГИСТР БУКВ И ПРОБЕЛЫ

Начнем с импорта библиотеки `pandas` в новый блокнот Jupyter:

```
In [1] import pandas as pd
```

Первый набор данных этой главы, `chicago_food_inspections.csv`, представляет собой список из результатов более чем 150 000 проведенных в Чикаго инспекций по проверке качества пищевых продуктов. CSV-файл содержит только два столбца: название заведения и уровень риска. Всего существует четыре уровня риска: Risk 1 (High), Risk 2 (Medium), Risk 3 (Low) — и отдельный вариант All для самых худших заведений:

```
In [2] inspections = pd.read_csv("chicago_food_inspections.csv")
      inspections
```

```
Out [2]
```

	Name	Risk
0	MARRIOT MARQUIS CHICAGO	Risk 1 (High)
1	JETS PIZZA	Risk 2 (Medium)
2	ROOM 1520	Risk 3 (Low)
3	MARRIOT MARQUIS CHICAGO	Risk 1 (High)
4	CHARTWELLS	Risk 1 (High)
...
153805	WOLCOTT'S	Risk 1 (High)
153806	DUNKIN DONUTS/BASKIN-ROBBINS	Risk 2 (Medium)
153807	Cafe 608	Risk 1 (High)
153808	mr.daniel's	Risk 1 (High)
153809	TEMPO CAFE	Risk 1 (High)

```
153810 rows x 2 columns
```

ПРИМЕЧАНИЕ

`chicago_food_inspections.csv` — модифицированная версия набора данных, предоставленного правительством Чикаго (<http://mng.bz/9N60>). В данных изначально присутствовали опечатки и расхождения; я постарался их сохранить, чтобы показать вам встречающиеся на практике отклонения в данных. Подумайте на досуге, как можно оптимизировать эти данные с помощью методик, которые вы изучите в этой главе.

Сразу же замечаем проблему в столбце `Name`: различные регистры букв. Большинство значений строк — в верхнем регистре, некоторые — полностью в нижнем (`mr.daniel's`), а часть — в обычном представлении (`Cafe 608`).

В выведенных выше результатах незаметна еще одна скрывающаяся в `inspections` проблема: значения столбца `Name` окружены пробелами. Их будет легче заметить,

если отделить объект `Series Name` с помощью синтаксиса с квадратными скобками. Обратите внимание на неровный ряд концов строк с учетом того, что они выровнены по правому краю:

```
In [3] inspections["Name"].head()

Out [3] 0      MARRIOT MARQUIS CHICAGO
        1              JETS PIZZA
        2              ROOM 1520
        3      MARRIOT MARQUIS CHICAGO
        4              CHARTWELLS
        Name: Name, dtype: object
```

Получить `ndarray` библиотеки NumPy, в котором фактически хранятся значения, можно с помощью атрибута `values` нашего объекта `Series`. Становится заметно, что пробелы встречаются как в начале, так и в конце значений:

```
In [4] inspections["Name"].head().values

Out [4] array([' MARRIOT MARQUIS CHICAGO ', ' JETS PIZZA ',
              ' ROOM 1520 ', ' MARRIOT MARQUIS CHICAGO ',
              ' CHARTWELLS  '], dtype=object)
```

Сначала займемся пробелами, а потом уже регистром символов.

Атрибут `str` объекта `Series` позволяет получить доступ к объекту `StringMethods` — замечательному набору инструментов для работы со строковыми значениями:

```
In [5] inspections["Name"].str

Out [5] <pandas.core.strings.StringMethods at 0x122ad8510>
```

Для выполнения операций со строковыми значениями лучше вызывать методы объекта `StringMethods`, а не самого объекта `Series`. Некоторые методы pandas аналогичны нативным методам для работы со строковыми значениями языка Python, а другие доступны только в pandas. Всесторонний обзор методов для работы со строковыми значениями языка Python можно найти в приложении Б.

Удалить пробельные символы из строкового значения можно с помощью методов из семейства `strip`. Метод `lstrip` (left strip — «очистка слева») удаляет пробелы из начала строки. Вот простейший пример:

```
In [6] dessert = "  cheesecake  "
        dessert.lstrip()

Out [6] 'cheesecake  '
```


Метод `rstrip` (right strip — «очистка справа») удаляет пробелы из конца строки:

```
In [7] dessert.rstrip()
```

```
Out [7] 'cheesecake'
```

Метод `strip` удаляет пробелы с обоих концов строки:

```
In [8] dessert.strip()
```

```
Out [8] 'cheesecake'
```

Эти три метода доступны в объекте `StringMethods`. Каждый из них возвращает новый объект `Series`, в котором содержатся значения уже после применения соответствующей операции. Попробуем вызывать эти методы по очереди:

```
In [9] inspections["Name"].str.lstrip().head()
```

```
Out [9] 0      MARRIOT MARQUIS CHICAGO
        1              JETS PIZZA
        2              ROOM 1520
        3      MARRIOT MARQUIS CHICAGO
        4              CHARTWELLS
        Name: Name, dtype: object
```

```
In [10] inspections["Name"].str.rstrip().head()
```

```
Out [10] 0      MARRIOT MARQUIS CHICAGO
         1              JETS PIZZA
         2              ROOM 1520
         3      MARRIOT MARQUIS CHICAGO
         4              CHARTWELLS
         Name: Name, dtype: object
```

```
In [11] inspections["Name"].str.strip().head()
```

```
Out [11] 0      MARRIOT MARQUIS CHICAGO
         1              JETS PIZZA
         2              ROOM 1520
         3      MARRIOT MARQUIS CHICAGO
         4              CHARTWELLS
         Name: Name, dtype: object
```

Теперь можно перезаписать наш объект `Series` новым, со значениями без лишних пробелов. Справа от знака равенства мы создадим новый объект `Series` с помощью метода `strip`. Слева же мы указываем перезаписываемый столбец посредством синтаксиса с квадратными скобками. Python обрабатывает сначала код

справа от знака равенства. Таким образом, мы создали новый объект `Series` без пробелов на основе столбца `Name`, а затем записали этот новый объект `Series` на место существующего столбца `Name`:

```
In [12] inspections["Name"] = inspections["Name"].str.strip()
```

Такое однострочное решение подходит для маленького набора данных, но при большом количестве столбцов быстро становится неудобным. Как быстро применить одну и ту же логику ко всем столбцам объекта `DataFrame`? Вспомните, что через атрибут `columns` доступен итерируемый объект `Index`, содержащий названия столбцов объекта `DataFrame`:

```
In [13] inspections.columns
```

```
Out [13] Index(['Name', 'Risk'], dtype='object')
```

Можно воспользоваться циклом `for` языка Python, чтобы пройти в цикле по всем столбцам, динамически извлечь каждый из объекта `DataFrame`, вызвать метод `str.strip`, возвращающий новый объект `Series`, и затем перезаписать исходный столбец. Для описания такой логики достаточно двух строк кода:

```
In [14] for column in inspections.columns:
        inspections[column] = inspections[column].str.strip()
```

В объекте `StringMethods` доступны также все методы изменения регистра символов языка Python. Метод `lower`, например, приводит все символы строкового значения к нижнему регистру:

```
In [15] inspections["Name"].str.lower().head()
```

```
Out [15] 0    marriot marquis chicago
         1             jets pizza
         2             room 1520
         3    marriot marquis chicago
         4             chartwells
         Name: Name, dtype: object
```

Парный к нему метод `str.upper` возвращает объект `Series` со строковыми значениями в верхнем регистре. В следующем примере мы вызываем этот метод для другого объекта `Series`, поскольку столбец `Name` и так уже практически полностью в верхнем регистре:

```
In [16] steaks = pd.Series(["porterhouse", "filet mignon", "ribeye"])
        steaks
```

```
Out [16] 0    porterhouse
         1    filet mignon
         2    ribeye
         dtype: object
```

```
In [17] steaks.str.upper()
```

```
Out [17] 0    PORTERHOUSE
          1    FILET MIGNON
          2    RIBEYE
          dtype: object
```

Предположим, нам нужно получить названия заведений в более стандартизированном, удобочитаемом формате. Можно воспользоваться методом `str.capitalize`, чтобы привести в верхний регистр первую букву каждого строкового значения в объекте `Series`:

```
In [18] inspections["Name"].str.capitalize().head()
```

```
Out [18] 0    Marriot marquis chicago
          1             Jets pizza
          2             Room 1520
          3    Marriot marquis chicago
          4             Chartwells
          Name: Name, dtype: object
```

Что ж, это шаг в нужную сторону, но лучше, вероятно, будет воспользоваться методом `str.title`, приводящим в верхний регистр первую букву каждого слова. Библиотека `pandas` определяет по наличию пробела, где заканчивается одно слово и начинается другое:

```
In [19] inspections["Name"].str.title().head()
```

```
Out [19] 0    Marriot Marquis Chicago
          1             Jets Pizza
          2             Room 1520
          3    Marriot Marquis Chicago
          4             Chartwells
          Name: Name, dtype: object
```

Метод `title` — замечательный вариант для работы с названиями мест, стран, городов и Ф. И. О. людей.

6.2. СРЕЗЫ СТРОКОВЫХ ЗНАЧЕНИЙ

Обратим теперь наше внимание на столбец `Risk`. Значения его в каждой из строк содержат как числовое, так и категориальное обозначение степени риска (например, 1 и "High"). Напомню, как выглядит содержимое этого столбца:

```
In [20] inspections["Risk"].head()
```

```
Out [20]
0    Risk 1 (High)
1    Risk 2 (Medium)
2    Risk 3 (Low)
3    Risk 1 (High)
4    Risk 1 (High)
Name: Risk, dtype: object
```

Допустим, нам нужно извлечь из всех строк числовые показатели степени риска. Эта операция может показаться очень простой вследствие кажущегося единообразия форматов строк, но не будем терять бдительность. Такой большой набор данных может таить в себе неожиданности:

```
In [21] len(inspections)
```

```
Out [21] 153810
```

Все ли строки соответствуют формату "Числовой показатель риска (Уровень риска)"? Выяснить это можно с помощью метода `unique`, возвращающего NumPy-объект `ndarray` с уникальными значениями столбца:

```
In [22] inspections["Risk"].unique()
```

```
Out [22] array(['Risk 1 (High)', 'Risk 2 (Medium)', 'Risk 3 (Low)', 'All',
               nan], dtype=object)
```

Необходимо учесть два дополнительных значения: отсутствующие `NaN` и строковое значение `'All'`. Как поступать с этими значениями, определяет аналитик на основе бизнес-требований. Важны ли `NaN` и `'All'`, или их можно отбросить? Давайте пойдем в этом сценарии на компромисс: удалим отсутствующие значения `NaN` и заменим значения `"All"` на `"Risk 4 (Extreme)"`. Воспользовавшись этим подходом, мы сможем гарантировать единый формат всех значений столбца `Risk`.

Удалить пропущенные значения из объекта `Series` можно с помощью метода `dropna`, с которым вы познакомились в главе 5. Передадим ему через параметр `subset` список столбцов объекта `DataFrame`, в которых pandas должна искать `NaN`. Ниже пример с обработкой объекта `inspections`, в нем удаляются строки, содержащие `NaN` в столбце `Risk`:

```
In [23] inspections = inspections.dropna(subset = ["Risk"])
```

Снова просмотрим список уникальных значений в столбце `Risk`:

```
In [24] inspections["Risk"].unique()
```

```
Out [24] array(['Risk 1 (High)', 'Risk 2 (Medium)', 'Risk 3 (Low)', 'All'],
               dtype=object)
```

Заменить все вхождения одного значения другим можно с помощью удобного метода `replace` объектов `DataFrame`. В первом параметре этого метода, `to_replace`, указывается искомое значение, а во втором, `value`, — значение, которым необходимо заменить каждое его вхождение. В следующем примере строковые значения `"All"` заменяются на `"Risk 4 (Extreme)"`:

```
In [25] inspections = inspections.replace(
        to_replace = "All", value = "Risk 4 (Extreme)"
    )
```

Теперь все значения столбца Risk имеют один и тот же формат:

```
In [26] inspections["Risk"].unique()
```

```
Out [26] array(['Risk 1 (High)', 'Risk 2 (Medium)', 'Risk 3 (Low)',
               'Risk 4 (Extreme)'], dtype=object)
```

Ну и далее продолжим решать нашу исходную задачу по извлечению показателей риска из всех строк.

6.3. СРЕЗЫ СТРОКОВЫХ ЗНАЧЕНИЙ И ЗАМЕНА СИМВОЛОВ

Для извлечения подстрок символов из строковых значений по позициям индексов можно воспользоваться методом `slice` объекта `StringMethods`. В качестве аргумента этот метод принимает начальный и конечный индексы. Нижняя граница (начальная точка) включается в интервал, а верхняя (конечная точка) — нет.

Наши числовые показатели риска начинаются во всех строковых значениях на позиции с индексом 5. В следующем примере мы извлекаем символы позиции с индексом 5 до (не включая) позиции с индексом 6:

```
In [27] inspections["Risk"].str.slice(5, 6).head()
```

```
Out [27] 0    1
          1    2
          2    3
          3    1
          4    1
          Name: Risk, dtype: object
```

Вместо метода `slice` можно использовать синтаксис срезов списков языка Python (см. приложение Б). Следующий код возвращает тот же результат, что и предыдущий:

```
In [28] inspections["Risk"].str[5:6].head()
```

```
Out [28] 0    1
          1    2
          2    3
          3    1
          4    1
          Name: Risk, dtype: object
```

А если нам нужно извлечь из строковых значений категориальную градацию ("High", "Medium", "Low" и "All")? Эта задача усложняется различающейся длиной слов; извлечь одинаковое количество символов, начиная с начальной

позиции индекса, не получится. Возможны несколько решений. Мы обсудим наиболее ошибкоустойчивый вариант, регулярные выражения, в разделе 6.7.

А пока будем подходить к решению задачи постепенно. Начнем с извлечения категорий риска из строк с помощью метода `slice`. Если передать методу `slice` в качестве аргумента одно значение, библиотека pandas будет рассматривать его как нижнюю границу и извлекать символы вплоть до конца строкового значения.

В следующем примере мы извлекаем символы, начиная с позиции индекса 8 до конца строковых значений. Символ на позиции с индексом 8 — первый символ в типах риска (буква H в High, M в Medium, L в Low и E в Extreme):

```
In [29] inspections["Risk"].str.slice(8).head()
```

```
Out [29] 0      High)
         1    Medium)
         2      Low)
         3     High)
         4     High)
         Name: Risk, dtype: object
```

Можно было воспользоваться и синтаксисом срезов списков Python. В этом случае внутри квадратных скобок укажите начальную позицию индекса с последующим двоеточием. Результат получается тот же:

```
In [30] inspections["Risk"].str[8:].head()
```

```
Out [30] 0      High)
         1    Medium)
         2      Low)
         3     High)
         4     High)
         Name: Risk, dtype: object
```

Остается что-то сделать с надоедливymi закрывающими скобками. Замечательное решение — передать методу `str.slice` отрицательный аргумент. При отрицательном аргументе граница индекса отсчитывается от конца строкового значения: при -1 извлекаются символы, вплоть до последнего, при -2 — до предпоследнего и т. д. Извлечем подстроку, начиная с позиции индекса 8 до последнего символа в каждом из строковых значений:

```
In [31] inspections["Risk"].str.slice(8, -1).head()
```

```
Out [31] 0      High
         1    Medium
         2      Low
         3     High
         4     High
         Name: Risk, dtype: object
```

Получилось! Если вы предпочитаете синтаксис срезов языка Python, можете указать `-1` после двоеточия внутри квадратных скобок:

```
In [32] inspections["Risk"].str[8:-1].head()
```

```
Out [32] 0      High
         1    Medium
         2      Low
         3      High
         4      High
         Name: Risk, dtype: object
```

Для удаления закрывающих скобок можно также воспользоваться методом `str.replace`, заменяя закрывающую скобку пустой строкой — строковое значение без символов.

Все методы `str` возвращают новый объект `Series` со своим собственным атрибутом `str`. Это позволяет последовательно связывать цепочкой несколько строковых методов, просто ссылаясь при вызове каждого метода на атрибут `str`. В следующем примере мы связываем цепочкой методы `slice` и `replace`:

```
In [33] inspections["Risk"].str.slice(8).str.replace(")", "").head()
```

```
Out [33] 0      High
         1    Medium
         2      Low
         3      High
         4      High
         Name: Risk, dtype: object
```

Благодаря срезу, начинающемуся со средней позиции индекса, и удалению закрывающих скобок мы смогли выделить из всех строк уровни риска.

6.4. БУЛЕВЫ МЕТОДЫ

В разделе 6.3 вы познакомились с такими методами, как `upper` и `slice`. Они возвращают объекты `Series` со строковыми значениями. В объекте `StringMethods` есть и методы, возвращающие булевы объекты `Series`. Особенно эти методы удобны для фильтрации объектов `DataFrame`.

Пусть нам нужно выбрать все заведения со словом `Pizza` в названии. В классическом Python для поиска подстроки в строковом значении используется оператор `in`:

```
In [34] "Pizza" in "Jets Pizza"
```

```
Out [34] True
```

Основная проблема при сопоставлении строковых значений — регистр. Например, Python не найдет строковое значение "pizza" в "Jets Pizza" из-за различного регистра букв "P" и "p":

```
In [35] "pizza" in "Jets Pizza"
```

```
Out [35] False
```

Таким образом, чтобы решить эту задачу, необходимо обеспечить единый регистр всех значений столбца, прежде чем искать подстроку. А потом искать значение "pizza" в объекте `Series`, все значения в котором — в нижнем регистре, или "PIZZA" в объекте `Series`, все значения в котором — в верхнем регистре. Остановимся на первом варианте.

Метод `contains` проверяет, входит ли подстрока в каждое из значений объекта `Series`. Он возвращает `True`, если библиотека pandas нашла аргумент метода внутри значения строки, и `False`, если нет. Рассмотрим код, в котором сначала столбец `Name` приводится к нижнему регистру с помощью метода `lower`, а затем во всех его строках производится поиск подстроки "pizza":

```
In [36] inspections["Name"].str.lower().str.contains("pizza").head()
```

```
Out [36] 0      False
         1       True
         2      False
         3      False
         4      False
         Name: Name, dtype: bool
```

Мы получили булев объект `Series`, на основе которого можем теперь извлечь из набора данных все заведения со словом `Pizza` в названии:

```
In [37] has_pizza = inspections["Name"].str.lower().str.contains("pizza")
         inspections[has_pizza]
```

```
Out [37]
```

	Name	Risk
1	JETS PIZZA	Risk 2 (Medium)
19	NANCY'S HOME OF STUFFED PIZZA	Risk 1 (High)
27	NARY'S GRILL & PIZZA ,INC.	Risk 1 (High)
29	NARYS GRILL & PIZZA	Risk 1 (High)
68	COLUTAS PIZZA	Risk 1 (High)
...
153756	ANGELO'S STUFFED PIZZA CORP	Risk 1 (High)
153764	COCHIAROS PIZZA #2	Risk 1 (High)
153772	FERNANDO'S MEXICAN GRILL & PIZZA	Risk 1 (High)
153788	REGGIO'S PIZZA EXPRESS	Risk 1 (High)
153801	State Street Pizza Company	Risk 1 (High)

```
3992 rows x 2 columns
```


Обратите внимание, что pandas сохраняет исходный регистр значений в столбце `Name`. Объект `DataFrame` `inspections` не меняется. Метод `lower` возвращает новый объект `Series`, а вызываемый для него метод `contains` возвращает еще один новый объект `Series`, на основе которого библиотека pandas фильтрует строки из исходного объекта `DataFrame`.

А если мы хотим уточнить выборку, например извлечь все названия заведений, которые начинаются с `tacos`? Именно начинаются! Для нас оказывается важна позиция разыскиваемой подстроки в строковом значении. Метод `str.startswith` решает эту задачу, возвращая `True`, если строка начинается с переданного ему аргумента:

```
In [38] inspections["Name"].str.lower().str.startswith("tacos").head()
```

```
Out [38] 0    False
         1    False
         2    False
         3    False
         4    False
         Name: Name, dtype: bool
```

```
In [39] starts_with_tacos = (
        inspections["Name"].str.lower().str.startswith("tacos")
    )

    inspections[starts_with_tacos]
```

```
Out [39]
```

	Name	Risk
69	TACOS NIETOS	Risk 1 (High)
556	TACOS EL TIO 2 INC.	Risk 1 (High)
675	TACOS DON GABINO	Risk 1 (High)
958	TACOS EL TIO 2 INC.	Risk 1 (High)
1036	TACOS EL TIO 2 INC.	Risk 1 (High)
...
143587	TACOS DE LUNA	Risk 1 (High)
144026	TACOS GARCIA	Risk 1 (High)
146174	Tacos Place's 1	Risk 1 (High)
147810	TACOS MARIO'S LIMITED	Risk 1 (High)
151191	TACOS REYNA	Risk 1 (High)

```
105 rows x 2 columns
```

Дополняющий его метод `str.endswith` ищет заданную подстроку в конце строковых значений объекта `Series`:

```
In [40] ends_with_tacos = (
        inspections["Name"].str.lower().str.endswith("tacos")
    )

    inspections[ends_with_tacos]
```

```
Out [40]
```

	Name	Risk
382	LAZO'S TACOS	Risk 1 (High)
569	LAZO'S TACOS	Risk 1 (High)
2652	FLYING TACOS	Risk 3 (Low)
3250	JONY'S TACOS	Risk 1 (High)
3812	PACO'S TACOS	Risk 1 (High)
...
151121	REYES TACOS	Risk 1 (High)
151318	EL MACHO TACOS	Risk 1 (High)
151801	EL MACHO TACOS	Risk 1 (High)
153087	RAYMOND'S TACOS	Risk 1 (High)
153504	MIS TACOS	Risk 1 (High)

304 rows x 2 columns

Ищите ли вы текст в начале, в середине или в конце строкового значения, в объекте `StringMethods` найдется подходящий метод.

6.5. РАЗБИЕНИЕ СТРОКОВЫХ ЗНАЧЕНИЙ

Следующий набор данных — список вымышленных покупателей. В каждой строке набора приведены имя (**Name**) и адрес (**Address**) покупателя. Давайте импортируем файл `customers.csv` с помощью функции `read_csv` и присвоим полученный объект `DataFrame` переменной `customers`:

```
In [41] customers = pd.read_csv("customers.csv")
        customers.head()
```

Out [41]

	Name	Address
0	Frank Manning	6461 Quinn Groves, East Matthew, New Hampshire,166...
1	Elizabeth Johnson	1360 Tracey Ports Apt. 419, Kyleport, Vermont,319...
2	Donald Stephens	19120 Fleming Manors, Prestonstad, Montana, 23495
3	Michael Vincent III	441 Olivia Creek, Jimmymouth, Georgia, 82991
4	Jasmine Zamora	4246 Chelsey Ford Apt. 310, Karamouth, Utah, 76...

Получить длины значений в каждой строке можно с помощью метода `str.len`. Например, длина значения "Frank Manning" из 0-й строки равна 13 символам:

```
In [42] customers["Name"].str.len().head()
```

```
Out [42] 0    13
         1    17
         2    15
         3    19
         4    14
         Name: Name, dtype: int64
```

Допустим, нам нужно выделить имя и фамилию каждого из покупателей в отдельные столбцы. Возможно, вы знакомы с методом `split` языка Python, разбивающим строковое значение по указанному разделителю. Он возвращает список, состоящий из всех подстрок, полученных в результате разбиения. В следующем примере мы разбиваем телефонный номер на список из трех строковых значений по разделителю `"-"`:

```
In [43] phone_number = "555-123-4567"
        phone_number.split("-")
```

```
Out [43] ['555', '123', '4567']
```

Метод `str.split` производит аналогичную операцию над каждой из строк объекта `Series` и возвращает объект `Series` со списками. Разделитель задается в первом параметре этого метода, `pat` (сокращение от `pattern` — «шаблон»). В примере ниже мы разбиваем значения в столбце `Name` по пробелам:

```
In [44] # Две строки ниже эквивалентны
        customers["Name"].str.split(pat = " ").head()
        customers["Name"].str.split(" ").head()
```

```
Out [44] 0      [Frank, Manning]
        1      [Elizabeth, Johnson]
        2      [Donald, Stephens]
        3      [Michael, Vincent, III]
        4      [Jasmine, Zamora]
        Name: Name, dtype: object
```

Теперь снова вызовем метод `str.len` для этого нового объекта `Series`, чтобы получить длину каждого из списков. Библиотека `pandas` динамически выбирает операцию на основе типа содержащихся в объекте `Series` данных:

```
In [45] customers["Name"].str.split(" ").str.len().head()
```

```
Out [45] 0      2
        1      2
        2      2
        3      3
        4      2
        Name: Name, dtype: int64
```

Небольшая проблема: вследствие таких суффиксов, как `MD` и `Jr`, некоторые имена состоят более чем из трех слов. Пример можно видеть на позиции с индексом 3: `Michael Vincent III`, имя которого библиотека `pandas` разбивает на список из трех элементов. Чтобы получить одинаковое количество элементов в списках, можно ограничить число разбиений. Если указать, что максимально возможно одно разбиение, библиотека `pandas` разобьет строковое значение по первому пробелу и на этом завершит обработку строкового значения. В результате

212 Часть II. Библиотека pandas на практике

получится объект `Series`, состоящий из двухэлементных списков. В каждом списке будут содержаться имя покупателя и отдельно все остальные символы, которые за ним следуют.

Вот как это выглядит на практике: в код для примера мы передадим аргумент 1 для параметра `n` метода `split`, задающий максимальное количество разбиений. Посмотрим, как библиотека `pandas` обработает при этом значение "Michael Vincent III" по индексу 3:

```
In [46] customers["Name"].str.split(pat = " ", n = 1).head()
```

```
Out [46] 0      [Frank, Manning]
         1      [Elizabeth, Johnson]
         2      [Donald, Stephens]
         3      [Michael, Vincent III]
         4      [Jasmine, Zamora]
         Name: Name, dtype: object
```

Теперь длины всех списков совпадают. Можно воспользоваться методом `str.get` для извлечения значений из списков строк на основе позиций индекса. Например, при индексе 0 извлекаются первые элементы всех списков, то есть имена покупателей:

```
In [47] customers["Name"].str.split(pat = " ", n = 1).str.get(0).head()
```

```
Out [47] 0      Frank
         1      Elizabeth
         2      Donald
         3      Michael
         4      Jasmine
         Name: Name, dtype: object
```

Для извлечения из списков фамилий можно указать в методе `get` позицию индекса 1:

```
In [48] customers["Name"].str.split(pat = " ", n = 1).str.get(1).head()
```

```
Out [48] 0      Manning
         1      Johnson
         2      Stephens
         3      Vincent III
         4      Zamora
         Name: Name, dtype: object
```

Метод `get` позволяет также указывать отрицательные аргументы. Аргумент -1 приводит к извлечению последнего элемента списка, вне зависимости от того, сколько в нем элементов. Следующий код дает тот же результат, что и предыдущий, и несколько удобнее в случаях, когда длины списков различаются:

```
In [49] customers["Name"].str.split(pat = " ", n = 1).str.get(-1).head()
```

```
Out [49] 0      Manning
          1      Johnson
          2      Stephens
          3  Vincent III
          4      Zamora
          Name: Name, dtype: object
```

Пока все идет как надо. Имя и фамилия извлечены в два отдельных объекта **Series** с помощью двух отдельных вызовов метода **get**. Не существует ли способа сделать то же самое за один вызов метода? Это было бы гораздо удобнее. К счастью, у метода **str.split** есть параметр **expand**, и, если указать для этого параметра аргумент **True**, метод вернет новый объект **DataFrame** вместо объекта **Series** со списками:

```
In [50] customers["Name"].str.split(
        pat = " ", n = 1, expand = True
    ).head()
```

```
Out [50]
```

```

           0      1
-----
0      Frank  Manning
1  Elizabeth  Johnson
2      Donald  Stephens
3   Michael  Vincent III
4    Jasmine   Zamora
```

Вот как! Сразу получен новый объект **DataFrame**! Но, поскольку мы не указали своих названий для столбцов, библиотека **pandas** по умолчанию задала по оси столбцов числовой индекс.

Осторожнее: если не ограничить количество разбиений с помощью параметра **n**, **pandas** поместит значения **None** в строках, где элементов недостаточно, вот так:

```
In [51] customers["Name"].str.split(pat = " ", expand = True).head()
```

```
Out [51]
```

```

           0      1      2
-----
0      Frank  Manning  None
1  Elizabeth  Johnson  None
2      Donald  Stephens  None
3   Michael  Vincent   III
4    Jasmine   Zamora  None
```

214 Часть II. Библиотека pandas на практике

Итак, мы выделили имена покупателей. Теперь присоединим новый объект `DataFrame` из двух столбцов к уже существующему объекту `DataFrame` с покупателями. Справа от знака равенства будет код с методом `split` для создания объекта `DataFrame`. Слева от знака равенства укажем список названий столбцов в квадратных скобках. Библиотека `pandas` присоединит эти столбцы к данным о покупателях. Это мы реализуем в примере ниже: добавляем два новых столбца, `First Name` и `Last Name`, и заполняем их данными из возвращаемого методом `split` объекта `DataFrame`:

```
In [52] customers[["First Name", "Last Name"]] = customers[
        "Name"
        ].str.split(pat = " ", n = 1, expand = True)
```

Взглянем на результат:

```
In [53] customers
```

```
Out [53]
```

	Name	Address	First Name	Last Name
0	Frank Manning	6461 Quinn Groves, E...	Frank	Manning
1	Elizabeth Johnson	1360 Tracey Ports Ap...	Elizabeth	Johnson
2	Donald Stephens	19120 Fleming Manors...	Donald	Stephens
3	Michael Vincent III	441 Olivia Creek, Ji...	Michael	Vincent III
4	Jasmine Zamora	4246 Chelsey Ford Ap...	Jasmine	Zamora
...
9956	Dana Browning	762 Andrew Views Apt...	Dana	Browning
9957	Amanda Anderson	44188 Day Crest Apt ...	Amanda	Anderson
9958	Eric Davis	73015 Michelle Squar...	Eric	Davis
9959	Taylor Hernandez	129 Keith Greens, Ha...	Taylor	Hernandez
9960	Sherry Nicholson	355 Griffin Valley, ...	Sherry	Nicholson

```
9961 rows x 4 columns
```

Замечательно, согласитесь! Мы выделили имена/фамилии покупателей в отдельные столбцы и теперь можем удалить исходный столбец `Name`. Сделать это можно, например вызвав метод `drop` нашего объекта `DataFrame` с покупателями. Передаем в его параметр `labels` название столбца, а в параметр `axis` — аргумент `"columns"`. Параметр `axis` нужен, чтобы `pandas` искала метку `Name` среди столбцов, а не строк:

```
In [54] customers = customers.drop(labels = "Name", axis = "columns")
```

Как вы помните, меняющие объекты операции не выводят результаты в блокноты Jupyter. Чтобы посмотреть результат, необходимо вывести объект `DataFrame` явным образом:

```
In [55] customers.head()
```

```
Out [55]
```

	Address	First Name	Last Name
0	6461 Quinn Groves, East Matthew, New Hampshire...	Frank	Manning
1	1360 Tracey Ports Apt. 419, Kyleport, Vermont...	Elizabeth	Johnson
2	19120 Fleming Manors, Prestonstad, Montana...	Donald	Stephens
3	441 Olivia Creek, Jimmymouth, Georgia...	Michael	Vincent III
4	4246 Chelsey Ford Apt. 310, Karamouth, Utah...	Jasmine	Zamora

Готово. Столбца Name больше нет, мы разбили его содержимое на два новых.

6.6. УПРАЖНЕНИЕ

И снова ловите счастливый шанс отработать на практике использование изложенных в этой главе идей!

6.6.1. Задача

Наш набор данных покупателей включает столбец **Address**. Адреса состоят из улицы, города, штата и почтового индекса. Ваша задача: разбить эти четыре значения по четырем новым столбцам **Street**, **City**, **State** и **Zip**, после чего удалить исходный столбец **Address**. Попробуйте решить эту задачу самостоятельно, а затем можете заглянуть в решение.

6.6.2. Решение

Первый шаг — разбить строковые значения с адресами по разделителю с помощью метода `split`. В качестве разделителя, похоже, подходит запятая:

```
In [56] customers["Address"].str.split(",").head()
```

```
Out [56] 0    [6461 Quinn Groves, East Matthew, New Hampsh...
          1    [1360 Tracey Ports Apt. 419, Kyleport, Vermo...
          2    [19120 Fleming Manors, Prestonstad, Montana,...
          3    [441 Olivia Creek, Jimmymouth, Georgia, 82991]
          4    [4246 Chelsey Ford Apt. 310, Karamouth, Utah...
          Name: Address, dtype: object
```

К сожалению, при подобном разбиении после запятых остаются пробелы. Можно очистить полученные значения дополнительно, с помощью методов наподобие

216 Часть II. Библиотека pandas на практике

`strip`, но существует лучшее решение. Если присмотреться, мы увидим, что части адреса разделены запятой и пробелом в совокупности. Следовательно, можно сразу передать методу `split` разделитель, состоящий из обоих этих символов:

```
In [57] customers["Address"].str.split(", ").head()

Out [57] 0    [6461 Quinn Groves, East Matthew, New Hampshir...
         1    [1360 Tracey Ports Apt. 419, Kyleport, Vermont...
         2    [19120 Fleming Manors, Prestonstad, Montana, 2...
         3    [441 Olivia Creek, Jimmymouth, Georgia, 82991]
         4    [4246 Chelsey Ford Apt. 310, Karamouth, Utah, ...
         Name: Address, dtype: object
```

Теперь никаких лишних пробелов в начале получившихся подстрок в списках нет.

По умолчанию метод `split` возвращает содержащий списки объект `Series`. Но если передать его параметру `expand` аргумент `True`, он вернет объект `DataFrame`:

```
In [58] customers["Address"].str.split(", ", expand = True).head()

Out [58]
```

	0	1	2	3
0	6461 Quinn Groves	East Matthew	New Hampshire	16656
1	1360 Tracey Ports Apt. 419	Kyleport	Vermont	31924
2	19120 Fleming Manors	Prestonstad	Montana	23495
3	441 Olivia Creek	Jimmymouth	Georgia	82991
4	4246 Chelsey Ford Apt. 310	Karamouth	Utah	76252

Осталось сделать еще буквально пару шагов. Добавим новый объект `DataFrame` из четырех столбцов к нашему существующему объекту `DataFrame` с данными о покупателях. Сначала опишем список с новыми названиями столбцов. На этот раз присвоим его переменной для большей удобочитаемости. А затем передадим его в квадратных скобках перед знаком равенства. Справа от знака равенства разместим вышеприведенный код для создания нового объекта `DataFrame`:

```
In [59] new_cols = ["Street", "City", "State", "Zip"]

        customers[new_cols] = customers["Address"].str.split(
            pat = ", ", expand = True
        )
```

Последнее, что осталось сделать, — удалить исходный столбец `Address`. Для этого прекрасно подойдет метод `drop`. Чтобы зафиксировать изменения в объекте `DataFrame`, не забудьте перезаписать объект `customers` возвращаемым объектом `DataFrame`:


```
In [60] customers.drop(labels = "Address", axis = "columns").head()
```

```
Out [60]
```

	First Name	Last Name	Street	City	State	Zip
0	Frank	Manning	6461 Quin...	East Matthew	New Hamps...	16656
1	Elizabeth	Johnson	1360 Trac...	Kyleport	Vermont	31924
2	Donald	Stephens	19120 Fle...	Prestonstad	Montana	23495
3	Michael	Vincent III	441 Olivi...	Jimmymouth	Georgia	82991
4	Jasmine	Zamora	4246 Chel...	Karamouth	Utah	76252

Можно также указать встроенное ключевое слово Python `del` перед нужным столбцом. Этот синтаксис позволяет изменить объект `DataFrame`:

```
In [61] del customers["Address"]
```

Давайте-ка взглянем, что получилось в итоге:

```
In [62] customers.tail()
```

```
Out [62]
```

	First Name	Last Name	Street	City	State	Zip
9956	Dana	Browning	762 Andrew ...	North Paul	New Mexico	28889
9957	Amanda	Anderson	44188 Day C...	Lake Marcia	Maine	37378
9958	Eric	Davis	73015 Miche...	Watsonville	West Virginia	03933
9959	Taylor	Hernandez	129 Keith G...	Haleyfurt	Oklahoma	98916
9960	Sherry	Nicholson	355 Griffin...	Davidtown	New Mexico	17581

Мы успешно извлекли содержимое столбца `Address` в четыре новых столбца. Поздравляю с выполнением упражнения!

6.7. ПРИМЕЧАНИЕ ОТНОСИТЕЛЬНО РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Любое обсуждение обработки текстовых данных было бы неполным без упоминания регулярных выражений (RegEx). *Регулярное выражение* (regular expression) — поисковый шаблон, предназначенный для поиска последовательности символов в строковом значении.

Регулярные выражения объявляются с помощью специального синтаксиса, включающего буквы и специальные символы. Например, комбинация `\d` соответствует любой цифре от 0 до 9. При помощи регулярных выражений можно описывать сложные поисковые шаблоны, искать символы в нижнем регистре,

в верхнем регистре, цифры, косые черты, пробельные символы, границы строковых значений и многое другое.

Пусть в большом строковом значении скрыт телефонный номер 555-555-5555. С помощью регулярных выражений можно описать алгоритм поиска для извлечения последовательности, состоящей из трех цифр одна за другой, дефиса, еще трех цифр, еще одного дефиса и еще четырех цифр. Именно подобная точность задания шаблонов и делает регулярные выражения таким мощным инструментом.

Вот небольшой пример работы этого синтаксиса. В приведенном ниже коде мы заменяем все вхождения четырех последовательных цифр на символ звездочки с помощью метода `replace`:

```
In [63] customers["Street"].head()

Out [63] 0          6461 Quinn Groves
        1    1360 Tracey Ports Apt. 419
        2      19120 Fleming Manors
        3          441 Olivia Creek
        4    4246 Chelsey Ford Apt. 310
        Name: Street, dtype: object

In [64] customers["Street"].str.replace(
        "\d{4,}", "*", regex = True
    ).head()

Out [64] 0          * Quinn Groves
        1    * Tracey Ports Apt. 419
        2          * Fleming Manors
        3          441 Olivia Creek
        4    * Chelsey Ford Apt. 310
        Name: Street, dtype: object
```

Регулярные выражения — очень узкий технический вопрос, подчас требующий от человека специальной подготовки. Нюансам работы с регулярными выражениями посвящены целые книги. Пока что вам достаточно знать, что в большинстве методов для работы со строковыми значениями библиотеки pandas поддерживаются аргументы RegEx. Более подробную информацию об этом можно найти в приложении Д.

РЕЗЮМЕ

- Атрибут `str` содержит объект `StringMethods`, включающий методы для операций над строковыми значениями объектов `Series`.
- Семейство методов `strip` позволяет удалить пробелы в начале, в конце строкового значения или с обеих его сторон.

- Менять регистр букв в строковых значениях можно с помощью методов `upper`, `lower`, `capitalize` и `title`.
- С помощью метода `contains` можно проверить, входит ли определенная подстрока в строковое значение.
- Метод `startswith` ищет подстроку в начале строкового значения.
- Дополняющий его метод `endswith` ищет подстроку в конце строкового значения.
- Метод `split` разбивает строковое значение на список по заданному разделителю. С его помощью можно разбить текст из столбца объекта `DataFrame` на несколько объектов `Series`.



Мультииндексные объекты *DataFrame*

В этой главе

- ✓ Создание объекта `MultiIndex`.
- ✓ Извлечение строк и столбцов из мультииндексного объекта `DataFrame`.
- ✓ Поперечные срезы мультииндексных `DataFrame`.
- ✓ Перестановка уровней объекта `MultiIndex` местами.

До сих пор в нашем путешествии по библиотеке `pandas` мы обсуждали только одномерные объекты `Series` и двумерные объекты `DataFrame`. Размерность исчисляется по числу координат, требуемых для извлечения значения из структуры данных. Для ссылки на значение в объекте `Series` достаточно одной метки или одной позиции индекса. Для ссылки на значение в объекте `DataFrame` необходимо две координаты: метка/индекс для строк и метка/индекс для столбцов. Может ли размерность превышать два? Безусловно! Библиотека `pandas` поддерживает наборы данных произвольной размерности благодаря классу `MultiIndex`.

`MultiIndex` — это объект для индекса, включающий несколько уровней. Каждый уровень содержит значение индекса для строки. `MultiIndex` наиболее удобен, когда идентификатор для строки данных состоит из некоего сочетания значений.

Например, возьмем набор данных на рис. 7.1, содержащий курсы акций на различные даты.

Stock	Date	Price
MSFT	02/08/2021	793.60
MSFT	02/09/2021	1,408.38
GOOG	02/08/2021	565.81
GOOG	02/09/2021	17.62

Рис. 7.1. Пример набора данных со столбцами Stock, Date и Price

Пусть нам нужен уникальный идентификатор для каждого из курсов акций. По отдельности ни названия акций, ни даты недостаточно, но сочетание обоих этих значений отлично подходит в качестве идентификатора. Акции MSFT встречаются дважды, дата 02/08/2021 встречается дважды, но их сочетание, одновременное присутствие — только один раз. Для этого набора данных отлично подойдет мультииндекс со значениями столбцов Stock и Date.

Объекты MultiIndex прекрасно подходят для *иерархических* данных — данных, в которых значения одного из столбцов являются подкатегорией значений другого столбца. Рассмотрим набор данных на рис. 7.2.

Group	Item	Calories
Fruit	Apple	95
Fruit	Banana	105
Vegetable	Broccoli	50
Vegetable	Tomato	22

Рис. 7.2. Пример набора данных со столбцами Group, Item и Calories

Значения столбца Item — подкатегории значений столбца Group. Яблоко (Apple) — вид фруктов (Fruit), а брокколи (Broccoli) — вид овощей (Vegetable). Следовательно, столбцы Group, Item вместе могут служить мультииндексом.

Мультииндексы — довольно непростая в понимании и использовании возможность библиотеки pandas, но ее изучение себя оправдывает. Возможность использования нескольких уровней индекса значительно повышает гибкость срезов наборов данных.

7.1. ОБЪЕКТ MULTIINDEX

Откроем новый блокнот Jupyter, импортируем библиотеку pandas и присвоим ей псевдоним pd:

```
In [1] import pandas as pd
```

Ради простоты начнем с создания объекта `MultiIndex`, а позже, в разделе 7.2, опробуем все освоенные идеи уже на импортированном наборе данных.

Помните встроенный объект языка Python — кортеж? Кортеж — это неизменяемая структура данных для хранения упорядоченной последовательности значений. Кортежи, по сути, представляют собой списки, которые нельзя модифицировать после создания. Подробнее об этой структуре данных можно прочитать в приложении Б.

Допустим, нам надо смоделировать адрес улицы. Такой адрес обычно включает название улицы, город, штат и почтовый индекс. Эти четыре элемента можно хранить в кортеже:

```
In [2] address = ("8809 Flair Square", "Toddside", "IL", "37206")
        address
```

```
Out [2] ('8809 Underwood Squares', 'Toddside', 'IL', '37206')
```

В индексах объектов `Series` и `DataFrame` могут содержаться различные типы данных: строковые значения, числа, метки даты/времени и многое другое. Но для каждой позиции индекса в этих объектах может содержаться лишь одно значение или одна метка для одной строки. У кортежей таких ограничений нет.

А если собрать несколько кортежей в список? Такой список будет выглядеть примерно следующим образом:

```
In [3] addresses = [
        ("8809 Flair Square", "Toddside", "IL", "37206"),
        ("9901 Austin Street", "Toddside", "IL", "37206"),
        ("905 Hogan Quarter", "Franklin", "IL", "37206"),
    ]
```

А теперь представьте себе эти кортежи в роли меток индекса объекта `DataFrame`. Надеюсь, идея не слишком заумная. Все операции остаются прежними. Так же как и раньше, можно будет ссылаться на строки по метке индекса, но метки индекса будут представлять собой контейнеры, содержащие несколько элементов. Объекты `MultiIndex` для начала очень удобно представлять себе в виде индексов, каждая метка в которых может содержать несколько элементов данных.

Объекты `MultiIndex` можно создавать отдельно от объектов `Series` или `DataFrame`. Класс `MultiIndex` доступен в виде атрибута верхнего уровня библиотеки pandas и включает метод `from_tuples` для создания объекта `MultiIndex` на основе списка

кортежей. Метод класса (class method) — метод, при вызове которого ссылаются на класс, а не на экземпляр. В следующем примере мы вызываем метод класса `from_tuples` и передаем ему список `addresses`:

```
In [4] # Две строки ниже эквивалентны
      pd.MultiIndex.from_tuples(addresses)
      pd.MultiIndex.from_tuples(tuples = addresses)

Out [4] MultiIndex([( '8809 Flair Square',    'Toddside', 'IL', '37206'),
                    ('9901 Austin Street',    'Toddside', 'IL', '37206'),
                    ( '905 Hogan Quarter',    'Franklin', 'IL', '37206')],
                  )
```

Наш первый объект `MultiIndex` содержит три кортежа по четыре элемента каждый. Все элементы кортежей организованы одинаково:

- первое значение — адрес;
- второе значение — город;
- третье значение — штат;
- четвертое значение — почтовый индекс.

В терминологии `pandas` набор значений кортежа, расположенных на одной позиции индекса, образует *уровень* (level) объекта `MultiIndex`. В предыдущем примере первый уровень объекта `MultiIndex` состоит из значений "8809 Flair Square", "9901 Austin Street" и "905 Hogan Quarter". Аналогично второй уровень объекта `MultiIndex` состоит из "Toddside", "Toddside" и "Franklin".

Можно задать названия для уровней объекта `MultiIndex`, передав список названий в качестве аргумента параметра `names` метода `from_tuples`. В данном случае мы присваиваем уровням названия "Street", "City", "State" и "Zip":

```
In [5] row_index = pd.MultiIndex.from_tuples(
      tuples = addresses,
      names = ["Street", "City", "State", "Zip"]
    )

      row_index

Out [5] MultiIndex([( '8809 Flair Square',    'Toddside', 'IL', '37206'),
                    ('9901 Austin Street',    'Toddside', 'IL', '37206'),
                    ( '905 Hogan Quarter',    'Franklin', 'IL', '37206')],
                  names=['Street', 'City', 'State', 'Zip'])
```

Подытожим: объект `MultiIndex` — контейнер, метки которого состоят из нескольких значений. Уровень состоит из значений, находящихся на одной позиции в метках.

Теперь присоединим полученный объект `MultiIndex` к объекту `DataFrame`. Простейший способ — воспользоваться параметром `index` конструктора объекта

224 Часть II. Библиотека pandas на практике

DataFrame. В предыдущих главах мы передавали в этот параметр список строковых значений, но можно передать и любой допустимый объект индекса. Давайте передадим туда объект **MultiIndex**, который присвоили выше переменной **row_index**. Поскольку объект **MultiIndex** содержит три кортежа (или, что эквивалентно, три метки), необходимо передать три строки данных:

```
In [6] data = [
    ["A", "B+"],
    ["C+", "C"],
    ["D-", "A"],
]

columns = ["Schools", "Cost of Living"]

area_grades = pd.DataFrame(
    data = data, index = row_index, columns = columns
)

area_grades
```

Out [6]

Street	City	State	Zip	Schools	Cost of Living
8809 Flair Square	Toddside	IL	37206	A	B+
9901 Austin Street	Toddside	IL	37206	C+	C
905 Hogan Quarter	Franklin	IL	37206	D-	A

Получился объект **MultiIndex** с мультииндексом по оси строк. Метка каждой строки содержит четыре значения: улицу, город, штат и почтовый индекс.

Займемся теперь осью столбцов. Библиотека **pandas** хранит заголовки столбцов **DataFrame** также в объекте индекса. Обращаться к этому индексу можно через атрибут **columns**:

```
In [7] area_grades.columns
```

Out [7] Index(['Schools', 'Cost of Living'], dtype='object')

Пока в результате наших операций **pandas** хранит эти два названия столбцов в одноуровневом объекте **Index**. Создадим второй объект **MultiIndex** и присоединим его к оси столбцов. Итак, в примере ниже мы снова вызываем метод класса **from_tuples**, передавая ему список четырех кортежей, каждый из которых содержит два строковых значения:

```
In [8] column_index = pd.MultiIndex.from_tuples(
    [
        ("Culture", "Restaurants"),
        ("Culture", "Museums"),
```



```

        ("Services", "Police"),
        ("Services", "Schools"),
    ]
)

column_index

Out [8] MultiIndex([( 'Culture', 'Restaurants'),
                    ( 'Culture',   'Museums'),
                    ('Services',   'Police'),
                    ('Services',   'Schools')]),
)
```

Теперь присоединим оба наших мультииндекса к объекту `DataFrame`. Объект `MultiIndex` для оси строк (`row_index`) требует, чтобы набор данных содержал три строки. Объект `MultiIndex` для оси столбцов (`column_index`) требует, чтобы набор данных содержал четыре столбца. Таким образом, форма нашего набора данных должна быть 3×4 . Создадим подобную структуру данных. В следующем примере объявим список из трех списков. Каждый из вложенных списков содержит четыре строковых значения:

```

In [9] data = [
    ["C-", "B+", "B-", "A"],
    ["D+", "C", "A", "C+"],
    ["A-", "A", "D+", "F"]
]
```

Теперь можно собрать все воедино и создать объект `DataFrame` с мультииндексами как по оси строк, так и по оси столбцов. Сделаем так: передадим соответствующие первые `MultiIndex` в параметры `index` и `columns` конструктора `DataFrame`:

```

In [10] pd.DataFrame(
    data = data, index = row_index, columns = column_index
)
```

Out [10]

Street	City	State	Zip	Culture		Services	
				Restaurants	Museums	Police	Schools
8809 Flai...	Toddside	IL	37206	C-	B+	B-	A
9901 Aust...	Toddside	IL	37206	D+	C	A	C+
905 Hogan...	Franklin	IL	37206	A-	A	D+	F

Ура! Мы успешно создали объект `DataFrame` с четырехуровневым мультииндексом по строкам и двухуровневым мультииндексом по столбцам. Мультииндекс — это индекс, состоящий из нескольких уровней, или слоев. Метки индекса в них состоят из нескольких компонентов. Вот и все.

7.2. ОБЪЕКТЫ DATAFRAME С МУЛЬТИИНДЕКСАМИ

Возьмем теперь набор данных большего размера. Набор данных `neighborhoods.csv` аналогичен созданному нами в разделе 7.1; он включает ~250 вымышленных адресов в городах США. Все адреса классифицированы по четырем позициям-группам социальных институтов: `Restaurants` (рестораны), `Museums` (музеи), `Police` (полиция) и `Schools` (школы).

Вот несколько первых строк из исходного CSV-файла. В CSV-файле соседние значения в строке данных разделяются запятыми. Таким образом, последовательные запятые, между которыми ничего нет, указывают на пропущенные значения:

```
,,,Culture,Culture,Services,Services
,,,Restaurants,Museums,Police,Schools
State,City,Street,,,,
MO,Fisherborough,244 Tracy View,C+,F,D-,A+
```

Как библиотека `pandas` импортирует данные из этого CSV-файла? Выясним это с помощью функции `read_csv`:

```
In [11] neighborhoods = pd.read_csv("neighborhoods.csv")
        neighborhoods.head()
```

Out [11]

	Unnamed: 0	Unnamed: 1	Unnamed: 2	Culture	Culture.1	Services	Services.1
0	NaN	NaN	NaN	Restau...	Museums	Police	Schools
1	State	City	Street	NaN	NaN	NaN	NaN
2	MO	Fisher...	244 Tr...	C+	F	D-	A+
3	SD	Port C...	446 Cy...	C-	B	B	D+
4	WV	Jimene...	432 Jo...	A	A+	F	B

Что-то здесь не так! Во-первых, три столбца — `Unnamed` (Не поименованы) и их названия оканчиваются различными цифрами. При импорте CSV-файла библиотека `pandas` предполагает, что первая строка файла содержит названия столбцов — заголовки. Если на соответствующей позиции этой строки значение отсутствует, библиотека `pandas` присваивает столбцу название `Unnamed`. В то же время `pandas` стремится избежать дублирования названий столбцов. И чтобы различать разные столбцы с отсутствующими заголовками, библиотека `pandas` добавляет к каждому из названий числовой индекс. Так возникли три столбца: `Unnamed: 0`, `Unnamed: 1` и `Unnamed: 2`.

Та же проблема наблюдается и у четырех столбцов справа от них. Обратите внимание, что библиотека `pandas` называет `Culture` столбец на позиции с индексом 3

и **Culture 1** — следующий. У двух ячеек заголовков в CSV-файле одно и то же значение "Culture", а за ними следуют две ячейки со значением "Services".

К сожалению, на этом наши проблемы не заканчиваются. В каждом из трех первых значений в строке 0 содержится **NaN**. В строке 1 значения **NaN** — в последних четырех столбцах. Проблема в том, что CSV-файл моделирует многоуровневый индекс по строкам и многоуровневый индекс по столбцам, но аргументы по умолчанию функции `read_csv` их не распознают. К счастью, эту проблему можно решить путем внесения изменений в аргументы нескольких параметров функции `read_csv`.

Первым делом необходимо сообщить библиотеке `pandas`, что три крайних слева столбца должны играть роль индекса **DataFrame**. Для этого можно передать в параметр `index_col` список чисел, каждое из которых отражает индекс (или числовую позицию) столбца, который должен входить в индекс объекта **DataFrame**. Этот индекс начинается с 0. Таким образом, позиции индекса первых трех столбцов (непоименованных) будут 0, 1 и 2. При передаче в параметр `index_col` списка из нескольких значений библиотека `pandas` автоматически создает **MultiIndex** для объекта **DataFrame**:

```
In [12] neighborhoods = pd.read_csv(
        "neighborhoods.csv",
        index_col = [0, 1, 2]
    )

    neighborhoods.head()
```

Out [12]

			Culture	Culture.1	Services	Services.1
NaN	NaN	NaN	Restaurants	Museums	Police	Schools
State	City	Street	NaN	NaN	NaN	NaN
MO	Fisherbor...	244 Tracy...	C+	F	D-	A+
SD	Port Curt...	446 Cynth...	C-	B	B	D+
WV	Jimenezview	432 John ...	A	A+	F	B

Половина пути пройдена. Далее необходимо сообщить библиотеке `pandas`, какие строки набора данных должны использоваться для заголовков объекта **DataFrame**. Функция `read_csv` предполагает, что заголовки могут содержаться только в первой строке. Но в этом наборе данных заголовки содержатся в первых двух строках. Можно задать свои заголовки объекта **DataFrame** с помощью параметра `header` функции `read_csv`. Он будет принимать список целых чисел, соответствующих *строкам*, которые библиотека `pandas` будет использовать в качестве заголовков столбцов. Если передать список из нескольких элементов, библиотека `pandas` задаст мультииндекс для столбцов. В следующем примере мы делаем заголовками столбцов первые две строки (индексы 0 и 1):

```
In [13] neighborhoods = pd.read_csv(
        "neighborhoods.csv",
```

```
        index_col = [0, 1, 2],
        header = [0, 1]
    )
    neighborhoods.head()
```

Out [13]

State	City	Street	Culture		Services	
			Restaurants	Museums	Police	Schools
MO	Fisherborough	244 Tracy View	C+	F	D-	A+
SD	Port Curtisv...	446 Cynthia ...	C-	B	B	D+
WV	Jimenezview	432 John Common	A	A+	F	B
AK	Stevenshire	238 Andrew Rue	D-	A	A-	A-
ND	New Joshuaport	877 Walter Neck	D+	C-	B	B

С этим набором уже можно работать!

Как упоминалось ранее, в наборе данных четыре показателя социальных институтов: *Restaurants*, *Museums*, *Police* и *Schools* сгруппированы по двум категориям (*Culture* и *Services*). Когда общие родительские категории охватывают маленькие дочерние категории, создание *MultiIndex* — оптимальный способ для быстрых срезов данных.

Вызовем несколько уже знакомых нам методов, чтобы посмотреть, как меняются выводимые результаты в случае *DataFrame* с *MultiIndex*. Для начала нам подойдет метод *info*:

```
In [14] neighborhoods.info()
```

Out [14]

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 251 entries, ('MO', 'Fisherborough', '244 Tracy View') to ('NE',
'South Kennethmouth', '346 Wallace Pass')
Data columns (total 4 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   (Culture, Restaurants)                251 non-null    object
1   (Culture, Museums)                   251 non-null    object
2   (Services, Police)                   251 non-null    object
3   (Services, Schools)                  251 non-null    object
dtypes: object(4)
memory use: 27.2+ KB
```

Обратите внимание, что pandas выводит названия каждого из столбцов в виде двухэлементного кортежа, например *(Culture, Restaurants)*. Аналогично библиотека pandas хранит метки строк в виде трехэлементных кортежей, например *('MO', 'Fisherborough', '244 Tracy View')*.

Обращаться к строкам объекта `MultiIndex` можно с помощью обычного атрибута `index`. В результатах можно видеть кортежи со значениями каждой из строк:

```
In [15] neighborhoods.index
```

```
Out [15] MultiIndex([
      ('MO',      'Fisherborough',      '244 Tracy View'),
      ('SD',      'Port Curtisville',    '446 Cynthia Inlet'),
      ('WV',      'Jimenezview',        '432 John Common'),
      ('AK',      'Stevenshire',        '238 Andrew Rue'),
      ('ND',      'New Joshuaport',      '877 Walter Neck'),
      ('ID',      'Wellsville',          '696 Weber Stravenue'),
      ('TN',      'Jodiburgh',          '285 Justin Corners'),
      ('DC',      'Lake Christopher',    '607 Montoya Harbors'),
      ('OH',      'Port Mike',          '041 Michael Neck'),
      ('ND',      'Hardyburgh',          '550 Gilmore Mountains'),
      ...
      ('AK',      'South Nicholasshire', '114 Jones Garden'),
      ('IA',      'Port Willieport',    '320 Jennifer Mission'),
      ('ME',      'Port Linda',         '692 Hill Glens'),
      ('KS',      'Kaylamouth',         '483 Freeman Via'),
      ('WA',      'Port Shawnfort',     '691 Winters Bridge'),
      ('MI',      'North Matthew',      '055 Clayton Isle'),
      ('MT',      'Chadton',            '601 Richards Road'),
      ('SC',      'Diazmouth',          '385 Robin Harbors'),
      ('VA',      'Laurentown',         '255 Gonzalez Land'),
      ('NE',      'South Kennethmouth', '346 Wallace Pass')],
      names=['State', 'City', 'Street'], length=251)
```

Обращаться к объекту `MultiIndex` для столбцов можно через атрибут `columns`, в котором для хранения вложенных меток столбцов также используются кортежи:

```
In [16] neighborhoods.columns
```

```
Out [16] MultiIndex([( 'Culture', 'Restaurants'),
      ( 'Culture',      'Museums'),
      ('Services',      'Police'),
      ('Services',      'Schools')],
      )
```

«За кулисами» библиотека `pandas` составляет `MultiIndex` из нескольких объектов `Index`. При импорте набора данных библиотека присваивает название каждому объекту `Index` в соответствии с заголовком CSV-файла. Обращаться к списку названий индексов можно с помощью атрибута `names` объекта `MultiIndex`. Названия трех столбцов CSV-файла, ставших нашим индексом, — `State`, `City` и `Street`:

```
In [17] neighborhoods.index.names
```

```
Out [17] FrozenList(['State', 'City', 'Street'])
```

Библиотека pandas упорядочивает вложенные уровни объекта `MultiIndex`. В текущем объекте `DataFrame neighborhoods`:

- позиция индекса уровня `State` — 0;
- позиция индекса уровня `City` — 1;
- позиция индекса уровня `Street` — 2.

Метод `get_level_values` извлекает объект `Index`, соответствующий заданному уровню объекта `MultiIndex`. В первый и единственный параметр этого метода, `level`, можно передать либо позицию индекса уровня, либо название уровня:

```
In [18] # Две строки ниже эквивалентны
        neighborhoods.index.get_level_values(1)
        neighborhoods.index.get_level_values("City")

Out [18] Index(['Fisherborough', 'Port Curtisville', 'Jimenezview',
               'Stevenshire', 'New Joshuaport', 'Wellsville', 'Jodiburgh',
               'Lake Christopher', 'Port Mike', 'Hardyburgh',
               ...
               'South Nicholasshire', 'Port Willieport', 'Port Linda',
               'Kaylamouth', 'Port Shawnfort', 'North Matthew', 'Chadton',
               'Diazmouth', 'Laurentown', 'South Kennethmouth'],
              dtype='object', name='City', length=251)
```

Уровни столбцов объекта `MultiIndex` не поименованы, поскольку в CSV-файле для них не указаны названия:

```
In [19] neighborhoods.columns.names

Out [19] FrozenList([None, None])
```

Исправим это. Обращаться к объекту `MultiIndex` для столбцов можно через атрибут `columns`. И еще можно присвоить атрибуту `names` объекта `MultiIndex` новый список названий столбцов. Названия `"Category"` и `"Subcategory"` прекрасно подходят для этой цели:

```
In [20] neighborhoods.columns.names = ["Category", "Subcategory"]
        neighborhoods.columns.names

Out [20] FrozenList(['Category', 'Subcategory'])
```

Названия уровней выводятся слева от заголовков столбцов. Вызовем метод `head` и посмотрим, что поменялось:

```
In [21] neighborhoods.head(3)

Out [21]
```

Category			Culture		Services	
Subcategory		Street	Restaurants	Museums	Police	Schools
State	City					
MO	Fisherbor...	244 Tracy...	C+	F	D-	A+
SD	Port Curt...	446 Cynth...	C-	B	B	D+
WV	Jimenezview	432 John ...	A	A+	F	B

Теперь, присвоив названия уровням, мы можем извлечь с помощью метода `get_level_values` любой объект `Index` из объекта `MultiIndex` для столбцов. Напомню, что передать в этот метод можно либо позицию индекса уровня, либо его название:

```
In [22] # Две строки ниже эквивалентны
neighborhoods.columns.get_level_values(0)
neighborhoods.columns.get_level_values("Category")

Out [22] Index(['Culture', 'Culture', 'Services', 'Services'],
dtype='object', name='Category')
```

Объект `MultiIndex` переносится и на новые объекты, полученные из этого набора данных. В зависимости от операции индекс может переходить на другую ось. Возьмем метод `nunique` объектов `DataFrame`, возвращающий объект `Series` с количеством уникальных значений в каждом столбце. Если вызвать метод `nunique` для `neighborhoods`, мультииндекс столбцов объекта `DataFrame` сменит ось и будет играть в итоговом объекте `Series` роль мультииндекса для строк:

```
In [23] neighborhoods.head(1)
```

```
Out [23]
```

Category			Culture		Services	
Subcategory		Street	Restaurants	Museums	Police	Schools
State	City					
AK	Rowlandchester	386 Rebecca ...		C-	A-	A+
						C

```
In [24] neighborhoods.nunique()
```

```
Out [24] Culture    Restaurants    13
           Museums      13
           Services  Police      13
           Schools     Schools    13
dtype: int64
```

Объект `Series` мультииндекса содержит количество уникальных значений, найденных библиотекой `pandas` в каждом из четырех столбцов. В данном случае значения равны, поскольку в каждом из четырех столбцов содержатся 13 возможных показателей (от A+ до F).

7.3. СОРТИРОВКА МУЛЬТИИНДЕКСОВ

Библиотека pandas может найти значение в упорядоченном наборе намного быстрее, чем в неупорядоченном. Хорошая аналогия — поиск слова в словаре. Когда слова расположены в алфавитном порядке, а не в случайной последовательности, найти конкретное слово намного проще. Следовательно, перед выбором строк и столбцов из объекта `DataFrame` имеет смысл отсортировать индекс.

В главе 4 вы познакомились с методом `sort_index`, предназначенным для сортировки объектов `DataFrame`. При вызове этого метода для мультииндексного объекта `DataFrame` библиотека pandas сортирует все уровни в порядке возрастания, обрабатывая их от самого внешнего к самому внутреннему. В следующем примере библиотека pandas сортирует сначала значения уровня `State`, затем уровня `City` и, наконец, значения уровня `Street`:

In [25] `neighborhoods.sort_index()`

Out [25]

Category				Culture		Services	
Subcategory				Restaurants	Museums	Police	Schools
State	City	Street					
AK	Rowlandchester	386	Rebecca ...	C-	A-	A+	C
	Scottstad	082	Leblanc ...	D	C-	D	B+
		114	Jones Ga...	D-	D-	D	D
	Stevenshire	238	Andrew Rue	D-	A	A-	A-
AL	Clarkland	430	Douglas ...	A	F	C+	B+
...							
WY	Lake Nicole	754	Weaver T...	B	D-	B	D
		933	Jennifer...	C	A+	A-	C
	Martintown	013	Bell Mills	C-	D	A-	B-
	Port Jason	624	Faulkner...	A-	F	C+	C+
	Reneeshire	717	Patel Sq...	B	B+	D	A

251 rows × 4 columns

Давайте убедимся, что в полученных результатах вам все понятно. Во-первых, библиотека pandas обрабатывает уровень `State`, располагая значение "AK" перед "AL". Затем, внутри штата AK, pandas сортирует города, располагая "Rowlandchester" перед "Scottstad". После чего применяет ту же логику к последнему уровню, `Street`.

У метода `sort_index` есть параметр `ascending`, в который можно передать булев аргумент для согласованной сортировки всех уровней мультииндекса. Приведу пример передачи аргумента `False`. Библиотека pandas сортирует значения уровня `State` в обратном алфавитном порядке, затем значения уровня `City`

в обратном алфавитном порядке и, наконец, значения уровня `Street` в обратном алфавитном порядке:

```
In [26] neighborhoods.sort_index(ascending = False).head()
```

```
Out [26]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
WY	Reneeshire	717 Patel Sq...	B	B+	D	A
	Port Jason	624 Faulkner...	A-	F	C+	C+
	Martintown	013 Bell Mills	C-	D	A-	B-
	Lake Nicole	933 Jennifer...	C	A+	A-	C
		754 Weaver T...	B	D-	B	D

А теперь нам нужно, скажем, задать различный порядок сортировки для разных уровней. Для этого можно передать в параметр `ascending` список булевых значений, каждое из которых задает порядок сортировки для следующего уровня мультииндекса в направлении от самого внешнего к самому внутреннему. Аргумент `[True, False, True]`, например, приведет к сортировке уровня `State` в порядке возрастания, уровня `City` в порядке убывания и уровня `Street` в порядке возрастания:

```
In [27] neighborhoods.sort_index(ascending = [True, False, True]).head()
```

```
Out [27]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
AK	Stevenshire	238 Andrew Rue	D-	A	A-	A-
	Scottstad	082 Leblanc ...	D	C-	D	B+
		114 Jones Ga...	D-	D-	D	D
	Rowlandchester	386 Rebecca ...	C-	A-	A+	C
AL	Vegaside	191 Mindy Me...	B+	A-	A+	D+

Можно также отсортировать отдельный уровень мультииндекса. Допустим, нам нужно отсортировать строки по значениям на втором уровне мультииндекса, `City`. Для этого можно передать позицию индекса данного уровня или его название в параметр `level` метода `sort_index`. Библиотека `pandas` проигнорирует остальные уровни при сортировке:

```
In [28] # Две строки ниже эквивалентны
         neighborhoods.sort_index(level = 1)
         neighborhoods.sort_index(level = "City")
```

```
Out [28]
```

Category			Culture		Services	
Subcategory		Street	Restaurants	Museums	Police	Schools
State	City					
AR	Allisonland	124 Diaz Brooks	C-	A+	F	C+
GA	Amyburgh	941 Brian Ex...	B	B	D-	C+
IA	Amyburgh	163 Heather ...	F	D	A+	A-
ID	Andrewshire	952 Ellis Drive	C+	A-	C+	A
UT	Baileyfort	919 Stewart ...	D+	C+	A	C
...
NC	West Scott	348 Jack Branch	A-	D-	A-	A
SD	West Scott	139 Hardy Vista	C+	A-	D+	B-
IN	Wilsonborough	066 Carr Road	A+	C-	B	F
NC	Wilsonshire	871 Christop...	B+	B	D+	F
NV	Wilsonshire	542 Jessica ...	A	A+	C-	C+

251 rows × 4 columns

В параметр `level` также можно передать список уровней. В примере ниже мы сортируем сначала значения уровня `City`, а затем значения уровня `Street`. Значения уровня `State` при этой сортировке вообще никак не затрагиваются:

```
In [29] # Две строки ниже эквивалентны
        neighborhoods.sort_index(level = [1, 2]).head()
        neighborhoods.sort_index(level = ["City", "Street"]).head()
```

Out [29]

Category			Culture		Services	
Subcategory		Street	Restaurants	Museums	Police	Schools
State	City					
AR	Allisonland	124 Diaz Brooks	C-	A+	F	C+
IA	Amyburgh	163 Heather ...	F	D	A+	A-
GA	Amyburgh	941 Brian Ex...	B	B	D-	C+
ID	Andrewshire	952 Ellis Drive	C+	A-	C+	A
VT	Baileyfort	831 Norma Cove	B	D+	A+	D+

Можно также сочетать параметры `ascending` и `level`. Обратите внимание, что в последнем примере pandas отсортировала два значения уровня `Street` для `Amyburgh` ("163 Heather Neck" и "941 Brian Expressway") в порядке возрастания (алфавитном порядке). В следующем примере мы сортируем уровень `City` в порядке возрастания, а уровень `Street` — в порядке убывания, таким образом меняя местами позиции двух значений `Street` для `Amyburgh`:

```
In [30] neighborhoods.sort_index(
        level = ["City", "Street"], ascending = [True, False]
    ).head()
```

Out [30]

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
AR	Allisonland	124 Diaz Brooks	C-	A+	F	C+
GA	Amyburgh	941 Brian Ex...	B	B	D-	C+
IA	Amyburgh	163 Heather ...	F	D	A+	A-
ID	Andrewshire	952 Ellis Drive	C+	A-	C+	A
UT	Baileyfort	919 Stewart ...	D+	C+	A	C

Можно провести сортировку мультииндекса для столбцов, указав параметр `axis` метода `sort_index`. Аргумент по умолчанию этого параметра равен 0, что соответствует индексу по строкам. Для сортировки столбцов можно передать либо число 1, либо строковое значение "columns". В примере ниже библиотека `pandas` сортирует сначала уровень `Category`, а затем уровень `Subcategory`. Значение `Culture` при этом предшествует значению `Services`. А в рамках уровня `Culture` значение `Museums` предшествует значению `Restaurants`. В рамках уровня `Services` значение `Police` предшествует значению `Schools`:

```
In [31] # Две строки ниже эквивалентны
neighborhoods.sort_index(axis = 1).head(3)
neighborhoods.sort_index(axis = "columns").head(3)
```

Out [31]

Category			Culture		Services	
Subcategory			Museums	Restaurants	Police	Schools
State	City	Street				
MO	Fisherborough	244 Tracy View	F	C+	D-	A+
SD	Port Curtisv...	446 Cynthia ...	B	C-	B	D+
WV	Jimenezview	432 John Common	A+	A	F	B

Есть возможность сочетать параметры `ascending` и `level` с параметром `axis` для еще более гибкого задания порядка сортировки столбцов. Отсортируем значения уровня `Subcategory` в порядке убывания. Значения уровня `Category` на сортировку не влияют, библиотека `pandas` их игнорирует. Визуально обратный алфавитный порядок подкатегорий ("Schools", "Restaurants", "Police" и "Museums") приводит к разрыву группы `Category`. Поэтому заголовки столбцов `Services` и `Culture` выводятся несколько раз:

```
In [32] neighborhoods.sort_index(
        axis = 1, level = "Subcategory", ascending = False
    ).head(3)
```

Out [32]

Category	Subcategory	State	City	Street	Services	Culture	Services	Culture
					Schools	Restaurants	Police	Museums
MO	Fisherborough	244	Tracy View		A+	C+	D-	F
SD	Port Curtisv...	446	Cynthia ...		D+	C-	B	B
WV	Jimenezview	432	John Common		B	A	F	A+

Позже, в разделе 7.4, вы научитесь извлекать строки и столбцы из мультииндексного объекта `DataFrame` с помощью уже знакомых вам атрибутов-получателей `loc` и `iloc`. Как уже упоминалось ранее, желательно отсортировать индекс, прежде чем искать какие-либо строки в наборе. Отсортируем уровни объекта `MultiIndex` в порядке возрастания и перезапишем объект `DataFrame` `neighborhoods`:

```
In [33] neighborhoods = neighborhoods.sort_index(ascending = True)
```

Вот что получается в результате:

```
In [34] neighborhoods.head(3)
```

```
Out [34]
```

Category	Subcategory	State	City	Street	Culture	Services
					Restaurants	Museums
AK	Rowlandchester	386	Rebecca ...		C-	A-
	Scottstad	082	Leblanc ...		D	C-
		114	Jones Ga...		D-	D-
					A+	D
					B+	D

Выглядит неплохо. Мы отсортировали все уровни объекта `MultiIndex` и можем переходить к следующему пункту программы.

7.4. ВЫБОРКА ДАННЫХ С ПОМОЩЬЮ МУЛЬТИИНДЕКСОВ

Извлечение строк и столбцов многоуровневых объектов `DataFrame` — более сложная задача, чем уже опробованное нами извлечение одноуровневых наборов. Основной вопрос, на который нужно ответить перед написанием какого-либо кода, — что именно вы хотите извлечь.

В главе 4 вы научились использовать синтаксис с квадратными скобками для выборки столбцов из объекта `DataFrame`. Напомню основное. Например, вот такой код создает объект `DataFrame` с двумя строками и двумя столбцами:

```
In [35] data = [
        [1, 2],
        [3, 4]
    ]

    df = pd.DataFrame(
        data = data, index = ["A", "B"], columns = ["X", "Y"]
    )

    df
```

```
Out [35]
```

```
   X  Y
A  1  2
B  3  4
```

Синтаксис с квадратными скобками позволяет извлечь столбец из объекта `DataFrame` в виде объекта `Series`:

```
In [36] df["X"]
```

```
Out [36] A    1
         B    3
         Name: X, dtype: int64
```

Предположим, нам нужно извлечь столбец из объекта `neighborhoods`. Для задания каждого из четырех столбцов нашего объекта `DataFrame` необходимо сочетание двух идентификаторов: `Category` и `Subcategory`. Что получится, если указать только один?

7.4.1. Извлечение одного или нескольких столбцов

Если передать в квадратных скобках только одно значение, библиотека `pandas` будет искать его на самом внешнем уровне объекта `MultiIndex` для столбцов. В следующем примере мы ищем "Services" — одно из возможных значений уровня `Category`:

```
In [37] neighborhoods["Services"]
```

```
Out [37]
```

Subcategory		Police Schools		
State	City	Street		
AK	Rowlandchester	386 Rebecca Cove	A+	C
	Scottstad	082 Leblanc Freeway	D	B+
		114 Jones Garden	D	D

	Stevenshire	238	Andrew Rue	A-	A-
AL	Clarkland	430	Douglas Mission	C+	B+
...					
WY	Lake Nicole	754	Weaver Turnpike	B	D
		933	Jennifer Burg	A-	C
	Martintown	013	Bell Mills	A-	B-
	Port Jason	624	Faulkner Orchard	C+	C+
	Reneeshire	717	Patel Square	D	A

251 rows x 2 columns

Обратите внимание, что в этом новом объекте `DataFrame` отсутствует уровень `Category`. Этот объект содержит простой `Index` с двумя значениями: `"Police"` и `"Schools"`. `MultiIndex` становится не нужен, потому что два столбца в этом объекте `DataFrame` представляют собой подкатегории, относящиеся к `Service`. На уровне `Category` больше нет различающихся значений, чтобы их нужно было упоминать.

Если на наружном уровне мультииндекса столбцов значения нет, библиотека pandas генерирует исключение `KeyError`:

```
In [38] neighborhoods["Schools"]
```

```
-----
KeyError                                Traceback (most recent call last)
```

```
KeyError: 'Schools'
```

А что, если нам нужно выбрать конкретную категорию, а затем подкатеорию внутри нее? Значения, относящиеся к различным уровням в мультииндексе столбцов, можно передать в виде кортежа. В следующем примере мы выбираем столбец со значением `"Services"` на уровне `Category` и значение `"Schools"` на уровне `Subcategory`:

```
In [39] neighborhoods[("Services", "Schools")]
```

```
Out [39] State  City          Street
AK           Rowlandchester  386 Rebecca Cove      C
           Scottstad      082 Leblanc Freeway    B+
           Stevenshire    114 Jones Garden      D
AL           Clarkland     238 Andrew Rue      A-
           Clarkland     430 Douglas Mission    B+
           ..
WY           Lake Nicole   754 Weaver Turnpike    D
           Lake Nicole   933 Jennifer Burg      C
           Martintown     013 Bell Mills      B-
           Port Jason     624 Faulkner Orchard  C+
           Reneeshire     717 Patel Square      A
Name: (Services, Schools), Length: 251, dtype: object
```

Метод возвращает объект `Series` без индекса по столбцам! Опять же указанием конкретного значения уровня мультииндекса мы делаем его наличие в итоговом результате ненужным. Мы явно указываем библиотеке `pandas`, какие значения нас интересуют на уровнях `Category` и `Subcategory`, так что `pandas` удаляет эти два уровня из индекса столбцов. А поскольку сочетание ("`Services`", "`Schools`") дает один столбец данных, библиотека `pandas` возвращает объект `Series`.

Для извлечения нескольких столбцов объекта `DataFrame` необходимо передать список кортежей в квадратных скобках. Каждый из кортежей должен задавать значения уровней для одного столбца. Порядок кортежей внутри списка задает порядок столбцов в итоговом объекте `DataFrame`. Итак, пример: извлекаем два столбца из объекта `neighborhoods`:

```
In [40] neighborhoods[("Services", "Schools"), ("Culture", "Museums")]
```

```
Out [40]
```

Category				Services	Culture
Subcategory				Schools	Museums
State	City	Street			
AK	Rowlandchester	386	Rebecca Cove	C	A-
		082	Leblanc Freeway	B+	C-
		114	Jones Garden	D	D-
	Stevenshire	238	Andrew Rue	A-	A
AL	Clarkland	430	Douglas Mission	B+	F
...
WY	Lake Nicole	754	Weaver Turnpike	D	D-
		933	Jennifer Burg	C	A+
	Martintown	013	Bell Mills	B-	D
	Port Jason	624	Faulkner Orchard	C+	F
	Reneeshire	717	Patel Square	A	B+

251 rows × 2 columns

Чем больше круглых и квадратных скобок, тем больше вероятность ошибок и путаницы в синтаксисе. Предыдущий код можно упростить, присвоив список переменной и разбив кортежи по нескольким строкам:

```
In [41] columns = [
            ("Services", "Schools"),
            ("Culture", "Museums")
        ]

        neighborhoods[columns]
```

```
Out [41]
```

Category			Services		Culture
Subcategory			Schools		Museums
State	City	Street			
AK	Rowlandchester	386 Rebecca Cove		C	A-
	Scottstad	082 Leblanc Freeway		B+	C-
		114 Jones Garden		D	D-
	Stevenshire	238 Andrew Rue		A-	A
AL	Clarkland	430 Douglas Mission		B+	F
...	
WY	Lake Nicole	754 Weaver Turnpike		D	D-
		933 Jennifer Burg		C	A+
	Martintown	013 Bell Mills		B-	D
	Port Jason	624 Faulkner Orchard		C+	F
	Reneeshire	717 Patel Square		A	B+

251 rows × 6 columns

Оба эти примера дают один результат, но читать второй вариант кода намного удобнее: его синтаксис четко демонстрирует, где начинается и заканчивается каждый из кортежей.

7.4.2. Извлечение одной или нескольких строк с помощью `loc`

В главе 4 я познакомил вас с методами-получателями `loc` и `iloc`, предназначенными для выборки строк и столбцов из объекта `DataFrame`. Метод-получатель `loc` извлекает данные по метке индекса, а `iloc` — по позиции индекса. Вот краткая демонстрация работы этих методов на примере объекта `DataFrame` `df`, объявленного нами в разделе 7.4.1:

```
In [42] df
```

```
Out [42]
```

```

   X  Y
-----
A  1  2
B  3  4
```

В следующем примере мы извлекаем с помощью `loc` строку с меткой индекса "A":

```
In [43] df.loc["A"]
```

```

Out [43] X    1
         Y    2
         Name: A, dtype: int64
```


А в примере ниже извлекаем с помощью `iloc` строку с позицией индекса 1:

```
In [44] df.iloc[1]

Out [44] X      3
         Y      4
         Name: B, dtype: int64
```

Методы-получатели `loc` и `iloc` можно применять и для извлечения строки из мультииндексных объектов `DataFrame`. Разберем по шагам, как это делается.

Объект `MultiIndex` объекта `DataFrame` `neighborhoods` содержит три уровня: `State`, `City` и `Address`. Если известно, какие значения на каждом из уровней нас интересуют, можно передать их в кортеже в квадратных скобках. При конкретном указании значения для уровня выводить этот уровень в результате уже не требуется. Укажем значение "TX" для уровня `State`, "Kingchester" для уровня `City` и "534 Gordon Falls" для `Address`. Библиотека `pandas` возвращает объект `Series`, индекс которого формируется из заголовков столбцов объекта `neighborhoods`:

```
In [45] neighborhoods.loc[("TX", "Kingchester", "534 Gordon Falls")]

Out [45] Category  Subcategory
         Culture   Restaurants      C
         Museums      D+
         Services   Police        B
         Schools      B
         Name: (TX, Kingchester, 534 Gordon Falls), dtype: object
```

Если передать в квадратных скобках только одну метку, библиотека `pandas` будет искать ее в самом внешнем уровне мультииндекса. Давайте извлечем строки со значением "CA" уровня `State`. `State` — первый уровень мультииндекса для строк:

```
In [46] neighborhoods.loc["CA"]
```

```
Out [46]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
City	Street					
Dustinmouth	793 Cynthia ...		A-	A+	C-	A
North Jennifer	303 Alisha Road		D-	C+	C+	A+
Ryanfort	934 David Run		F	B+	F	D-

Библиотека `pandas` возвращает объект `DataFrame` с двухуровневым мультииндексом. Обратите внимание, что в нем отсутствует уровень `State`. Поскольку все три строки относятся к одному и тому же значению этого уровня, он больше не нужен.

Обычно второй аргумент в квадратных скобках отмечает столбец (-ы), который мы хотим извлечь, но можно также указать значение для поиска на следующем уровне мультииндекса. В следующем примере мы извлекаем строки со значением "CA" уровня *State* и значением "Dustinmouth" уровня *City*. И снова библиотека pandas возвращает объект *DataFrame* с меньшим на единицу количеством уровней. А поскольку остается только один уровень, pandas снова переходит на хранение меток строк с уровня *Street* в обычном объекте *Index*:

```
In [47] neighborhoods.loc["CA", "Dustinmouth"]
```

```
Out [47]
```

Category	Culture	Services
Subcategory	Restaurants	Museums
Street	Police	Schools
793 Cynthia Square	A-	A+

По-прежнему можно указать извлекаемый столбец (-ы) с помощью второго аргумента метода *loc*. Приведу пример извлечения строк со значением "CA" уровня *State* мультииндекса строк и значением "Culture" уровня *Category* мультииндекса столбцов:

```
In [48] neighborhoods.loc["CA", "Culture"]
```

```
Out [48]
```

Subcategory	Restaurants	Museums
City	Street	
Dustinmouth	793 Cynthia Square	A-
North Jennifer	303 Alisha Road	D-
Ryanfort	934 David Run	F

Синтаксис в предыдущих двух примерах неидеален из-за возможной неоднозначности. Второй аргумент метода *loc* может соответствовать либо значению со второго уровня мультииндекса строк, либо значению с первого уровня мультииндекса столбцов.

Документация библиотеки pandas¹ рекомендует следующую стратегию индексации во избежание подобной неопределенности. Используйте первый аргумент *loc* для меток индекса строк, а второй — для меток индекса столбцов. Обертывайте все аргументы для конкретного индекса в кортеж. При следовании такому стандарту нужно поместить задаваемые значения уровней мультииндекса строк в кортеж, как и значения уровней мультииндекса столбцов. Рекоменду-

¹ См. Advanced indexing with hierarchical index, <http://mng.bz/5WJO>.

емый способ обращения к строкам со значением "CA" уровня State и значением "Dustinmouth" уровня City таков:

```
In [49] neighborhoods.loc[("CA", "Dustinmouth")]
```

```
Out [49]
```

Category Subcategory Street	Culture		Services	
	Restaurants	Museums	Police	Schools
793 Cynthia Square	A-	A+	C-	A

Этот синтаксис проще и нагляднее: при нем второй аргумент `loc` всегда отражает искомые метки индекса столбцов. В следующем примере извлечем столбцы `Services` для того же штата "CA" и города "Dustinmouth". При этом передадим "Services" в виде кортежа. Кортеж из одного элемента требует указания запятой в конце, чтобы интерпретатор Python понял, что это кортеж:

```
In [50] neighborhoods.loc[("CA", "Dustinmouth"), ("Services",)]
```

```
Out [50]
```

Subcategory Street	Police	Schools
793 Cynthia Square	C-	A

Еще один полезный совет: библиотека `pandas` различает списки и кортежи в качестве аргументов методов-получателей. Используйте списки для хранения нескольких ключей, а кортежи — для хранения компонентов одного многоуровневого ключа.

Значения уровней в мультииндексе для столбцов можно передать в виде кортежа во втором аргументе метода `loc`. Вот пример задания значений:

- "CA" и "Dustinmouth" для уровней мультииндекса строк;
- "Services" и "Schools" для уровней мультииндекса столбцов.

Благодаря помещению "Services" и "Schools" в один кортеж библиотека `pandas` рассматривает их как компоненты одной метки. "Services" — значение для уровня `Category`, а "Schools" — для уровня `Subcategory`:

```
In [51] neighborhoods.loc[("CA", "Dustinmouth"), ("Services", "Schools")]
```

```
Out [51] Street
```

```
793 Cynthia Square    A
Name: (Services, Schools), dtype: object
```

А как выбрать несколько последовательных строк? Для этого можно воспользоваться синтаксисом срезов списков языка Python, указав двоеточие между начальной и конечной точкой последовательности. Приведу пример кода, в котором мы извлечем все последовательные строки со значениями уровня State от "NE" до "NH". В срезах библиотеки pandas конечная точка (значение после двоеточия) включается в интервал:

```
In [52] neighborhoods["NE":"NH"]
```

```
Out [52]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
NE	Barryborough	460 Anna Tunnel	A+	A+	B	A
	Shawncchester	802 Cook Cliff	D-	D+	D	A
	South Kennet...	346 Wallace ...	C-	B-	A	A-
	South Nathan	821 Jake Fork	C+	D	D+	A
NH	Courtneyfort	697 Spencer ...	A+	A+	C+	A+
	East Deborah...	271 Ryan Mount	B	C	D+	B-
	Ingramton	430 Calvin U...	C+	D+	C	C-
	North Latoya	603 Clark Mount	D-	A-	B+	B-
	South Tara	559 Michael ...	C-	C-	F	B

Синтаксис срезов списков языка Python можно сочетать с аргументами-кортежами. Извлечем строки:

- начиная со значения "NE" на уровне State и значения "Shawncchester" на уровне City;
- заканчивая значением "NH" на уровне State и значением "North Latoya" на уровне City.

```
In [53] neighborhoods.loc[("NE", "Shawncchester"):(("NH", "North Latoya"))]
```

```
Out [53]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
NE	Shawncchester	802 Cook Cliff	D-	D+	D	A
	South Kennet...	346 Wallace ...	C-	B-	A	A-
	South Nathan	821 Jake Fork	C+	D	D+	A
NH	Courtneyfort	697 Spencer ...	A+	A+	C+	A+
	East Deborah...	271 Ryan Mount	B	C	D+	B-
	Ingramton	430 Calvin U...	C+	D+	C	C-
	North Latoya	603 Clark Mount	D-	A-	B+	B-

Будьте осторожнее с этим синтаксисом: одна пропущенная скобка или запятая может привести к генерации исключения. Код можно упростить, присвоив кортежи переменным с понятными названиями и разбив код извлечения на меньшие части. Следующий код возвращает тот же результат, что и предыдущий, но читать его намного удобнее:

```
In [54] start = ("NE", "Shawncchester")
        end   = ("NH", "North Latoya")
        neighborhoods.loc[start:end]
```

Out [54]

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
NE	Shawncchester	802 Cook Cliff	D-	D+	D	A
	South Kennet...	346 Wallace ...	C-	B-	A	A-
	South Nathan	821 Jake Fork	C+	D	D+	A
NH	Courtneyfort	697 Spencer ...	A+	A+	C+	A+
	East Deborah...	271 Ryan Mount	B	C	D+	B-
	Ingramton	430 Calvin U...	C+	D+	C	C-
	North Latoya	603 Clark Mount	D-	A-	B+	B-

Указывать в каждом кортеже значения для всех уровней не обязательно. Приведу пример, который не включает значения уровня `City` для второго кортежа:

```
In [55] neighborhoods.loc[("NE", "Shawncchester"):(("NH"))]
```

Out [55]

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
NE	Shawncchester	802 Cook Cliff	D-	D+	D	A
	South Kennet...	346 Wallace ...	C-	B-	A	A-
	South Nathan	821 Jake Fork	C+	D	D+	A
NH	Courtneyfort	697 Spencer ...	A+	A+	C+	A+
	East Deborah...	271 Ryan Mount	B	C	D+	B-
	Ingramton	430 Calvin U...	C+	D+	C	C-
	North Latoya	603 Clark Mount	D-	A-	B+	B-
	South Tara	559 Michael ...	C-	C-	F	B

Библиотека `pandas` извлекает при этом все строки, начиная с `("NE", "Shawncchester")`, до тех пор, пока не достигнет конца строк со значением `"NH"` на уровне `State`.

7.4.3. Извлечение одной или нескольких строк с помощью `iloc`

Метод-получатель `iloc` извлекает строки и столбцы по позиции индекса. Пример ниже служит для того, чтобы освежить в вашей памяти изложенное в главе 4. Извлечем одну строку, передав методу `iloc` позицию индекса:

```
In [56] neighborhoods.iloc[25]
```

```
Out [56] Category  Subcategory
          Culture    Restaurants    A+
                Museums           A
          Services    Police       A+
                Schools       C+
          Name: (CT, East Jessicaland, 208 Todd Knolls), dtype: object
```

Методу `iloc` можно передать два аргумента, отражающих индексы строки и столбца. Приведу пример извлечения значения на пересечении строки с позицией индекса 25 и столбца с позицией индекса 2:

```
In [57] neighborhoods.iloc[25, 2]
```

```
Out [57] 'A+'
```

Можно извлечь и несколько строк, обернув их позиции индекса в список:

```
In [58] neighborhoods.iloc[[25, 30]]
```

```
Out [58]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
CT	East Jessica...	208 Todd Knolls	A+	A	A+	C+
DC	East Lisaview	910 Sandy Ramp	A-	A+	B	B

Срезы с помощью методов `loc` и `iloc` серьезно различаются. При срезах по индексам с помощью `iloc` конечная точка не включается в интервал. В предыдущем примере у записи с улицей 910 Sandy Ramp — позиция индекса 30. Если же указать 30 в качестве конечной точки для `iloc` в следующем примере, библиотека pandas извлечет все предшествующее этому индексу, но не включит его самого:

```
In [59] neighborhoods.iloc[25:30]
```

```
Out [59]
```

Category			Culture		Services	
Subcategory			Restaurants	Museums	Police	Schools
State	City	Street				
CT	East Jessica...	208 Todd Knolls	A+	A	A+	C+
	New Adrianhaven	048 Brian Cove	A-	C+	A+	D-
	Port Mike	410 Keith Lodge	D-	A	B+	D
	Sethstad	139 Bailey G...	C	C-	C+	A+
DC	East Jessica	149 Norman C...	A-	C-	C+	A-

Срезы по столбцам происходят аналогично. Извлечем столбцы с позициями индекса от 1 до 3 (не включая последний):

```
In [60] neighborhoods.iloc[25:30, 1:3]
```

```
Out [60]
```

Category			Culture	Services
Subcategory			Museums	Police
State	City	Street		
CT	East Jessica...	208 Todd Knolls	A	A+
	New Adrianhaven	048 Brian Cove	C+	A+
	Port Mike	410 Keith Lodge	A	B+
	Sethstad	139 Bailey G...	C-	C+
DC	East Jessica	149 Norman C...	C-	C+

Библиотека pandas допускает также и отрицательные срезы. Вот, например, так мы извлекаем строки, начиная от четвертой с конца, и столбцы, начиная с предпоследнего:

```
In [61] neighborhoods.iloc[-4:, -2:]
```

```
Out [61]
```

Category			Services	
Subcategory			Police	Schools
State	City	Street		
WY	Lake Nicole	933 Jennifer...	A-	C
	Martintown	013 Bell Mills	A-	B-
	Port Jason	624 Faulkner...	C+	C+
	Reneeshire	717 Patel Sq...	D	A

Библиотека pandas ставит в соответствие позицию индекса каждой строке объекта `DataFrame`, но не каждому значению на конкретном уровне индекса. Следовательно, индексация последовательных уровней мультииндекса с помощью `iloc` невозможна. Это ограничение — вполне обдуманый выбор команды создателей библиотеки pandas. Как указывает разработчик Джефф

Рибэк (Jeff Reback), `iloc` играет роль «строго позиционного средства доступа по индексу», которое «вообще не принимает во внимание внутреннюю структуру объекта `DataFrame`»¹.

7.5. ПОПЕРЕЧНЫЕ СРЕЗЫ

Метод `xs` дает возможность извлекать строки, указывая значение *одного* уровня мультииндекса. Этому методу передается параметр `key` с искомым значением. А в параметре `level` передается либо числовая позиция, либо название уровня индекса, на котором следует искать это значение. Допустим, мы хотим найти все адреса в городе **Lake Nicole**, неважно, в каком штате или на какой улице. `City` — второй уровень мультииндекса; в иерархии уровней у него позиция индекса 1:

```
In [62] # Две строки ниже эквивалентны
neighborhoods.xs(key = "Lake Nicole", level = 1)
neighborhoods.xs(key = "Lake Nicole", level = "City")
```

Out [62]

Category		Culture	Services		
Subcategory		Restaurants	Museums	Police	Schools
State	Street				
OR	650 Angela Track	D	C-	D	F
WY	754 Weaver Turnpike	B	D-	B	D
	933 Jennifer Burg	C	A+	A-	C

В городе **Lake Nicole** нашлось три адреса в двух разных штатах. Обратите внимание, что библиотека `pandas` убрала уровень `City` из мультииндекса нового объекта `DataFrame`. Значение `City` — фиксированное ("**Lake Nicole**"), так что включать его нет смысла.

Ту же методику извлечения данных можно применить и к столбцам, передав аргумент `"columns"` параметра `axis`. В следующем примере мы извлекаем столбцы с ключом `"Museums"` на уровне `Subcategory` мультииндекса столбцов. Этому описанию удовлетворяет только один столбец:

```
In [63] neighborhoods.xs(
axis = "columns", key = "Museums", level = "Subcategory"
).head()
```

Out [63]

¹ См.: *Reback, J.* Inconsistent behavior of `loc` and `iloc` for `MultiIndex`, <https://github.com/pandas-dev/pandas/issues/15228>.

Category		Culture		
State	City	Street		
AK	Rowlandchester	386	Rebecca Cove	A-
	Scottstad	082	Leblanc Freeway	C-
		114	Jones Garden	D-
	Stevenshire	238	Andrew Rue	A
AL	Clarkland	430	Douglas Mission	F

Обратите внимание, что уровень `Subcategory` отсутствует в возвращаемом объекте `DataFrame`, в отличие от уровня `Category`. Библиотека `pandas` включает туда `Category`, поскольку различные значения на его уровне все еще возможны. Извлекаемые из промежуточного уровня значения могут относиться к нескольким меткам верхнего уровня.

Мы также можем передавать методу `xs` ключи, относящиеся к уровням мультииндекса, расположенным в порядке иерархии уровней непоследовательно. Передавать их можно в кортеже. Пусть, к примеру, нас интересуют строки со значением "238 Andrew Rue" уровня `Street` и значением "AK" уровня `State`, вне зависимости от значения уровня `City`. Нужно сделать их выборку. Благодаря методу `xs` реализовать это очень просто:

```
In [64] # Две строки ниже эквивалентны
neighborhoods.xs(
    key = ("AK", "238 Andrew Rue"), level = ["State", "Street"]
)
neighborhoods.xs(
    key = ("AK", "238 Andrew Rue"), level = [0, 2]
)
```

Out [64]

Category	Culture		Services	
	Subcategory	Restaurants	Museums	Police Schools
City				

Stevenshire		D-	A	A- A-

Выборка значений, относящихся только к конкретному уровню, — замечательная возможность `MultiIndex`.

7.6. ОПЕРАЦИИ НАД ИНДЕКСОМ

В начале этой главы мы привели наш набор данных `neighborhoods` к текущей форме на этапах его создания путем подбора нужных параметров функции `read_csv`. Библиотека `pandas` позволяет также производить различные операции над существующим мультииндексом существующего объекта `DataFrame`. Давайте взглянем, как это происходит.

7.6.1. Замена индекса

В настоящее время самый внешний уровень мультииндекса в объекте `DataFrame` `neighborhoods` — `State`, а за ним следуют `City` и `Street`:

```
In [65] neighborhoods.head()
```

Out [65]

Category				Culture		Services	
Subcategory				Restaurants	Museums	Police	Schools
State	City	Street					
AK	Rowlandchester	386 Rebecca Cove		C-	A-	A+	C
	Scottstad	082 Leblanc Fr...		D	C-	D	B+
		114 Jones Garden		D-	D-	D	D
	Stevenshire	238 Andrew Rue		D-	A	A-	A-
AL	Clarkland	430 Douglas Mi...		A	F	C+	B+

Метод `reorder_levels` изменяет упорядоченность уровней мультииндекса заданным образом. Желаемый порядок задается путем передачи списка уровней в параметре `order`. В следующем примере уровни `City` и `State` меняются местами:

```
In [66] new_order = ["City", "State", "Street"]
        neighborhoods.reorder_levels(order = new_order).head()
```

Out [66]

Category				Culture		Services	
Subcategory				Restaurants	Museums	Police	Schools
City	State	Street					
Rowlandchester	AK	386 Rebecca ...		C-	A-	A+	C
Scottstad	AK	082 Leblanc ...		D	C-	D	B+
		114 Jones Ga...		D-	D-	D	D
Stevenshire	AK	238 Andrew Rue		D-	A	A-	A-
Clarkland	AL	430 Douglas ...		A	F	C+	B+

Можно также передать в параметр `order` список целых чисел, соответствующих текущим позициям индекса уровней мультииндекса. Чтобы, например, `State` стал первым уровнем в новом мультииндексе, необходимо начать список с 1 — позиции индекса уровня `State` в текущем мультииндексе. Следующий пример кода возвращает тот же результат, что и предыдущий:

```
In [67] neighborhoods.reorder_levels(order = [1, 0, 2]).head()
```

Out [67]

Category Subcategory			Culture		Services	
	State	Street	Restaurants	Museums	Police	Schools
City						
Rowlandchester	AK	386 Rebecca ...	C-	A-	A+	C
Scottstad	AK	082 Leblanc ...	D	C-	D	B+
		114 Jones Ga...	D-	D-	D	D
Stevenshire	AK	238 Andrew Rue	D-	A	A-	A-
Clarkland	AL	430 Douglas ...	A	F	C+	B+

А если необходимо, скажем, удалить индекс? Например, если мы хотим использовать в качестве меток индекса другой набор столбцов? Метод `reset_index` возвращает новый объект `DataFrame`, включающий уровни предыдущего объекта `MultiIndex` в качестве столбцов. Библиотека `pandas` заменяет предыдущий мультииндекс на стандартный числовой:

```
In [68] neighborhoods.reset_index().tail()
```

```
Out [68]
```

Category Subcategory	State	City	Street	Culture		Services	
				Restaurants	Museums	Police	Schools
246	WY	Lake...	754 ...	B	D-	B	D
247	WY	Lake...	933 ...	C	A+	A-	C
248	WY	Mart...	013 ...	C-	D	A-	B-
249	WY	Port...	624 ...	A-	F	C+	C+
250	WY	Rene...	717 ...	B	B+	D	A

Обратите внимание, что три новых столбца (`State`, `City` и `Street`) становятся значениями в `Category` — на самом внешнем уровне мультииндекса столбцов. Ради согласованности столбцов (чтобы каждый был кортежем из двух значений) библиотека `pandas` присваивает трем новым столбцам равное пустой строке значение на уровне `Subcategory`.

`Pandas` предоставляет возможность добавить эти три столбца на другой уровень мультииндекса. Передайте нужную позицию индекса или название в параметр `col_level` метода `reset_index`. В следующем примере столбцы `State`, `City` и `Street` включаются в уровень `Subcategory` мультииндекса столбцов:

```
In [69] # Две строки ниже эквивалентны
         neighborhoods.reset_index(col_level = 1).tail()
         neighborhoods.reset_index(col_level = "Subcategory").tail()
```

```
Out [69]
```

Category	Subcategory	State	City	Street	Culture		Services	
					Restaurants	Museums	Police	Schools
246		WY	Lake...	754 ...	B	D-	B	D
247		WY	Lake...	933 ...	C	A+	A-	C
248		WY	Mart...	013 ...	C-	D	A-	B-
249		WY	Port...	624 ...	A-	F	C+	C+
250		WY	Rene...	717 ...	B	B+	D	A

Теперь библиотека pandas по умолчанию будет использовать пустую строку в качестве значения **Category** — родительского уровня, включающего уровень **Subcategory**, к которому относятся **State**, **City** и **Street**. Пустую строку можно заменить любым нужным нам значением, передав соответствующий аргумент для параметра **col_fill**. В следующем примере мы группируем три новых столбца на родительском уровне **Address**. Теперь самый внешний уровень **Category** включает три различных значения **Address**, **Culture** и **Services**:

```
In [70] neighborhoods.reset_index(
        col_fill = "Address", col_level = "Subcategory"
    ).tail()
```

Out [70]

Category	Address			Street	Culture		Services	
	Subcategory	State	City		Restaurants	Museums	Police	Schools
246		WY	Lake...	754 ...	B	D-	B	D
247		WY	Lake...	933 ...	C	A+	A-	C
248		WY	Mart...	013 ...	C-	D	A-	B-
249		WY	Port...	624 ...	A-	F	C+	C+
250		WY	Rene...	717 ...	B	B+	D	A

Обычный вызов метода **reset_index** приводит к преобразованию всех уровней индекса в обычные столбцы. Можно также перенести в обычный столбец и отдельный уровень индекса, передав его название в параметре **levels**. Например, перенесем уровень **Street** из мультииндекса в обычный столбец объекта **DataFrame**:

```
In [71] neighborhoods.reset_index(level = "Street").tail()
```

Out [71]

Category		Street		Culture		Services		
Subcategory				Restaurants	Museums	Police	Schools	
State	City							
WY	Lake Nicole	754	Weaver Tur...		B	D-	B	D
	Lake Nicole	933	Jennifer Burg		C	A+	A-	C
	Martintown		013 Bell Mills		C-	D	A-	B-
	Port Jason	624	Faulkner O...		A-	F	C+	C+
	Reneeshire	717	Patel Square		B	B+	D	A

А можно перенести в столбцы несколько уровней индекса, передав их в виде списка:

```
In [72] neighborhoods.reset_index(level = ["Street", "City"]).tail()
```

```
Out [72]
```

Category Subcategory State	City	Street	Culture		Services		
			Restaurants	Museums	Police	Schools	
WY	Lake Nicole	754 Weav...		B	D-	B	D
WY	Lake Nicole	933 Jenn...		C	A+	A-	C
WY	Martintown	013 Bell...		C-	D	A-	B-
WY	Port Jason	624 Faul...		A-	F	C+	C+
WY	Reneeshire	717 Pate...		B	B+	D	A

А произвести удаление уровня из мультииндекса? Конечно, можно! Если задать значение `True` для параметра `drop` метода `reset_index`, библиотека `pandas` удалит указанный уровень вместо добавления его в число столбцов. В следующем примере `reset_index` мы удаляем таким образом уровень `Street`:

```
In [73] neighborhoods.reset_index(level = "Street", drop = True).tail()
```

```
Out [73]
```

Category Subcategory State	City	Culture		Services	
		Restaurants	Museums	Police	Schools
WY	Lake Nicole	B	D-	B	D
	Lake Nicole	C	A+	A-	C
	Martintown	C-	D	A-	B-
	Port Jason	A-	F	C+	C+
	Reneeshire	B	B+	D	A

Чтобы подготовиться к подразделу 7.6.2, где мы займемся созданием новых индексов, зафиксируем наши изменения индекса, перезаписав переменную `neighborhoods` новым объектом `DataFrame`. При этом все три уровня индекса перенесутся в столбцы объекта `DataFrame`:

```
In [74] neighborhoods = neighborhoods.reset_index()
```

7.6.2. Задание индекса

Освежим в памяти, как выглядит наш объект `DataFrame`:

```
In [75] neighborhoods.head(3)
```

```
Out [75]
```

Category Subcategory	State	City	Street	Culture		Services	
				Restaurants	Museums	Police	Schools
0	AK	Rowl...	386 ...	C-	A-	A+	C
1	AK	Scot...	082 ...	D	C-	D	B+
2	AK	Scot...	114 ...	D-	D-	D	D

Метод `set_index` устанавливает в качестве нового индекса один или несколько столбцов объекта `DataFrame`. Нужные столбцы можно передать через параметр `keys`:

```
In [76] neighborhoods.set_index(keys = "City").head()
```

```
Out [76]
```

Category Subcategory City	State	Street	Culture		Services	
			Restaurants	Museums	Police	Schools
Rowlandchester	AK	386 Rebecca...	C-	A-	A+	C
Scottstad	AK	082 Leblanc...	D	C-	D	B+
Scottstad	AK	114 Jones G...	D-	D-	D	D
Stevenshire	AK	238 Andrew Rue	D-	A	A-	A-
Clarkland	AL	430 Douglas...	A	F	C+	B+

Нам нужно, чтобы роль индекса играл один из четырех последних столбцов? Пожалуйста! В следующем примере мы передаем в параметр `keys` кортеж с целевыми значениями для всех уровней мультииндекса:

```
In [77] neighborhoods.set_index(keys = ("Culture", "Museums")).head()
```

```
Out [77]
```

Category Subcategory (Cultur...	State	City	Street	Culture	Services	
				Restaurants	Police	Schools
A-	AK	Rowlan...	386 Re...	C-	A+	C
C-	AK	Scottstad	082 Le...	D	D	B+
D-	AK	Scottstad	114 Jo...	D-	D	D
A	AK	Steven...	238 An...	D-	A-	A-
F	AL	Clarkland	430 Do...	A	C+	B+

Для создания мультииндекса на оси строк можно передать в параметр `keys` список, содержащий несколько столбцов:

```
In [78] neighborhoods.set_index(keys = ["State", "City"]).head()
```

```
Out [78]
```

Category		Street		Culture		Services		
Subcategory				Restaurants	Museums	Police	Schools	
State	City							
AK	Rowlandchester	386	Rebecca...		C-	A-	A+	C
	Scottstad	082	Leblanc...		D	C-	D	B+
	Scottstad	114	Jones G...		D-	D-	D	D
	Stevenshire	238	Andrew Rue		D-	A	A-	A-
AL	Clarkland	430	Douglas...		A	F	C+	B+

Одним словом, при работе с библиотекой `pandas` можно придать набору данных подходящую для анализа форму с помощью множества вариантов сочетаний и преобразований. Мы уже не раз это делали и даже вошли во вкус. Надо только четко определить для себя задачи проводимого анализа и вид необходимых наборов данных, а затем задавать индексы объекта `DataFrame`. Спросите себя перед тем, как приступить к преобразованиям, какие значения важнее всего для решения поставленной перед вами задачи. Какая информация является ключевой? Связаны ли друг с другом неразрывно какие-то элементы данных? Составляют ли строки или столбцы группу или категорию? Мультииндекс — эффективный вариант хранения и доступа к данным для решения множества самых разнообразных задач.

7.7. УПРАЖНЕНИЯ

Пришло время попрактиковаться в использовании изложенных в этой главе идей и затем проверить себя. Начнем, пожалуй!

7.7.1. Задачи

Набор данных `investments.csv` взят с веб-сайта Crunchbase. Он включает более чем 27 000 записей о вложении средств в стартапы. У каждого стартапа есть название (`Name`), рынок (`Market`), состояние (`Status`), штат проведения деятельности (`State of operation`) и несколько раундов инвестиций (`Funding Rounds`):

```
In [79] investments = pd.read_csv("investments.csv")
        investments.head()
```

Out [79]

	Name	Market	Status	State	Funding Rounds
0	#waywire	News	Acquired	NY	1
1	&TV Communications	Games	Operating	CA	2
2	-R- Ranch and Mine	Tourism	Operating	TX	2

3	004 Technologies	Software	Operating	IL	1
4	1-4 All	Software	Operating	NC	1

Добавим в этот объект `DataFrame` мультииндекс. Можно начать с определения количества уникальных значений в каждом из столбцов с помощью метода `nunique`. Столбцы с небольшим количеством уникальных значений обычно выражают категориальные данные и хорошо подходят на роль уровней индекса:

```
In [80] investments.nunique()
Out [80] Name          27763
        Market         693
        Status          3
        State          61
        Funding Rounds  16
        dtype: int64
```

Создадим трехуровневый мультииндекс со столбцами `Status`, `Funding Rounds` и `State`. Для начала упорядочим эти столбцы так, чтобы первыми шли столбцы с меньшим количеством значений. Чем меньше уникальных значений на уровне, тем быстрее библиотека `pandas` сможет извлечь его строки. Кроме того, отсортируем индекс объекта `DataFrame` для ускорения поиска:

```
In [81] investments = investments.set_index(
        keys = ["Status", "Funding Rounds", "State"]
    ).sort_index()
```

Теперь объект `investments` выглядит следующим образом:

```
In [82] investments.head()
Out [82]
```

Status	Funding Rounds	State	Name	Market
Acquired	1	AB	Hallpass Media	Games
		AL	EnteGreat	Enterprise Soft...
		AL	Onward Behaviora...	Biotechnology
		AL	Proxsys	Biotechnology
		AZ	Envox Group	Public Relations

А вот и задачи для этой главы.

1. Извлеките все строки с состоянием "Closed".
2. Извлеките все строки с состоянием "Acquired" и десятью раундами инвестиций.
3. Извлеките все строки с состоянием "Operating", шестью раундами инвестиций и штатом "NJ".
4. Извлеките все строки с состоянием "Closed" и восемью раундами инвестиций. Извлеките только столбец `Name`.

5. Извлеките все строки со штатом "NJ" вне зависимости от значений уровней `Status` и `Funding Rounds`.
6. Верните уровни мультииндекса обратно в число столбцов объекта `DataFrame`.

7.7.2. Решения

Давайте решать задачи по очереди.

1. Для извлечения всех строк с состоянием "Closed" можно воспользоваться методом-получателем `loc`. Передаем в него кортеж с одним значением "Closed". Напомню, что кортеж из одного элемента требует указания запятой в конце:

```
In [83] investments.loc[("Closed",)].head()
```

```
Out [83]
```

Funding Rounds State			Name	Market

1	AB	Cardinal Media Technologies	Social Network Media	
	AB	Easy Bill Online	Tracking	
	AB	Globe1 Direct	Public Relations	
	AB	Ph03nix New Media	Games	
	AL	Naubo	News	

2. Далее требуется извлечь строки, удовлетворяющие двум условиям: значению "Acquired" уровня `Status` и значению 10 уровня `Funding Rounds`. Эти уровни идут в мультииндексе один за другим, так что можно передать кортеж с соответствующими значениями в метод-получатель `loc`:

```
In [84] investments.loc[("Acquired", 10)]
```

```
Out [84]
```

State		Name	Market

NY	Genesis Networks	Web Hosting	
TX	ACTIVE Network	Software	

3. Можно воспользоваться тем же решением, что и для предыдущих двух задач. На этот раз нам нужно указать кортеж из трех значений, по одному для каждого уровня мультииндекса:

```
In [85] investments.loc[("Operating", 6, "NJ")]
```

```
Out [85]
```

Status	Funding Rounds	State	Name	Market
Operating	6	NJ	Agile Therapeutics	Biotechnology
		NJ	Agilence	Retail Technology
		NJ	Edge Therapeutics	Biotechnology
		NJ	Nistica	Web Hosting

4. Для извлечения столбцов объекта `DataFrame` можно передать второй аргумент методу-получателю `loc`. В этой задаче мы передадим кортеж из одного элемента — столбца `Name`. В первом аргументе по-прежнему содержатся значения для уровней `Status` и `Funding Rounds`:

```
In [86] investments.loc[("Closed", 8), ("Name",)]
```

```
Out [86]
```

State	Name
CA	CipherMax
CA	Dilithium Networks
CA	Moblyng
CA	SolFocus
CA	Solyndra
FL	Extreme Enterprises
GA	MedShape
NC	Biolex Therapeutics
WA	Cozi Group

5. Очередная задача требует от нас извлечения строк со значением "NJ" уровня `State`. Можно воспользоваться методом `xs`, передав в его параметр `level` либо позицию индекса этого уровня, либо его название:

```
In [87] # Две строки ниже эквивалентны
         investments.xs(key = "NJ", level = 2).head()
         investments.xs(key = "NJ", level = "State").head()
```

```
Out [87]
```

Status	Funding Rounds	Name	Market
Acquired	1	AkaRx	Biotechnology
	1	Aptalis Pharma	Biotechnology
	1	Cadent	Software
	1	Cancer Genetics	Health And Wellness
	1	Clacendix	E-Commerce

6. Наконец, мы хотим добавить уровни мультииндекса обратно в объект `DataFrame` в качестве столбцов. Для включения уровней индекса обратно мы

вызовом метод `reset_index` и перезапишем объект DataFrame `investments`, чтобы зафиксировать изменения:

```
In [88] investments = investments.reset_index()
        investments.head()
```

Out [88]

	Status	Funding Rounds	State	Name	Market
0	Acquired	1	AB	Hallpass Media	Games
1	Acquired	1	AL	EnteGreat	Enterprise Software
2	Acquired	1	AL	Onward Behaviora...	Biotechnology
3	Acquired	1	AL	Proxsys	Biotechnology
4	Acquired	1	AZ	Envox Group	Public Relations

Поздравляю с успешным выполнением упражнений!

РЕЗЮМЕ

- Мультииндекс (объект `MultiIndex`) — это индекс, состоящий из нескольких уровней.
- Для хранения меток в мультииндексах используются кортежи.
- Объект `DataFrame` может включать мультииндексы как по оси строк, так и по оси столбцов.
- Метод `sort_index` сортирует уровни мультииндекса. Библиотека `pandas` может сортировать уровни индекса по отдельности или группами.
- Методы-получатели `loc` (на основе меток) и `iloc` (на основе позиций) требуют указания дополнительных аргументов для извлечения нужного сочетания столбцов и строк.
- Во избежание неоднозначностей передавайте в методы-получатели `loc` и `iloc` кортежи.
- Метод `reset_index` включает уровни индекса обратно в объект `DataFrame` в качестве столбцов.
- Для формирования мультииндекса на основе существующих столбцов объекта `DataFrame` необходимо передать методу `set_index` список столбцов.

8

Изменение формы и сводные таблицы

В этой главе

- ✓ Сравнение широких и узких данных.
- ✓ Генерация сводной таблицы на основе объекта `DataFrame`.
- ✓ Агрегирование значений по сумме, среднему значению, количеству: тонкости процесса.
- ✓ Перенос уровней индексов объекта `DataFrame` с оси столбцов на ось строк и наоборот.
- ✓ Расплавление объекта `DataFrame`.

Набор данных изначально может находиться в непригодном для требуемого анализа формате. Например, проблемы могут быть в неподходящем типе данных столбца, отсутствующих значениях в строке или неправильном регистре символов в ячейке. Иногда несоответствия желаемому представлению данных ограничиваются конкретным столбцом, строкой или ячейкой. Но случается, что проблемы с набором оказываются структурными и выходят за пределы самих данных. Например, значения набора данных могут находиться в формате, при котором можно легко извлечь конкретную строку, но сложно агрегировать данные.

Изменение формы (reshaping) набора данных означает приведение его в другой вид, позволяющий получить информацию, которую невозможно/сложно получить из данных в исходном виде. То есть мы варьируем представление данных

через изменение формы. Навык изменения формы данных жизненно важен; согласно оценке одного исследования, 80 % анализа данных составляют их очистка и приведение в другую форму¹.

В этой главе мы рассмотрим новые технологии, реализованные в библиотеке *pandas* для приведения наборов данных в нужную нам форму. Для начала это будет методика получения сжатого представления большого набора данных в виде сводной таблицы. А затем — противоположная задача — трансформация агрегированного набора данных в плоскую структуру. К концу прочтения главы вы будете уметь мастерски преобразовывать данные в представление, лучше всего подходящее для ваших целей.

8.1. ШИРОКИЕ И УЗКИЕ ДАННЫЕ

Прежде чем заниматься методами вплотную, вкратце обсудим структуру наборов данных. Данные в наборе могут храниться в широком или узком формате. *Узкий* (narrow) набор данных называют также *длинным* (long) или *высоким* (tall). Эти названия отражают направление распространения данных по мере добавления новых. *Широкий* (wide) набор данных растет по ширине, разрастается. Узкий/длинный/высокий набор данных растет по высоте, углубляется.

Взгляните на следующую таблицу с измерениями температуры в двух городах за два дня:

	Weekday	Miami	New York
0	Monday	100	65
1	Tuesday	105	70

Рассмотрим *переменные* величины — измерения, значения которых варьируются. Можно подумать, что единственные переменные в этом наборе данных — дни недели и температура. Но в названиях столбцов скрывается еще одна переменная — города. Набор данных хранит одну и ту же смысловую переменную — температуру — в двух столбцах вместо одного. Заголовки *Miami* и *New York* никак не связаны по смыслу с температурой, не описывают хранящиеся в соответствующих столбцах данные — то есть *100* не является разновидностью *Miami*, подобно тому как, например, *Monday* — разновидность *Weekday*. В наборе данных переменная «город» скрыта посредством хранения ее в заголовках столбцов. Эту таблицу можно считать широким набором данных. Широкий набор данных при добавлении города расширяется горизонтально.

¹ См.: Wickham H. Tidy Data, Journal of Statistical Software, <https://vita.had.co.nz/papers/tidy-data.pdf>.

Допустим, мы захотели добавить в набор данных измерения температуры еще для двух городов. Нам пришлось бы добавить при этом два новых столбца для той же смысловой переменной: температуры. Обратите внимание на направление расширения набора данных. Таблица с данными становится шире, а не выше:

	Weekday	Miami	New York	Chicago	San Francisco
0	Monday	100	65	50	60
1	Tuesday	105	70	58	62

Горизонтальное расширение — это плохо? Не обязательно. Широкий набор данных идеально подходит для изучения общей картины происходящего. Набор данных удобен для чтения и понимания, если нас волнуют только температуры в понедельник и вторник. Но у широкого формата есть и свои недостатки. По мере добавления новых столбцов работать с данными становится все сложнее. Скажем, мы написали код для вычисления средней температуры за все дни. В настоящее время температуры хранятся в четырех столбцах. Если добавить столбец еще для одного города, придется менять логику вычислений, чтобы учесть это. Такая архитектура недостаточно гибкая.

Узкие наборы данных растут вертикально. Узкий формат упрощает операции над уже существующими данными и добавление новых записей. Все переменные локализованы в отдельных столбцах. Сравните первую таблицу этого раздела со следующей:

	Weekday	City	Temperature
0	Monday	Miami	100
1	Monday	New York	65
2	Tuesday	Miami	105
3	Tuesday	New York	70

Для включения в набор температур данных еще двух городов необходимо добавлять строки, а не столбцы. Данные становятся выше, а не шире:

	Weekday	City	Temperature
0	Monday	Miami	100
1	Monday	New York	65
2	Monday	Chicago	50
3	Monday	San Francisco	60
4	Tuesday	Miami	105
5	Tuesday	New York	70
6	Tuesday	Chicago	58
7	Tuesday	San Francisco	62

Проще ли стало отыскать температуру в понедельник для конкретных городов? Я бы сказал, что нет, поскольку теперь данные по каждому из них разбросаны по четырем строкам. Но вычислить среднюю температуру проще, поскольку

значения температуры ограничены отдельным столбцом. Логика вычисления среднего значения при добавлении новых строк не меняется.

Оптимальный формат хранения набора данных зависит от того, какую информацию мы хотим из него извлечь. А уж библиотека `pandas` предоставит инструменты для преобразования объектов `DataFrame` из узкого формата в широкий и наоборот. В данной главе вы узнаете, как это сделать.

8.2. СОЗДАНИЕ СВОДНОЙ ТАБЛИЦЫ ИЗ ОБЪЕКТА DATAFRAME

Наш первый набор данных, `sales_by_employee.csv`, представляет собой список коммерческих сделок вымышленной компании. Каждая строка включает дату продажи (`Date`), имя/название торгового агента (`Name`), покупателя (`Customer`), а также выручку (`Revenue`) и издержки (`Expenses`) от продажи:

```
In [1] import pandas as pd
```

```
In [2] pd.read_csv("sales_by_employee.csv").head()
```

```
Out [2]
```

	Date	Name	Customer	Revenue	Expenses
0	1/1/20	Oscar	Logistics XYZ	5250	531
1	1/1/20	Oscar	Money Corp.	4406	661
2	1/2/20	Oscar	PaperMaven	8661	1401
3	1/3/20	Oscar	PaperGenius	7075	906
4	1/4/20	Oscar	Paper Pound	2524	1767

Ради удобства одновременно с импортом преобразуем с помощью параметра `parse_dates` функции `read_csv` строковые значения из столбца `Date` в объекты даты/времени. После этой операции набор уже будет пригоден для дальнейшей работы. Можем присвоить полученный объект `DataFrame` переменной `sales`:

```
In [3] sales = pd.read_csv(
        "sales_by_employee.csv", parse_dates = ["Date"]
    )
    sales.tail()
```

```
Out [3]
```

	Date	Name	Customer	Revenue	Expenses
21	2020-01-01	Creed	Money Corp.	4430	548
22	2020-01-02	Creed	Average Paper Co.	8026	1906
23	2020-01-02	Creed	Average Paper Co.	5188	1768
24	2020-01-04	Creed	PaperMaven	3144	1314
25	2020-01-05	Creed	Money Corp.	938	1053

Набор данных загружен, давайте оценим, как можно агрегировать его данные с помощью сводной таблицы.

8.2.1. Метод `pivot_table`

Сводная таблица (pivot table) агрегирует значения столбца и группирует результаты по значениям из других столбцов. Слово «агрегирует» обозначает вычисление сводных показателей на основе нескольких значений. Примеры агрегирования: вычисление среднего значения, медианы, суммирование и подсчет количества значений. Сводная таблица в библиотеке pandas эквивалентна сводной таблице Excel.

Как обычно, лучше всего изучать что бы то ни было на примере, так что приступим к первой из наших задач. Несколько торговых агентов закрыли сделки в один день. Кроме того, каждый из агентов закрыл в один день по несколько сделок. Просуммируем выручку по дате и посмотрим, какой вклад внес каждый из агентов в общую выручку за день.

Для создания сводной таблицы выполним четыре шага.

1. Выберем столбцы, значения которых хотим агрегировать.
2. Выберем применяемую к столбцам операцию агрегирования.
3. Выберем столбцы, на основе значений которых будем группировать агрегированные данные по категориям.
4. Выберем, где размещать группы — на оси строк, оси столбцов или на обеих осях координат.

Пройдем по шагам поочередно. Во-первых, необходимо вызвать метод `pivot_table` для нашего объекта `DataFrame sales`. Параметр `index` этого метода принимает столбец со значениями, составляющими метки индекса сводной таблицы. Библиотека pandas группирует результаты по уникальным значениям из этого столбца.

Приведу пример, в котором роль меток индекса сводной таблицы играют значения столбца `Date`. Столбец `Date` содержит пять неповторяющихся дат. Библиотека pandas применяет ко всем числовым столбцам в `sales (Expenses и Revenue)` агрегирующую операцию по умолчанию — среднее значение:

```
In [4] sales.pivot_table(index = "Date")
```

```
Out [4]
```


Date	Expenses	Revenue
2020-01-01	637.500000	4293.500000
2020-01-02	1244.400000	7303.000000
2020-01-03	1313.666667	4865.833333
2020-01-04	1450.600000	3948.000000
2020-01-05	1196.250000	4834.750000

Метод возвращает обычный объект `DataFrame`. Может показаться странным, но как раз он представляет собой именно сводную таблицу! Эта таблица отражает средние издержки и среднюю выручку, сгруппированные по пяти уникальным датам из столбца `Date`.

Агрегирующая функция объявляется при помощи параметра `aggfunc`, аргумент по умолчанию которого — `"mean"`. Следующий код дает тот же результат, что и предыдущий:

```
In [5] sales.pivot_table(index = "Date", aggfunc = "mean")
```

```
Out [5]
```

Date	Expenses	Revenue
2020-01-01	637.500000	4293.500000
2020-01-02	1244.400000	7303.000000
2020-01-03	1313.666667	4865.833333
2020-01-04	1450.600000	3948.000000
2020-01-05	1196.250000	4834.750000

Напомню, что наша цель — просуммировать выручку за каждый день по торговым агентам, для ее достижения нам придется модифицировать часть аргументов метода `pivot_table`. Сначала поменяем аргумент параметра `aggfunc` на `"sum"`, чтобы получить сумму значений из столбцов `Expenses` и `Revenue`:

```
In [6] sales.pivot_table(index = "Date", aggfunc = "sum")
```

```
Out [6]
```

Date	Expenses	Revenue
2020-01-01	3825	25761
2020-01-02	6222	36515
2020-01-03	7882	29195
2020-01-04	7253	19740
2020-01-05	4785	19339

Пока нас интересует только суммирование значений из столбца `Revenue`. Агрегируемый столбец (-ы) объекта `DataFrame` указывается в параметре `values`. Чтобы агрегировать значения только одного столбца, можно передать в указанном параметре строковое значение с его названием:

```
In [7] sales.pivot_table(
        index = "Date", values = "Revenue", aggfunc = "sum"
    )
```

Out [7]

Date	Revenue
2020-01-01	25761
2020-01-02	36515
2020-01-03	29195
2020-01-04	19740
2020-01-05	19339

Для агрегирования значений в нескольких столбцах можно передать в `values` список значений.

Итак, мы получили суммарную выручку по дням. Осталось только вычислить вклад каждого из торговых агентов в сумму выручки за день. Удобнее всего будет вывести имя каждого из агентов в отдельном столбце. Другими словами, мы хотим воспользоваться уникальными значениями столбца `Name` в качестве заголовков столбцов сводной таблицы. Добавим в вызов метода параметр `columns`, указав для него аргумент `"Name"`:

```
In [8] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = "Revenue",
        aggfunc = "sum"
    )
```

Out [8]

Name	Creed	Dwight	Jim	Michael	Oscar
2020-01-01	4430.0	2639.0	1864.0	7172.0	9656.0
2020-01-02	13214.0	NaN	8278.0	6362.0	8661.0
2020-01-03	NaN	11912.0	4226.0	5982.0	7075.0
2020-01-04	3144.0	NaN	6155.0	7917.0	2524.0
2020-01-05	938.0	7771.0	NaN	7837.0	2793.0

Вот и все! Мы агрегировали сумму выручки, организовав ее по датам на оси строк и торговым агентам по оси столбцов. Обратите внимание на наличие `NaN` в наборе данных. `NaN` означает, что в `sales` для данного торгового агента нет

строк со значением в столбце **Revenue** на конкретную дату. Библиотека **pandas** заполняет пропуски значениями **NaN**. А наличие **NaN** инициирует приведение типа: из целых чисел в числа с плавающей точкой.

Для замены всех **NaN** сводной таблицы на конкретное значение можно воспользоваться параметром **fill_value**. Заполним пропуски в данных нулями:

```
In [9] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = "Revenue",
        aggfunc = "sum",
        fill_value = 0
    )
```

Out [9]

Name	Creed	Dwight	Jim	Michael	Oscar
Date					
2020-01-01	4430	2639	1864	7172	9656
2020-01-02	13214	0	8278	6362	8661
2020-01-03	0	11912	4226	5982	7075
2020-01-04	3144	0	6155	7917	2524
2020-01-05	938	7771	0	7837	2793

Интересно будет также взглянуть на промежуточные итоги для всех сочетаний дат и агентов. Для вывода промежуточных итогов по всем строкам и столбцам можно задать аргумент **True** для параметра **margins**:

```
In [10] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = "Revenue",
        aggfunc = "sum",
        fill_value = 0,
        margins = True
    )
```

Out [10]

Name	Creed	Dwight	Jim	Michael	Oscar	All
Date						
2020-01-01 00:00:00	4430	2639	1864	7172	9656	25761
2020-01-02 00:00:00	13214	0	8278	6362	8661	36515
2020-01-03 00:00:00	0	11912	4226	5982	7075	29195
2020-01-04 00:00:00	3144	0	6155	7917	2524	19740
2020-01-05 00:00:00	938	7771	0	7837	2793	19339
All	21726	22322	20523	35270	30709	130550

Обратите внимание, как появление `All` в списке меток строк меняет визуальное представление дат, которые теперь включают часы, минуты и секунды. Библиотеке pandas теперь нужно поддерживать метки индекса в виде даты одновременно в виде строковых (текстовых) значений. А строковое значение — единственный тип данных, который может выражать как дату, так и текст. Поэтому библиотека pandas преобразует индекс из типа `DatetimeIndex`, предназначенного для дат, в обычный `Index` для строковых значений. При преобразовании объекта даты/времени в строковое представление библиотека pandas включает в это представление также и время, предполагая, что дата без времени соответствует началу суток.

Метки промежуточных итогов можно настроить нужным образом с помощью параметра `margins_name`. В следующем примере мы меняем метки с `All` на `Total`:

```
In [11] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = "Revenue",
        aggfunc = "sum",
        fill_value = 0,
        margins = True,
        margins_name = "Total"
    )
```

Out [11]

Name	Creed	Dwight	Jim	Michael	Oscar	Total
Date						
2020-01-01 00:00:00	4430	2639	1864	7172	9656	25761
2020-01-02 00:00:00	13214	0	8278	6362	8661	36515
2020-01-03 00:00:00	0	11912	4226	5982	7075	29195
2020-01-04 00:00:00	3144	0	6155	7917	2524	19740
2020-01-05 00:00:00	938	7771	0	7837	2793	19339
Total	21726	22322	20523	35270	30709	130550

Пользователи Excel, по идее, должны чувствовать себя как рыба в воде при работе с этими возможностями.

8.2.2. Дополнительные возможности для работы со сводными таблицами

Сводные таблицы поддерживают множество разнообразных операций агрегирования. Представьте себе, что нас интересует число закрытых сделок по дням. Можно передать аргумент `"count"` для параметра `aggfunc`, чтобы подсчитать число строк объекта `sales` для каждого сочетания даты и сотрудника:

```
In [12] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = "Revenue",
        aggfunc = "count"
    )
```

Out [12]

Name	Creed	Dwight	Jim	Michael	Oscar
Date					
2020-01-01	1.0	1.0	1.0	1.0	2.0
2020-01-02	2.0	NaN	1.0	1.0	1.0
2020-01-03	NaN	3.0	1.0	1.0	1.0
2020-01-04	1.0	NaN	2.0	1.0	1.0
2020-01-05	1.0	1.0	NaN	1.0	1.0

Опять же значение NaN указывает, что конкретный агент не закрыл ни одной сделки в соответствующую дату. Например, Крид не закрыл ни одной сделки 2020-01-03, а Дуайт закрыл целых три. В табл. 8.1 приведено еще несколько возможных аргументов параметра `aggfunc`.

Таблица 8.1

Аргумент	Описание
max	Максимальное значение в группе
min	Минимальное значение в группе
std	Стандартное отклонение значений в группе
median	Медиана (средняя точка) значений в группе
size	Число значений в группе (эквивалентно count)

В параметр `aggfunc` метода `pivot_table` можно передать даже список агрегирующих функций. При этом сводная таблица создаст мультииндекс по оси столбцов, а группы будут находиться на самом внешнем ее уровне. В следующем примере мы агрегируем сумму выручки по датам и количество сделок по датам:

```
In [13] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = "Revenue",
        aggfunc = ["sum", "count"],
        fill_value = 0
    )
```

Out [13]

Name Date	sum			count					
	Creed	Dwight	Jim Michael	Oscar	Creed	Dwight	Jim Michael	Oscar	
2020-01-01	4430	2639	1864	7172	9656	1	1	1	2
2020-01-02	13214	0	8278	6362	8661	2	0	1	1
2020-01-03	0	11912	4226	5982	7075	0	3	1	1
2020-01-04	3144	0	6155	7917	2524	1	0	2	1
2020-01-05	938	7771	0	7837	2793	1	1	0	1

Можно произвести над различными столбцами различные операции агрегирования, передав в параметр `aggfunc` ассоциативный массив. Ключи ассоциативного массива должны соответствовать столбцам объекта `DataFrame`, а значения — задавать операции агрегирования над столбцами. В следующем примере мы извлекаем минимальную выручку и максимальные издержки для каждого сочетания даты и торгового агента:

```
In [14] sales.pivot_table(
        index = "Date",
        columns = "Name",
        values = ["Revenue", "Expenses"],
        fill_value = 0,
        aggfunc = { "Revenue": "min", "Expenses": "max" }
    )
```

Out [14]

Name Date	Expenses					Revenue				
	Creed	Dwight	Jim Michael	Oscar		Creed	Dwight	Jim Michael	Oscar	
20...	548	368	1305	412	661	4430	2639	1864	7172	4406
20...	1906	0	462	685	1401	5188	0	8278	6362	8661
20...	0	1321	1923	1772	906	0	2703	4226	5982	7075
20...	1314	0	1889	1857	1767	3144	0	2287	7917	2524
20...	1053	1475	0	1633	624	938	7771	0	7837	2793

Можно также собрать несколько группировок на одной оси, передав в параметр `index` список столбцов. В следующем примере мы агрегируем сумму издержек по торговым агентам и датам на оси строк. Библиотека pandas возвращает объект `DataFrame` с двухуровневым мультииндексом:

```
In [15] sales.pivot_table(
        index = ["Name", "Date"], values = "Revenue", aggfunc = "sum"
    ).head(10)
```

Out [15]

		Revenue	
Name	Date		
Creed	2020-01-01	4430	
	2020-01-02	13214	
	2020-01-04	3144	

	2020-01-05	938
Dwight	2020-01-01	2639
	2020-01-03	11912
	2020-01-05	7771
Jim	2020-01-01	1864
	2020-01-02	8278
	2020-01-03	4226

Для перестановки уровней в мультииндексе сводной таблицы можно поменять местами строковые значения в списке `index`. Например, ниже `Name` и `Date` меняются местами:

```
In [16] sales.pivot_table(
        index = ["Date", "Name"], values = "Revenue", aggfunc = "sum"
    ).head(10)
```

Out [16]

Date	Name	Revenue
2020-01-01	Creed	4430
	Dwight	2639
	Jim	1864
	Michael	7172
	Oscar	9656
2020-01-02	Creed	13214
	Jim	8278
	Michael	6362
	Oscar	8661
2020-01-03	Dwight	11912

Сводная таблица сначала группирует и сортирует значения `Date`, а затем группирует и сортирует значения `Name` в пределах каждой даты.

8.3. ПЕРЕНОС УРОВНЕЙ ИНДЕКСОВ С ОСИ СТОЛБЦОВ НА Ось СТРОК И НАОБОРОТ

Напомню, как выглядит сейчас объект `sales`:

```
In [17] sales.head()
```

Out [17]

	Date	Name	Customer	Revenue	Expenses
0	2020-01-01	Oscar	Logistics XYZ	5250	531
1	2020-01-01	Oscar	Money Corp.	4406	661
2	2020-01-02	Oscar	PaperMaven	8661	1401
3	2020-01-03	Oscar	PaperGenius	7075	906
4	2020-01-04	Oscar	Paper Pound	2524	1767

272 Часть II. Библиотека pandas на практике

Создадим сводную таблицу продаж, сгруппировав выручку по имени сотрудника и дате. Даты мы разместим по оси столбцов, а имена — по оси строк:

```
In [18] by_name_and_date = sales.pivot_table(
        index = "Name",
        columns = "Date",
        values = "Revenue",
        aggfunc = "sum"
    )
    by_name_and_date.head(2)
```

Out [18]

Date	2020-01-01	2020-01-02	2020-01-03	2020-01-04	2020-01-05
Creed	4430.0	13214.0	NaN	3144.0	938.0
Dwight	2639.0	NaN	11912.0	NaN	7771.0

Иногда бывает нужно перенести уровень индекса с одной оси на другую, представить данные в другом виде и затем выбрать, какое представление нам лучше подходит.

Метод `stack` переносит уровень индекса с оси столбцов на ось строк. В следующем примере мы переносим уровень индекса `Date` с оси столбцов на ось строк. Библиотека `pandas` создает объект `MultiIndex` для хранения двух уровней по строкам: `Name` и `Date`. А поскольку в результате остается только один столбец значений, библиотека `pandas` возвращает объект `Series`:

```
In [19] by_name_and_date.stack().head(7)
```

Out [19]

Name	Date	
Creed	2020-01-01	4430.0
	2020-01-02	13214.0
	2020-01-04	3144.0
	2020-01-05	938.0
	2020-01-03	NaN
Dwight	2020-01-01	2639.0
	2020-01-03	11912.0
	2020-01-05	7771.0

dtype: float64

Обратите внимание, что значения `NaN` объекта `DataFrame` в объекте `Series` отсутствуют. Библиотека `pandas` оставляла `NaN` в сводной таблице `by_name_and_date` ради структурной целостности строк и столбцов. А форма нового мультииндексного объекта `Series` позволяет `pandas` все эти значения `NaN` отбросить.

Дополняющий `stack` метод `unstack` переносит уровень индекса с оси строк на ось столбцов. Рассмотрим следующую сводную таблицу, в которой выручка

сгруппирована по покупателям и торговым агентам. На оси строк мультииндекс двухуровневый, а на оси столбцов — обычный:

```
In [20] sales_by_customer = sales.pivot_table(
        index = ["Customer", "Name"],
        values = "Revenue",
        aggfunc = "sum"
    )
    sales_by_customer.head()
```

Out [20]

		Revenue
Customer	Name	
Average Paper Co.	Creed	13214
	Jim	2287
Best Paper Co.	Dwight	2703
	Michael	15754
Logistics XYZ	Dwight	9209

Метод `unstack` переносит внутренний уровень индекса строк в индекс столбцов:

```
In [21] sales_by_customer.unstack()
```

Out [21]

Name Customer	Revenue				
	Creed	Dwight	Jim	Michael	Oscar
Average Paper Co.	13214.0	NaN	2287.0	NaN	NaN
Best Paper Co.	NaN	2703.0	NaN	15754.0	NaN
Logistics XYZ	NaN	9209.0	NaN	7172.0	5250.0
Money Corp.	5368.0	NaN	8278.0	NaN	4406.0
Paper Pound	NaN	7771.0	4226.0	NaN	5317.0
PaperGenius	NaN	2639.0	1864.0	12344.0	7075.0
PaperMaven	3144.0	NaN	3868.0	NaN	8661.0

В этом новом объекте `DataFrame` мультииндекс оси столбцов двухуровневый, а на оси строк — обычный, одноуровневый.

8.4. РАСПЛАВЛЕНИЕ НАБОРА ДАННЫХ

Сводная таблица агрегирует значения набора данных. В этом разделе вы научитесь делать противоположное: производить декомпозицию агрегированного набора данных в неагрегированный.

Применим нашу концептуальную схему «широкие/узкие данные» к объекту `DataFrame` `sales`. Эффективная стратегия определения того, узкий ли формат

данных, такова: пройти по одной строке значений, выясняя, является ли значение каждой ячейки отдельным наблюдением переменной, описываемой заголовком столбца. Вот, например, первая строка `sales`:

```
In [22] sales.head(1)
```

```
Out [22]
```

	Date	Name	Customer	Revenue	Expenses
0	2020-01-01	Oscar	Logistics XYZ	5250	531

В этом примере `2020-01-01` — дата, `Oscar` — имя агента, `Logistics XYZ` — покупатель, `5250` — размер выручки, а `531` — размер издержек. Объект `DataFrame sales` — пример узкого набора данных. Каждое значение строки соответствует отдельному наблюдению заданной переменной. Ни одна переменная не повторяется в разных столбцах.

При операциях над данными в широком или узком формате нередко приходится выбирать между гибкостью и удобочитаемостью. Последние четыре столбца (`Name`, `Customer`, `Revenue` и `Expenses`) можно представить в виде полей одного столбца `Category` (следующий пример), но никакой реальной выгоды от этого не будет, поскольку эти четыре переменные совершенно отдельные. Агрегировать данные в формате вроде следующего можно только с потерей удобства восприятия и анализа данных:

	Date	Category	Value
0	2020-01-01	Name	Oscar
1	2020-01-01	Customer	Logistics XYZ
2	2020-01-01	Revenue	5250
3	2020-01-01	Expenses	531

Следующий набор данных, `video_game_sales.csv`, представляет собой список местных продаж для более чем 16 000 компьютерных игр. Каждая из строк включает название игры, а также количество проданных единиц товара (в миллионах) в Северной Америке (`NA`), Европе (`EU`), Японии (`JP`) и других (`Other`) регионах:

```
In [23] video_game_sales = pd.read_csv("video_game_sales.csv")
        video_game_sales.head()
```

```
Out [23]
```

	Name	NA	EU	JP	Other
0	Wii Sports	41.49	29.02	3.77	8.46
1	Super Mario Bros.	29.08	3.58	6.81	0.77
2	Mario Kart Wii	15.85	12.88	3.79	3.31
3	Wii Sports Resort	15.75	11.01	3.28	2.96
4	Pokemon Red/Poke...	11.27	8.89	10.22	1.00

Опять же пройдем по одной строке и посмотрим, содержит ли каждая ее ячейка правильную информацию. Скажем, вот первая строка `video_game_sales`:

```
In [24] video_game_sales.head(1)
```

```
Out [24]
```

	Name	NA	EU	JP	Other
0	Wii Sports	41.49	29.02	3.77	8.46

С первой ячейкой все в порядке. `Wii Sports` — разновидность `Name`. Со следующими четырьмя ячейками сложнее. Значение `41.49` — отнюдь не смысловая разновидность/наблюдение `NA`. `NA` (Северная Америка) не является переменной величиной, значения которой меняются от одной ячейки столбца к другой. Переменными данными в столбце `NA` являются числовые значения по продажам в регионе `NA`. А сама `NA` — совершенно отдельная сущность.

Таким образом, данные в наборе `video_game_sales` хранятся в широком формате. В четырех столбцах (`NA`, `EU`, `JP` и `Other`) хранится одна и та же смысловая часть данных: количество проданных единиц товара в регионе. Если добавить еще столбцы для продаж по регионам, набор данных будет расти горизонтально. Если можно сгруппировать несколько заголовков столбцов в одну категорию, значит, данные в наборе хранятся в широком формате.

Перенесем значения `NA`, `EU`, `JP` и `Other` в новую категорию `Region`. Сравните предыдущее представление со следующим:

	Name	Region	Sales
0	Wii Sports	NA	41.49
1	Wii Sports	EU	29.02
2	Wii Sports	JP	3.77
3	Wii Sports	Other	8.46

В каком-то смысле мы производим над объектом `DataFrame` `video_game_sales` операцию, обратную созданию сводной таблицы. Мы преобразуем агрегированное итоговое представление данных в представление, где в каждом столбце хранится одна переменная часть информации.

Для расплавления объектов `DataFrame` в библиотеке `pandas` служит метод `melt` (*расплавлением* (`melting`)) называется процесс преобразования широкого набора данных в узкий). Этот метод принимает два основных параметра.

- Параметр `id_vars` задает столбец-идентификатор, по которому агрегируются данные широкого набора. Столбец-идентификатор набора `video_game_sales` — `Name`. То есть агрегируются продажи по каждой компьютерной игре.

- Параметр `value_vars` принимает столбец (-ы), значения которого библиотека pandas расплавляет и сохраняет в новом столбце.

Начнем с самого простого случая — расплавления значений одного только столбца `NA`. В следующем примере библиотека pandas проходит в цикле по всем значениям столбца `NA`, создавая для каждого отдельную строку в новом объекте `DataFrame`. Pandas сохраняет предшествующее название столбца (`NA`) в отдельном столбце `variable`:

```
In [25] video_game_sales.melt(id_vars = "Name", value_vars = "NA").head()
```

```
Out [25]
```

	Name	variable	value
0	Wii Sports	NA	41.49
1	Super Mario Bros.	NA	29.08
2	Mario Kart Wii	NA	15.85
3	Wii Sports Resort	NA	15.75
4	Pokemon Red/Pokemon Blue	NA	11.27

Теперь расплавим все четыре столбца продаж по регионам. Во фрагменте кода ниже мы передаем в параметр `value_vars` список всех четырех столбцов продаж по регионам из `video_game_sales`:

```
In [26] regional_sales_columns = ["NA", "EU", "JP", "Other"]
        video_game_sales.melt(
            id_vars = "Name", value_vars = regional_sales_columns
        )
```

```
Out [26]
```

	Name	variable	value
0	Wii Sports	NA	41.49
1	Super Mario Bros.	NA	29.08
2	Mario Kart Wii	NA	15.85
3	Wii Sports Resort	NA	15.75
4	Pokemon Red/Pokemon Blue	NA	11.27
...
66259	Woody Woodpecker in Crazy Castle 5	Other	0.00
66260	Men in Black II: Alien Escape	Other	0.00
66261	SCORE International Baja 1000: The Official Game	Other	0.00
66262	Know How 2	Other	0.00
66263	Spirits & Spells	Other	0.00

```
66264 rows x 3 columns
```

Метод `melt` вернул нам объект `DataFrame` из 66 264 строк! Для сравнения: `video_game_sales` содержал только 16 566 строк. Новый набор данных в четыре раза длиннее, поскольку на каждую строку `video_game_sales` в нем приходится четыре строки данных. Этот набор данных содержит:

- 16 566 строк для компьютерных игр и соответствующих им объемов продаж по региону `NA`;
- 16 566 строк для компьютерных игр и соответствующих им объемов продаж по региону `EU`;
- 16 566 строк для компьютерных игр и соответствующих им объемов продаж по региону `JP`;
- 16 566 строк для компьютерных игр и соответствующих им объемов продаж по региону `Other`.

Столбец `variable` содержит четыре названия столбцов регионов из `video_game_sales`. Столбец `value` содержит показатели продаж из этих четырех столбцов для различных регионов. И это в то время, как в предыдущих результатах данные показывали, что в столбце `Other` файла `video_game_sales` для компьютерной игры `Woody Woodpecker in Crazy Castle 5` содержалось значение `0.00`.

Можно задать свои названия расплавленных столбцов объекта `DataFrame`, передав соответствующие аргументы для параметров `var_name` и `value_name`. В следующем примере для столбца `variable` взято название `Region`, а для столбца `value` — `Sales`:

```
In [27] video_game_sales_by_region = video_game_sales.melt(
        id_vars = "Name",
        value_vars = regional_sales_columns,
        var_name = "Region",
        value_name = "Sales"
    )
    video_game_sales_by_region.head()
```

Out [27]

	Name	Region	Sales
0	Wii Sports	NA	41.49
1	Super Mario Bros.	NA	29.08
2	Mario Kart Wii	NA	15.85
3	Wii Sports Resort	NA	15.75
4	Pokemon Red/Pokemon Blue	NA	11.27

Узкие данные легче агрегировать, чем широкие. Допустим, нам нужно узнать сумму продаж каждой компьютерной игры по всем регионам. Воспользовавшись

расплавленным набором данных, можно решить эту задачу при помощи всего нескольких строк кода с помощью метода `pivot_table`:

```
In [28] video_game_sales_by_region.pivot_table(
        index = "Name", values = "Sales", aggfunc = "sum"
    ).head()
```

```
Out [28]
```

	Sales
Name	
-----	-----
'98 Koshien	0.40
.hack//G.U. Vol.1//Rebirth	0.17
.hack//G.U. Vol.2//Reminisce	0.23
.hack//G.U. Vol.3//Redemption	0.17
.hack//Infection Part 1	1.26

Узкий формат набора данных упрощает процесс создания сводной таблицы.

8.5. РАЗВЕРТЫВАНИЕ СПИСКА ЗНАЧЕНИЙ

Иногда в наборе данных одна ячейка может содержать много значений (кластер). И может потребоваться разбить такой кластер данных, чтобы в каждой строке содержалось одно значение. Рассмотрим набор данных `recipes.csv`, содержащий три рецепта, с названиями и списками ингредиентов. Ингредиенты хранятся в строковых значениях и разделены запятыми:

```
In [29] recipes = pd.read_csv("recipes.csv")
        recipes
```

```
Out [29]
```

	Recipe	Ingredients
0	Cashew Crusted Chicken	Apricot preserves, Dijon mustard, cu...
1	Tomato Basil Salmon	Salmon filets, basil, tomato, olive ...
2	Parmesan Cheese Chicken	Bread crumbs, Parmesan cheese, Itali...

Помните метод `str.split`, с которым вы познакомились в главе 6? Он позволяет разбить строковое значение на подстроки по заданному разделителю. Например, можно разбить строковые значения из столбца `Ingredients` по запятым. Приведу пример кода, в котором библиотека pandas возвращает объект `Series` со списками, в каждом из которых содержатся ингредиенты из соответствующей строки:

```
In [30] recipes["Ingredients"].str.split(",")
```

```
Out [30]
```

```
0 [Apricot preserves, Dijon mustard, curry pow...
1 [Salmon filets, basil, tomato, olive oil, ...
2 [Bread crumbs, Parmesan cheese, Italian seas...
Name: Ingredients, dtype: object
```

Перепишем исходный столбец `Ingredients` новым:

```
In [31] recipes["Ingredients"] = recipes["Ingredients"].str.split(",")
       recipes
```

```
Out [31]
```

	Recipe	Ingredients
0	Cashew Crusted Chicken	[Apricot preserves, Dijon mustard, ...
1	Tomato Basil Salmon	[Salmon filets, basil, tomato, ol...
2	Parmesan Cheese Chicken	[Bread crumbs, Parmesan cheese, It...

А как теперь развернуть значения из каждого списка на несколько строк? Метод `explode` создает по отдельной строке для каждого элемента списка в объекте `Series`. Вызовем этот метод для нашего объекта `DataFrame`, передав в него столбец со списками в качестве аргумента:

```
In [32] recipes.explode("Ingredients")
```

```
Out [32]
```

	Recipe	Ingredients
0	Cashew Crusted Chicken	Apricot preserves
0	Cashew Crusted Chicken	Dijon mustard
0	Cashew Crusted Chicken	curry powder
0	Cashew Crusted Chicken	chicken breasts
0	Cashew Crusted Chicken	cashews
1	Tomato Basil Salmon	Salmon filets
1	Tomato Basil Salmon	basil
1	Tomato Basil Salmon	tomato
1	Tomato Basil Salmon	olive oil
1	Tomato Basil Salmon	Parmesan cheese
2	Simply Parmesan Cheese	Bread crumbs
2	Simply Parmesan Cheese	Parmesan cheese
2	Simply Parmesan Cheese	Italian seasoning
2	Simply Parmesan Cheese	egg
2	Simply Parmesan Cheese	chicken breasts

Замечательно! Мы разнесли все ингредиенты по отдельным строчкам. Обратите внимание, что для работы метода `explode` требуется объект `Series` со списками.

8.6. УПРАЖНЕНИЯ

А вот и ваш шанс попрактиковаться в изменении формы представления данных, создании сводных таблиц и расплавлении — возможностях, о которых вы узнали из этой главы.

8.6.1. Задачи

У нас есть два набора данных, с которыми вы можете экспериментировать. `used_cars.csv` — перечень поддержанных автомобилей, продаваемых на сайте частных объявлений **Craigslist**. Каждая строка включает производителя машины, год выпуска, тип топлива, тип коробки передач и цену:

```
In [33] cars = pd.read_csv("used_cars.csv")
        cars.head()
```

Out [33]

	Manufacturer	Year	Fuel	Transmission	Price
0	Acura	2012	Gas	Automatic	10299
1	Jaguar	2011	Gas	Automatic	9500
2	Honda	2004	Gas	Automatic	3995
3	Chevrolet	2016	Gas	Automatic	41988
4	Kia	2015	Gas	Automatic	12995

Набор данных `minimum_wage.csv` представляет собой набор минимальных зарплат по штатам США. Он включает столбец `State` и несколько столбцов, соответствующих различным годам:

```
In [34] min_wage = pd.read_csv("minimum_wage.csv")
        min_wage.head()
```

Out [34]

	State	2010	2011	2012	2013	2014	2015	2016	2017
0	Alabama	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	Alaska	8.90	8.63	8.45	8.33	8.20	9.24	10.17	10.01
2	Arizona	8.33	8.18	8.34	8.38	8.36	8.50	8.40	10.22
3	Arkansas	7.18	6.96	6.82	6.72	6.61	7.92	8.35	8.68
4	California	9.19	8.91	8.72	8.60	9.52	9.51	10.43	10.22

А ниже даны ваши задачи.

1. Агрегируйте сумму цен машин в `cars`. Сгруппируйте результаты по типу топлива по оси строк.
2. Агрегируйте количество машин в `cars`. Сгруппируйте результаты по производителю на оси строк и типу коробки передач на оси столбцов.

3. Агрегируйте среднюю цену машин в `cars`. Сгруппируйте результаты по году и типу топлива на оси строк и типу коробки передач на оси столбцов.
4. В объекте `DataFrame` из предыдущей задачи перенесите уровень `Transmission` с оси столбцов на ось строк.
5. Преобразуйте объект `min_wage` из широкого формата в узкий. Другими словами, перенесите данные из восьми столбцов годов (с 2010 по 2017 год) в один столбец.

8.6.2. Решения

Давайте решать задачи в порядке очередности.

1. Оптимальное решение для суммирования значений из столбца `Price` и группировки итоговых сумм по типу топлива — метод `pivot_table`. Для задания меток сводной таблицы можно воспользоваться параметром `index`, в который мы передадим аргумент `"Fuel"`. А с помощью параметра `aggfunc` зададим операцию агрегирования — `"sum"`:

```
In [35] cars.pivot_table(
        values = "Price", index = "Fuel", aggfunc = "sum"
    )
```

Out [35]

Fuel	Price
Diesel	986177143
Electric	18502957
Gas	86203853926
Hybrid	44926064
Other	242096286

2. Для подсчета количества машин по производителю и типу коробки передач также можно воспользоваться методом `pivot_table`. Чтобы выбрать в качестве меток столбцов сводной таблицы значения столбца `Transmission`, воспользуемся параметром `columns`. Не забудьте задать аргумент `True` параметра `margins` для отображения промежуточных итогов по строкам и столбцам:

```
In [36] cars.pivot_table(
        values = "Price",
        index = "Manufacturer",
        columns = "Transmission",
        aggfunc = "count",
        margins = True
    ).tail()
```

Out [36]

Transmission Manufacturer	Automatic	Manual	Other	All
Tesla	179.0	NaN	59.0	238
Toyota	31480.0	1367.0	2134.0	34981
Volkswagen	7985.0	1286.0	236.0	9507
Volvo	2665.0	155.0	50.0	2870
All	398428.0	21005.0	21738.0	441171

3. Для группировки средних цен машин по году и типу топлива на оси строк сводной таблицы можно передать список строковых значений в параметр `index` функции `pivot_table`:

```
In [37] cars.pivot_table(
        values = "Price",
        index = ["Year", "Fuel"],
        columns = ["Transmission"],
        aggfunc = "mean"
    )
```

Out [37]

Transmission Year Fuel	Automatic	Manual	Other
2000 Diesel	11326.176962	14010.164021	11075.000000
Electric	1500.000000	NaN	NaN
Gas	4314.675996	6226.140327	3203.538462
Hybrid	2600.000000	2400.000000	NaN
Other	16014.918919	11361.952381	12984.642857
...
2020 Diesel	63272.595930	1.000000	1234.000000
Electric	8015.166667	2200.000000	20247.500000
Gas	34925.857933	36007.270833	20971.045455
Hybrid	35753.200000	NaN	1234.000000
Other	22210.306452	NaN	2725.925926

102 rows × 3 columns

Присвоим предыдущую сводную таблицу переменной `report`. Она пригодится нам для следующей задачи:

```
In [38] report = cars.pivot_table(
        values = "Price",
        index = ["Year", "Fuel"],
        columns = ["Transmission"],
        aggfunc = "mean"
    )
```

4. Следующее упражнение состоит в переносе типа коробки передач из индекса по столбцам в индекс по строкам. Это можно сделать с помощью метода `stack`.

Он возвращает мультииндексный объект `Series`, включающий три уровня: `Year`, `Fuel` и только что добавленный `Transmission`:

```
In [39] report.stack()
```

```
Out [39]
```

```
Year  Fuel      Transmission
2000  Diesel    Automatic      11326.176962
      Diesel    Manual        14010.164021
      Diesel    Other         11075.000000
      Electric Automatic      1500.000000
      Gas      Automatic      4314.675996
      ...
2020  Gas      Other         20971.045455
      Hybrid   Automatic      35753.200000
      Hybrid   Other          1234.000000
      Other    Automatic      22210.306452
      Other    Other          2725.925926
Length: 274, dtype: float64
```

- Пришла очередь преобразовать набор данных `min_wage` из широкого формата в узкий. В восьми его столбцах содержится одна и та же переменная: сами минимальные зарплаты. Подходящее решение этой задачи — использование метода `melt`. Роль столбца-идентификатора будет играть столбец `State`, а восемь столбцов для различных лет пополнят своими значениями столбец переменных:

```
In [40] year_columns = [
        "2010", "2011", "2012", "2013",
        "2014", "2015", "2016", "2017"
    ]
min_wage.melt(id_vars = "State", value_vars = year_columns)
```

```
Out [40]
```

	State	variable	value
0	Alabama	2010	0.00
1	Alaska	2010	8.90
2	Arizona	2010	8.33
3	Arkansas	2010	7.18
4	California	2010	9.19
...
435	Virginia	2017	7.41
436	Washington	2017	11.24
437	West Virginia	2017	8.94
438	Wisconsin	2017	7.41
439	Wyoming	2017	5.26

```
440 rows x 3 columns
```

Небольшой совет в качестве бонуса: можно убрать параметр `value_vars` из вызова метода `melt`, и все равно получится тот же объект `DataFrame`. По умолчанию библиотека `pandas` расплавляет данные из всех столбцов, кроме переданного в параметре `id_vars`:

```
In [41] min_wage.melt(id_vars = "State")
```

```
Out [41]
```

	State	variable	value
0	Alabama	2010	0.00
1	Alaska	2010	8.90
2	Arizona	2010	8.33
3	Arkansas	2010	7.18
4	California	2010	9.19
...
435	Virginia	2017	7.41
436	Washington	2017	11.24
437	West Virginia	2017	8.94
438	Wisconsin	2017	7.41
439	Wyoming	2017	5.26

```
440 rows x 3 columns
```

Можно также задать свои названия для столбцов с помощью параметров `var_name` и `value_name`. Например, для предыдущего набора используем названия `"Year"` и `"Wage"`, чтобы понятнее было, что за значения содержит каждый из столбцов:

```
In [42] min_wage.melt(
        id_vars = "State", var_name = "Year", value_name = "Wage"
    )
```

```
Out [42]
```

	State	Year	Wage
0	Alabama	2010	0.00
1	Alaska	2010	8.90
2	Arizona	2010	8.33
3	Arkansas	2010	7.18
4	California	2010	9.19
...
435	Virginia	2017	7.41
436	Washington	2017	11.24
437	West Virginia	2017	8.94
438	Wisconsin	2017	7.41
439	Wyoming	2017	5.26

```
440 rows x 3 columns
```

Поздравляю! Упражнения выполнены!

РЕЗЮМЕ

- Метод `pivot_table` служит для агрегирования данных объекта `DataFrame`.
- Операции агрегирования сводных таблиц включают суммирование, подсчет количества и среднее значение.
- Можно задавать пользовательские метки столбцов и строк сводной таблицы.
- Значения одного или нескольких столбцов можно использовать в качестве меток индекса сводной таблицы.
- Метод `stack` переносит уровень индекса из столбцов в строки.
- Метод `unstack` переносит уровень индекса из строк в столбцы.
- Метод `melt` производит операцию, обратную созданию сводной таблицы, «расплавляя» — распределяя — данные агрегированной таблицы по отдельным строкам. Этот процесс приводит к преобразованию широкого набора данных в узкий.
- Метод `explode` создает по отдельной строке для каждого элемента списка; он требует на входе объекта `Series` со списками.

Объект `GroupBy`

В этой главе

- ✓ Группировка содержимого `DataFrame` с помощью метода `groupby`.
- ✓ Извлечение первой и последней строк в группах в объекте `GroupBy`.
- ✓ Выполнение агрегатных операций над группами в `GroupBy`.
- ✓ Итерации по экземплярам `DataFrames` в объекте `GroupBy`.

Объект `GroupBy` в библиотеке `pandas` — это контейнер, предназначенный для группировки записей из набора данных `DataFrame`. Он представляет собой набор методов для агрегирования и анализа каждой независимой группы в коллекции, позволяет извлекать из каждой группы записи с определенными индексами, а также предлагает удобный способ выполнения итераций по группам записей. В объекте `GroupBy` заключено много возможностей, поэтому рассмотрим подробнее, на что он способен.

Для описания категорий, в которые попадают одна или несколько записей, мы можем использовать такие взаимозаменяемые термины, как «*группы*», «*корзины*» и «*кластеры*».

9.1. СОЗДАНИЕ ОБЪЕКТА GROUPBY С НУЛЯ

Создадим новый блокнот Jupyter и импортируем библиотеку pandas:

```
In [1] import pandas as pd
```

Для начала рассмотрим небольшой пример, а затем, в разделе 9.2, углубимся в технические детали. Начнем с создания объекта **DataFrame**, хранящего цены на фрукты и овощи в супермаркете:

```
In [2] food_data = {
    "Item": ["Banana", "Cucumber", "Orange", "Tomato", "Watermelon"],
    "Type": ["Fruit", "Vegetable", "Fruit", "Vegetable", "Fruit"],
    "Price": [0.99, 1.25, 0.25, 0.33, 3.00]
}

supermarket = pd.DataFrame(data = food_data)

supermarket
```

```
Out [2]
```

	Item	Type	Price
0	Banana	Fruit	0.99
1	Cucumber	Vegetable	1.25
2	Orange	Fruit	0.25
3	Tomato	Vegetable	0.33
4	Watermelon	Fruit	3.00

Столбец **Type** (Тип) определяет группу, к которой принадлежит товар (столбец **Item**). В наборе данных супермаркета есть два типа, то есть две группы товаров: фрукты (**Fruit**) и овощи (**Vegetable**).

Объект **GroupBy** организует записи в **DataFrame** в группы на основе значений в некотором столбце. Предположим, что нас интересует средняя цена фруктов и средняя цена овощей. Если бы мы могли разделить по группам записи со значениями **"Fruit"** и **"Vegetable"** в столбце **Type**, то легко справились бы с вычислениями необходимых величин.

Начнем с вызова метода **groupby** объекта **DataFrame**, представляющего супермаркет. Этому методу нужно передать столбец, на основе которого pandas выполнит группировку. Роль такого столбца в следующем примере играет столбец **Type**.

Метод возвращает объект, с которым в этой книге мы еще не сталкивались: `DataFrameGroupBy`. Он очевидно отличается от `DataFrame`:

```
In [3] groups = supermarket.groupby("Type")
      groups
```

```
Out [3] <pandas.core.groupby.generic.DataFrameGroupBy object at
      0x114f2db90>
```

Столбец `Type` имеет два уникальных значения, поэтому объект `GroupBy` будет хранить две группы. Метод `get_group` принимает имя группы и возвращает `DataFrame` с соответствующими записями. Извлечем записи, принадлежащие группе `Fruit`:

```
In [4] groups.get_group("Fruit")
```

```
Out [4]
```

	Item	Type	Price
0	Banana	Fruit	0.99
2	Orange	Fruit	0.25
4	Watermelon	Fruit	3.00

Аналогично можно извлечь записи, относящиеся к группе `Vegetable`:

```
In [5] groups.get_group("Vegetable")
```

```
Out [5]
```

	Item	Type	Price
1	Cucumber	Vegetable	1.25
3	Tomato	Vegetable	0.33

Объект `GroupBy` незаменим в агрегатных операциях. Наша первоначальная задача — вычислить среднюю цену фруктов и овощей в супермаркете. Мы можем вызвать метод `mean` объекта `groups` и получить среднюю цену товаров в каждой группе. Итак, с помощью нескольких строк кода мы успешно разбили, агрегировали и проанализировали набор данных:

```
In [6] groups.mean()
```

```
Out [6]
```

Type	Price
Fruit	1.413333
Vegetable	0.790000

Теперь, получив базовое представление о группах, перейдем к более сложному набору данных.

9.2. СОЗДАНИЕ ОБЪЕКТА GROUPBY ИЗ НАБОРА ДАННЫХ

Fortune 1000 — это список 1000 крупнейших компаний США по объему выручки. Список ежегодно обновляется деловым журналом *Fortune*. Файл `Fortune1000.csv` содержит перечень компаний из списка Fortune 1000 за 2018 год. Каждая запись содержит название компании, объем выручки, размер прибыли, количество сотрудников, сектор и отрасль:

```
In [7] fortune = pd.read_csv("fortune1000.csv")
      fortune
```

```
Out [7]
```

	Company	Revenues	Profits	Employees	Sector	Industry

0	Walmart	500343.0	9862.0	2300000	Retailing	General M...
1	Exxon Mobil	244363.0	19710.0	71200	Energy	Petroleum...
2	Berkshire...	242137.0	44940.0	377000	Financials	Insurance...
3	Apple	229234.0	48351.0	123000	Technology	Computers...
4	UnitedHea...	201159.0	10558.0	260000	Health Care	Health Ca...
...						
995	SiteOne L...	1862.0	54.6	3664	Wholesalers	Wholesale...
996	Charles R...	1858.0	123.4	11800	Health Care	Health Ca...
997	CoreLogic	1851.0	152.2	5900	Business ...	Financial...
998	Ensign Group	1849.0	40.5	21301	Health Care	Health Ca...
999	HCP	1848.0	414.2	190	Financials	Real estate

```
1000 rows x 6 columns
```

Сектор может включать множество компаний. Например, Apple и Amazon.com относятся к сектору **Technology** (Технология).

Отрасль (**Industry**) — это подкатегория внутри сектора. Например, отрасли **Pipelines** (Трубопроводы) и **Petroleum Refining** (Нефтепереработка) относятся к сектору **Energy** (Энергетика).

Столбец **Sector** содержит 21 уникальный сектор. Предположим, нам понадобилось определить среднюю выручку компаний в каждом секторе. Прежде чем использовать объект **GroupBy**, попробуем решить задачу, выбрав альтернативный подход. В главе 5 было показано, как создать объект **Series** с булевыми значениями для извлечения подмножества записей из **DataFrame**. Пример ниже выводит все компании со значением **"Retailing"** (Розничная торговля) в столбце **Sector**:

```
In [8] in_retailing = fortune["Sector"] == "Retailing"
      retail_companies = fortune[in_retailing]
      retail_companies.head()
```

```
Out [8]
```

	Company	Revenues	Profits	Employees	Sector	Industry
0	Walmart	500343.0	9862.0	2300000	Retailing	General Mercha...
7	Amazon.com	177866.0	3033.0	566000	Retailing	Internet Servi...
14	Costco	129025.0	2679.0	182000	Retailing	General Mercha...
22	Home Depot	100904.0	8630.0	413000	Retailing	Specialty Reta...
38	Target	71879.0	2934.0	345000	Retailing	General Mercha...

Мы можем извлечь значения из столбца **Revenues** этого подмножества, используя квадратные скобки:

```
In [9] retail_companies["Revenues"].head()
```

```
Out [9] 0    500343.0
        7    177866.0
        14   129025.0
        22   100904.0
        38    71879.0
        Name: Revenues, dtype: float64
```

И наконец, можно вычислить средний объем выручки в секторе **Retailing**, применив метод **mean** к столбцу **Revenues**:

```
In [10] retail_companies["Revenues"].mean()
```

```
Out [10] 21874.714285714286
```

Описанный сценарий прекрасно справляется с задачей вычисления среднего объема выручки в одном секторе. Однако, чтобы применить ту же логику к другим 20 секторам, придется написать много дополнительного кода. То есть проблема в том, что код предложенного сценария плохо масштабируется. Python может автоматизировать некоторые повторения, но объект **GroupBy** — все-таки лучшее решение поставленной задачи. В нем разработчики pandas уже решили проблему работы со множеством рассматриваемых групп.

Вызовем метод **groupby** объекта **fortune**. Этому методу нужно передать столбец, на основе которого pandas выполнит группировку. Столбец считается подходящим кандидатом для такой роли, если хранит категориальные данные. Желательно, чтобы к каждой категории относилось несколько записей. Например, в нашем наборе данных с 1000 уникальных компаний имеется всего 21 уникальный сектор, поэтому столбец **Sector** хорошо подходит для агрегирования:

```
In [11] sectors = fortune.groupby("Sector")
```

Выведем переменную **sectors**, чтобы посмотреть, что за объект в ней хранится:

```
In [12] sectors
```

```
Out [12] <pandas.core.groupby.generic.DataFrameGroupBy object at 0x1235b1d10>
```

Объект `DataFrameGroupBy` содержит набор экземпляров `DataFrame`. «За кулисами» выводимой информации `pandas` повторила процесс, который мы использовали для извлечения сектора `Retailing`, проведя его для каждого из 21 уникального значения в столбце `Sector`.

Мы можем подсчитать количество групп в `sectors`, передав объекту `GroupBy` встроенную функцию `len`:

```
In [13] len(sectors)
```

```
Out [13] 21
```

Объект `sectors` типа `GroupBy` содержит 21 экземпляр `DataFrame`. Это число равно количеству уникальных значений в столбце `Sector`, в чем можно убедиться, вызвав метод `nunique`:

```
In [14] fortune["Sector"].nunique()
```

```
Out [14] 21
```

Что же это за сектора и сколько компаний из списка `Fortune 1000` принадлежит каждому из них? Ответить на эти вопросы можно с помощью метода `size` объекта `GroupBy`. Он возвращает экземпляр `Series` со списком групп и количеством записей в каждой. Следующий результат сообщает нам, что 25 компаний из списка `Fortune 1000` принадлежат сектору `Aerospace & Defense` (Аэрокосмическая и оборонная промышленность), 14 принадлежат сектору `Apparel` (Швейная промышленность) и т. д.:

```
In [15] sectors.size()
```

```
Out [15] Sector
Aerospace & Defense      25
Apparel                  14
Business Services       53
Chemicals                33
Energy                  107
Engineering & Construction  27
Financials              155
Food & Drug Stores        12
Food, Beverages & Tobacco  37
Health Care              71
Hotels, Restaurants & Leisure  26
Household Products       28
Industrials              49
Materials                45
Media                    25
Motor Vehicles & Parts     19
Retailing                77
Technology               103
Telecommunications       10
Transportation           40
Wholesalers              44
dtype: int64
```

Теперь, разбив записи на группы, посмотрим, что можно сделать с объектом `GroupBy`, какому интересующему нас анализу подвергнуть данные.

9.3. АТРИБУТЫ И МЕТОДЫ ОБЪЕКТА `GROUPBY`

Чтобы сделать работу с объектом `GroupBy` более осознанной, я рекомендую представить его как словарь, отображающий 21 сектор в наборы соответствующих записей. Такая аналогия не просто визуальный образ, а вполне применимая на практике модель.

Атрибут `groups` хранит словарь с этими ассоциациями между группами и записями; его ключи — это имена секторов, а его значения — объекты `Index`, хранящие позиции записей из объекта `fortune` типа `DataFrame`. Всего в словаре 21 пара «ключ/значение», но для экономии в следующем примере я сократил вывод:

```
In [16] sectors.groups
```

```
Out [16]
```

```
'Aerospace & Defense': Int64Index([ 26,  50,  58,  98, 117, 118, 207, 224,
                                   275, 380, 404, 406, 414, 540, 660,
                                   661, 806, 829, 884, 930, 954, 955,
                                   959, 975, 988], dtype='int64'),
'Apparel': Int64Index([88, 241, 331, 420, 432, 526, 529, 554, 587, 678,
                       766, 774, 835, 861], dtype='int64'),
```

Согласно полученному выводу, строки с индексами 26, 50, 58, 98 и т. д. имеют значение "Aerospace & Defense" в столбце `Sector`.

В главе 4 мы познакомились с методом доступа `loc` для извлечения записей и столбцов из `DataFrame` по их индексам. В первом аргументе он принимает индекс записи, а во втором — индекс столбца. Можем извлечь какую-нибудь запись из `fortune`, чтобы убедиться, что pandas помещает ее в правильную группу. Попробуем получить запись с индексом 26 — первым в группе "Aerospace & Defense":

```
In [17] fortune.loc[26, "Sector"]
```

```
Out [17] 'Aerospace & Defense'
```

А если потребуется найти самую эффективную компанию (по объему выручки) в каждом секторе? В объекте `GroupBy` есть метод `first`, извлекающий первую запись в каждом секторе. Поскольку наш `DataFrame` сортируется по объему выручки, то первая компания в каждой группе и будет самой эффективной

компаний в соответствующем секторе. Метод `first` возвращает `DataFrame` с 21 записью (по одной компании в каждой группе):

```
In [18] sectors.first()
```

```
Out [18]
```

Sector	Company	Revenues	Profits	Employees	Industry

Aerospace &...	Boeing	93392.0	8197.0	140800	Aerospace ...
Apparel	Nike	34350.0	4240.0	74400	Apparel
Business Se...	ManpowerGroup	21034.0	545.4	29000	Temporary ...
Chemicals	DowDuPont	62683.0	1460.0	98000	Chemicals
Energy	Exxon Mobil	244363.0	19710.0	71200	Petroleum ...
...
Retailing	Walmart	500343.0	9862.0	2300000	General Me...
Technology	Apple	229234.0	48351.0	123000	Computers,...
Telecommuni...	AT&T	160546.0	29450.0	254000	Telecommun...
Transportation	UPS	65872.0	4910.0	346415	Mail, Pack...
Wholesalers	McKesson	198533.0	5070.0	64500	Wholesaler...

Парный ему метод `last` извлекает последнюю компанию из каждой группы. И снова `pandas` извлекает записи в том же порядке, в каком они расположены в `DataFrame`. Поскольку в наборе данных `fortune` компании отсортированы в порядке убывания выручки, метод `last` вернет компании с самым низким объемом выручки в каждом секторе:

```
In [19] sectors.last()
```

```
Out [19]
```

Sector	Company	Revenues	Profits	Employees	Industry

Aerospace &...	Aerojet Ro...	1877.0	-9.2	5157	Aerospace ...
Apparel	Wolverine ...	2350.0	0.3	3700	Apparel
Business Se...	CoreLogic	1851.0	152.2	5900	Financial ...
Chemicals	Stepan	1925.0	91.6	2096	Chemicals
Energy	Superior E...	1874.0	-205.9	6400	Oil and Ga...
...
Retailing	Childrens ...	1870.0	84.7	9800	Specialty ...
Technology	VeriFone S...	1871.0	-173.8	5600	Financial ...
Telecommuni...	Zayo Group...	2200.0	85.7	3794	Telecommun...
Transportation	Echo Globa...	1943.0	12.6	2453	Transporta...
Wholesalers	SiteOne La...	1862.0	54.6	3664	Wholesaler...

Объект `GroupBy` присваивает индексы строкам в каждой группе. Первая запись в секторе `Aerospace & Defense` имеет индекс 0 в своей группе. Аналогично первая запись в секторе `Apparel` имеет индекс 0 в своей группе. Индексы групп не связаны между собой и независимы друг от друга.

Метод `nth` извлекает запись с указанным индексом в своей группе. Если вызвать метод `nth` с аргументом `0`, он вернет первые компании из каждой группы. Следующий `DataFrame` идентичен тому, что был получен вызовом метода `first`:

```
In [20] sectors.nth(0)
```

```
Out [20]
```

Sector	Company	Revenues	Profits	Employees	Industry
Aerospace &...	Boeing	93392.0	8197.0	140800	Aerospace ...
Apparel	Nike	34350.0	4240.0	74400	Apparel
Business Se...	ManpowerGroup	21034.0	545.4	29000	Temporary ...
Chemicals	DowDuPont	62683.0	1460.0	98000	Chemicals
Energy	Exxon Mobil	244363.0	19710.0	71200	Petroleum ...
...
Retailing	Walmart	500343.0	9862.0	2300000	General Me...
Technology	Apple	229234.0	48351.0	123000	Computers,...
Telecommuni...	AT&T	160546.0	29450.0	254000	Telecommun...
Transportation	UPS	65872.0	4910.0	346415	Mail, Pack...
Wholesalers	McKesson	198533.0	5070.0	64500	Wholesaler...

Приведем такой пример: методу `nth` передается аргумент `3`, чтобы получить компании, занимающие четвертое место в каждом секторе в наборе данных `fortune`. В результат вошла 21 компания, занявшая четвертое место по размеру выручки в своем секторе:

```
In [21] sectors.nth(3)
```

```
Out [21]
```

Sector	Company	Revenues	Profits	Employees	Industry
Aerospace &...	General Dy...	30973.0	2912.0	98600	Aerospace ...
Apparel	Ralph Lauren	6653.0	-99.3	18250	Apparel
Business Se...	Aramark	14604.0	373.9	215000	Diversifie...
Chemicals	Monsanto	14640.0	2260.0	21900	Chemicals
Energy	Valero Energy	88407.0	4 065.0	10015	Petroleum ...
...
Retailing	Home Depot	100904.0	8630.0	413000	Specialty ...
Technology	IBM	79139.0	5753.0	397800	Informatio...
Telecommuni...	Charter Co...	41581.0	9895.0	94800	Telecommun...
Transportation	Delta Air ...	41244.0	3577.0	86564	Airlines
Wholesalers	Sysco	55371.0	1142.5	66500	Wholesaler...

Обратите внимание, что значением для сектора `Apparel` указана компания `Ralph Lauren`. Мы можем подтвердить правильность вывода, отфильтровав записи `Apparel` в `fortune`. И действительно, `Ralph Lauren` стоит в списке именно четвертой:

```
In [22] fortune[fortune["Sector"] == "Apparel"].head()
```

```
Out [22]
```

	Company	Revenues	Profits	Employees	Sector	Industry
88	Nike	34350.0	4240.0	74400	Apparel	Apparel
241	VF	12400.0	614.9	69000	Apparel	Apparel
331	PVH	8915.0	537.8	28050	Apparel	Apparel
420	Ralph Lauren	6653.0	-99.3	18250	Apparel	Apparel
432	Hanesbrands	6478.0	61.9	67200	Apparel	Apparel

Метод `head` извлекает несколько записей из каждой группы. В следующем примере вызов `head(2)` извлекает первые две записи в каждом секторе. Результатом является `DataFrame` с 42 записями (21 уникальный сектор, по две записи для каждого сектора). Не путайте метод `head` объекта `GroupBy` с методом `head` объекта `DataFrame`:

```
In [23] sectors.head(2)
```

```
Out [23]
```

	Company	Revenues	Profits	Employees	Sector	Industry
0	Walmart	500343.0	9862.0	2300000	Retailing	General M...
1	Exxon Mobil	244363.0	19710.0	71200	Energy	Petroleum...
2	Berkshire...	242137.0	44940.0	377000	Financials	Insurance...
3	Apple	229234.0	48351.0	123000	Technology	Computers...
4	UnitedHea...	201159.0	10558.0	260000	Health Care	Health Ca...

160	Visa	18358.0	6699.0	15000	Business	... Financial...
162	Kimberly-...	18259.0	2278.0	42000	Household...	Household...
163	AECOM	18203.0	339.4	87000	Engineeri...	Engineeri...
189	Sherwin-W...	14984.0	1772.3	52695	Chemicals	Chemicals
241	VF	12400.0	614.9	69000	Apparel	Apparel

Парный ему метод `tail` извлекает последние записи из каждой группы. Например, `tail(3)` извлекает последние три записи в каждом секторе. В результате получается `DataFrame` с 63 записями (21 сектор × 3 записи):

```
In [24] sectors.tail(3)
```

```
Out [24]
```

	Company	Revenues	Profits	Employees	Sector	Industry
473	Windstrea...	5853.0	-2116.6	12979	Telecommu...	Telecommu...
520	Telephone...	5044.0	153.0	9900	Telecommu...	Telecommu...
667	Weis Markets	3467.0	98.4	23000	Food & D...	Food and ...
759	Hain Cele...	2853.0	67.4	7825	Food, Bev...	Food Cons...
774	Fossil Group	2788.0	-478.2	12300	Apparel	Apparel


```

995 SiteOne L...      1862.0    54.6      3664 Wholesalers Wholesale...
996 Charles R...     1858.0   123.4     11800 Health Care Health Ca...
997   CoreLogic      1851.0   152.2      5900 Business ... Financial...
998 Ensign Group     1849.0    40.5     21301 Health Care Health Ca...
999       HCP        1848.0   414.2       190 Financials Real estate

```

```
63 rows × 6 columns
```

Получить все записи из данной группы можно вызовом метода `get_group`. Он возвращает `DataFrame` с соответствующими записями. Вот пример, который показывает, как получить все компании из сектора `Energy`:

```
In [25] sectors.get_group("Energy").head()
```

```
Out [25]
```

	Company	Revenues	Profits	Employees	Sector	Industry
1	Exxon Mobil	244363.0	19710.0	71200	Energy	Petroleum R...
12	Chevron	134533.0	9195.0	51900	Energy	Petroleum R...
27	Phillips 66	91568.0	5106.0	14600	Energy	Petroleum R...
30	Valero Energy	88407.0	4065.0	10015	Energy	Petroleum R...
40	Marathon Pe...	67610.0	3432.0	43800	Energy	Petroleum R...

Теперь, разобравшись с механикой работы объекта `GroupBy`, обсудим возможность агрегирования значений в каждой группе.

9.4. АГРЕГАТНЫЕ ОПЕРАЦИИ

Объект `GroupBy` предлагает методы для применения агрегатных операций к каждой группе. Метод `sum`, например, складывает значения столбцов в каждой группе. По умолчанию, если не ограничивать ее работу, `pandas` нацелена рассчитывать суммы во всех числовых столбцах в `DataFrame`. В следующем примере метод `sum` применяется к объекту `GroupBy` и вычисляет суммы по секторам для трех числовых столбцов (`Revenues` (Выручка), `Profits` (Прибыль) и `Employees` (Количество сотрудников)) в наборе данных `fortune`:

```
In [26] sectors.sum().head(10)
```

```
Out [26]
```

Sector	Revenues	Profits	Employees
Aerospace & Defense	383835.0	26733.5	1010124
Apparel	101157.3	6350.7	355699
Business Services	316090.0	37179.2	1593999
Chemicals	251151.0	20475.0	474020
Energy	1543507.2	85369.6	981207
Engineering & Construction	172782.0	7121.0	420745
Financials	2442480.0	264253.5	3500119

Food & Drug Stores	405468.0	8440.3	1398074
Food, Beverages & Tobacco	510232.0	54902.5	1079316
Health Care	1507991.4	92791.1	2971189

Проверим результаты, полученные в примере. Pandas сообщает сумму выручки 383 835 долларов в секторе **Aerospace & Defense**. Используем метод `get_group`, чтобы получить **DataFrame** с компаниями в группе **Aerospace & Defense**, отберем его столбец **Revenues** и вызовем метод `sum` для вычисления суммы значений в нем:

```
In [27] sectors.get_group("Aerospace & Defense").head()
```

```
Out [27]
```

	Company	Revenues	Profits	Employees	Sector	Industry
26	Boeing	93392.0	8197.0	140800	Aerospace...	Aerospace...
50	United Te...	59837.0	4552.0	204700	Aerospace...	Aerospace...
58	Lockheed ...	51048.0	2002.0	100000	Aerospace...	Aerospace...
98	General D...	30973.0	2912.0	98600	Aerospace...	Aerospace...
117	Northrop ...	25803.0	2015.0	70000	Aerospace...	Aerospace...

```
In [28] sectors.get_group("Aerospace & Defense").loc[:, "Revenues"].head()
```

```
Out [28] 26    93392.0
          50    59837.0
          58    51048.0
          98    30973.0
          117   25803.0
          Name: Revenues, dtype: float64
```

```
In [29] sectors.get_group("Aerospace & Defense").loc[:, "Revenues"].sum()
```

```
Out [29] 383835.0
```

Значения, полученные первым и вторым способом, равны между собой. Pandas выполнила расчеты верно! В вызове единственного метода `sum` библиотека применяет логику вычислений к каждому набору данных **DataFrame** в объекте **sectors**. Итак, мы выполнили агрегатную операцию минимальным объемом кода.

Объект **GroupBy** поддерживает множество других агрегатных методов. Например, вызовем метод `mean`, вычисляющий средние значения по столбцам **Revenues**, **Profits** и **Employees** для каждого сектора. И снова pandas включает в свои вычисления только числовые столбцы:

```
In [30] sectors.mean().head()
```

```
Out [30]
```

Sector	Revenues	Profits	Employees
Aerospace & Defense	15353.400000	1069.340000	40404.960000
Apparel	7225.521429	453.621429	25407.071429

Business Services	5963.962264	701.494340	30075.452830
Chemicals	7610.636364	620.454545	14364.242424
Energy	14425.300935	805.373585	9170.158879

Можно выбрать один столбец в `fortune`, передав его имя в квадратных скобках после объекта `GroupBy`. В этом случае pandas вернет новый объект `SeriesGroupBy`:

```
In [31] sectors["Revenues"]
```

```
Out [31] <pandas.core.groupby.generic.SeriesGroupBy object at 0x114778210>
```

«За кулисами» выводимых результатов объект `DataFrameGroupBy` хранит коллекцию объектов `SeriesGroupBy`, которые могут выполнять агрегатные операции с отдельными столбцами. Pandas организует результаты по группам. Вот, например, как можно вычислить сумму выручки по секторам:

```
In [32] sectors["Revenues"].sum().head()
```

```
Out [32] Sector
Aerospace & Defense    383835.0
Apparel                101157.3
Business Services     316090.0
Chemicals              251151.0
Energy                 1543507.2
Name: Revenues, dtype: float64
```

А в следующем примере вычисляется среднее количество работников по секторам:

```
In [33] sectors["Employees"].mean().head()
```

```
Out [33] Sector
Aerospace & Defense    40404.960000
Apparel                25407.071429
Business Services     30075.452830
Chemicals              14364.242424
Energy                 9170.158879
Name: Employees, dtype: float64
```

Метод `max` возвращает максимальное значение в заданном столбце. Приведу пример, в котором определяются максимальные значения в столбце `Profits` по секторам. Согласно результатам, самая эффективная компания в секторе `Aerospace & Defense` получила прибыль 8197 долларов:

```
In [34] sectors["Profits"].max().head()
```

```
Out [34] Sector
Aerospace & Defense    8197.0
Apparel                4240.0
```

```

Business Services      6699.0
Chemicals              3000.4
Energy                19710.0
Name: Profits, dtype: float64

```

Парный ему метод `min` возвращает минимальное значение в заданном столбце. Например, ниже показано минимальное количество сотрудников в одной компании по секторам. Судя по результатам, самая маленькая компания в секторе **Aerospace & Defense** насчитывает 5157 сотрудников:

```
In [35] sectors["Employees"].min().head()
```

```

Out [35] Sector
Aerospace & Defense    5157
Apparel               3700
Business Services     2338
Chemicals             1931
Energy                593
Name: Employees, dtype: int64

```

Метод `agg` применяет несколько агрегатных операций к разным столбцам и принимает словарь в качестве аргумента. В каждой паре «ключ/значение» в этом словаре ключ задает столбец, а значение — операцию, применяемую к столбцу. В следующем примере показано, как узнать минимальную выручку, максимальную прибыль и среднее количество сотрудников для каждого сектора, фактически выполнив только одну строку кода:

```

In [36] aggregations = {
        "Revenues": "min",
        "Profits": "max",
        "Employees": "mean"
    }

sectors.agg(aggregations).head()

```

```
Out [36]
```

Sector	Revenues	Profits	Employees
Aerospace & Defense	1877.0	8197.0	40404.960000
Apparel	2350.0	4240.0	25407.071429
Business Services	1851.0	6699.0	30075.452830
Chemicals	1925.0	3000.4	14364.242424
Energy	1874.0	19710.0	9170.158879

Pandas возвращает **DataFrame** с ключами из словаря агрегирования в качестве заголовков столбцов. Названия секторов играют роль индексных меток.

9.5. ПРИМЕНЕНИЕ СОБСТВЕННЫХ ОПЕРАЦИЙ КО ВСЕМ ГРУППАМ НАБОРА

Предположим, что нам понадобилось применить свою собственную операцию к каждой группе в объекте `GroupBy`. В разделе 9.4 был использован метод `max` объекта `GroupBy`, чтобы найти максимальную сумму выручки в каждом секторе. А теперь представим, что нам понадобилось определить, какая компания в каждом секторе получила максимальную выручку. Мы уже решили эту задачу выше, но предположим, что набор данных `fortune` не упорядочен.

Метод `nlargest` объекта `DataFrame` извлекает записи с наибольшим значением в заданном столбце. Следующий пример возвращает пять записей из `fortune` с наибольшими значениями в столбце `Profits`:

```
In [37] fortune.nlargest(n = 5, columns = "Profits")
```

```
Out [37]
```

	Company	Revenues	Profits	Employees	Sector	Industry
3	Apple	229234.0	48351.0	123000	Technology	Computers...
2	Berkshire...	242137.0	44940.0	377000	Financials	Insurance...
15	Verizon	126034.0	30101.0	155400	Telecommu...	Telecommu...
8	AT&T	160546.0	29450.0	254000	Telecommu...	Telecommu...
19	JPMorgan ...	113899.0	24441.0	252539	Financials	Commercia...

Если бы мы могли вызвать метод `nlargest` для каждого `DataFrame` в `sectors`, то получили бы искомые результаты — компании с наибольшей выручкой в каждом секторе.

Для решения этой задачи можно использовать метод `apply` объекта `GroupBy`. Он принимает функцию, вызывает ее один раз для каждой группы в объекте `GroupBy`, собирает полученные результаты и возвращает их в новом `DataFrame`.

Для начала определим свою функцию `get_largest_row`, которая принимает один аргумент: `DataFrame`. Она должна вернуть запись из `DataFrame` с наибольшим значением в столбце `Revenues`. Функция динамическая и может применяться к любому `DataFrame`, имеющему столбец `Revenues`:

```
In [38] def get_largest_row(df):
         return df.nlargest(1, "Revenues")
```

Теперь можно вызвать метод `apply` и передать ему функцию `get_largest_row`. Pandas вызовет `get_largest_row` один раз для каждого сектора и вернет `DataFrame` с компаниями, получившими наибольшую выручку в своем секторе:

```
In [39] sectors.apply(get_largest_row).head()
```

```
Out [39]
```

Sector		Company	Revenues	Profits	Employees	Industry
Aerospace ...	26	Boeing	93392.0	8197.0	140800	Aerospace...
Apparel	88	Nike	34350.0	4240.0	74400	Apparel
Business S...	142	ManpowerG...	21034.0	545.4	29000	Temporary...
Chemicals	46	DowDuPont	62683.0	1460.0	98000	Chemicals
Energy	1	Exxon Mobil	244363.0	19710.0	71200	Petroleum...

Используйте метод `apply` в случаях, когда `pandas` не поддерживает нужную агрегатную операцию.

9.6. ГРУППИРОВКА ПО НЕСКОЛЬКИМ СТОЛБЦАМ

Можно создать объект `GroupBy`, сгруппировав данные по значениям из нескольких столбцов в `DataFrame`. Эта возможность особенно удобна, когда группы идентифицируются значениями комбинаций столбцов. В следующем примере методу `groupby` передается список из двух позиций. `Pandas` группирует записи сначала по значениям столбца `Sector`, а затем по значениям столбца `Industry`. Помните, что отрасль (`Industry`), которой принадлежит компания, является подкатегорией более крупной категории `Sector`:

```
In [40] sector_and_industry = fortune.groupby(by = ["Sector", "Industry"])
```

Метод `size` объекта `GroupBy` возвращает серии `MultiIndex` с количествами записей в каждой группе. Суммарно этот объект `GroupBy` включает 82 записи, то есть в исходном наборе имеется 82 уникальные комбинации сектора и отрасли:

```
In [41] sector_and_industry.size()
```

```
Out [41]
```

Sector	Industry	
Aerospace & Defense	Aerospace and Defense	25
Apparel	Apparel	14
Business Services	Advertising, marketing	2
	Diversified Outsourcing Services	14
	Education	2

Transportation	Trucking, Truck Leasing	11
Wholesalers	Wholesalers: Diversified	24
	Wholesalers: Electronics and Office Equipment	8
	Wholesalers: Food and Grocery	6
	Wholesalers: Health Care	6

```
Length: 82, dtype: int64
```

302 Часть II. Библиотека pandas на практике

Метод `get_group` принимает кортеж значений и возвращает выборку данных `DataFrame` из коллекции `GroupBy`. Пример ниже демонстрирует извлечение записей с сектором "Business Services" и отраслью "Education":

```
In [42] sector_and_industry.get_group(("Business Services", "Education"))
```

```
Out [42]
```

	Company	Revenues	Profits	Employees	Sector	Industry
567	Laureate ...	4378.0	91.5	54500	Business ...	Education
810	Graham Ho...	2592.0	302.0	16153	Business ...	Education

Все агрегатные операции в `pandas` возвращают наборы `MultiIndex DataFrame` с результатами. Следующий пример вычисляет сумму трех числовых столбцов в `fortune` (`Revenues`, `Profits` и `Employees`), предварительно сгруппировав записи по секторам, а затем по отраслям в каждом секторе:

```
In [43] sector_and_industry.sum().head()
```

```
Out [43]
```

		Revenues	Profits	Employees
Sector	Industry			
Aerospace & Defense	Aerospace and Def...	383835.0	26733.5	1010124
Apparel	Apparel	101157.3	6350.7	355699
Business Services	Advertising, mark...	23156.0	1667.4	127500
	Diversified Outso...	74175.0	5043.7	858600
	Education	6970.0	393.5	70653

При необходимости можно извлечь отдельные столбцы из `fortune` для агрегирования, используя тот же синтаксис, что и в разделе 9.5. Для этого нужно указать имя столбца в квадратных скобках после объекта `GroupBy`, а затем вызвать агрегатный метод. Следующий пример вычисляет среднюю выручку для компаний в каждой комбинации «сектор/отрасль»:

```
In [44] sector_and_industry["Revenues"].mean().head(5)
```

```
Out [44]
```

Sector	Industry	
Aerospace & Defense	Aerospace and Defense	15353.400000
Apparel	Apparel	7225.521429
Business Services	Advertising, marketing	11578.000000
	Diversified Outsourcing Services	5298.214286
	Education	3485.000000

Name: Revenues, dtype: float64

Таким образом, объект `GroupBy` является оптимальной структурой данных для организации и агрегирования значений в `DataFrame`. Если понадобится использовать несколько столбцов для идентификации групп, то просто передайте методу `groupby` список столбцов.

9.7. УПРАЖНЕНИЯ

Представленные здесь упражнения основаны на наборе данных `cereals.csv` — списке, включающем 80 популярных сухих завтраков. Каждая запись содержит название завтрака, название производителя, тип, количество калорий, вес клетчатки в граммах и вес сахара в граммах. Посмотрим, как выглядит этот список:

```
In [45] cereals = pd.read_csv("cereals.csv")
        cereals.head()
```

Out [45]

	Name	Manufacturer	Type	Calories	Fiber	Sugars
0	100% Bran	Nabisco	Cold	70	10.0	6
1	100% Natural Bran	Quaker Oats	Cold	120	2.0	8
2	All-Bran	Kellogg's	Cold	70	9.0	5
3	All-Bran with Ex...	Kellogg's	Cold	50	14.0	0
4	Almond Delight	Ralston Purina	Cold	110	1.0	8

9.7.1. Задачи

Решите следующие задачи.

1. Сгруппируйте завтраки по значениям в столбце `Manufacturer` (Производитель).
2. Определите общее количество групп и завтраков в каждой группе.
3. Извлеките список завтраков, выпускаемых производителем `Nabisco`.
4. Вычислите средние значения по столбцам `Calories`, `Fiber` и `Sugars` для каждого производителя.
5. Найдите максимальное значение в столбце `Sugars` для каждого производителя.
6. Найдите минимальное значение в столбце `Fiber` для каждого производителя.
7. Найдите завтраки с наименьшим количеством сахара (столбец `Sugars`) по производителям и верните их в виде нового `DataFrame`.

9.7.2. Решения

Ниже приведу решения поставленных задач.

1. Сгруппировать завтраки по производителю можно с помощью метода `groupby` объекта `DataFrame`, с которым мы работаем, передав ему имя столбца `Manufacturer`. Для организации данных в группы pandas будет использовать уникальные значения в этом столбце:

```
In [46] manufacturers = cereals.groupby("Manufacturer")
```

2. Чтобы подсчитать общее количество групп по производителям, можно передать объект `GroupBy` стандартной функции `len`:

```
In [47] len(manufacturers)
```

```
Out [47] 7
```

Для определения количества завтраков, выпускаемых каждым производителем, воспользуемся методом `size` объекта `GroupBy`, который возвращает серию с количеством записей в каждой группе:

```
In [48] manufacturers.size()
```

```
Out [48] Manufacturer
American Home Food Products    1
General Mills                  22
Kellogg's                      23
Nabisco                        6
Post                           9
Quaker Oats                    8
Ralston Purina                 8
dtype: int64
```

3. Получим список завтраков, выпускаемых производителем `Nabisco`, вызвав метод `get_group` объекта `GroupBy`. Pandas вернет вложенный набор данных `DataFrame` со значением `Nabisco` в столбце `Manufacturer`:

```
In [49] manufacturers.get_group("Nabisco")
```

```
Out [49]
```

	Name	Manufacturer	Type	Calories	Fiber	Sugars
0	100% Bran	Nabisco	Cold	70	10.0	6
20	Cream of Wheat (Quick)	Nabisco	Hot	100	1.0	0
63	Shredded Wheat	Nabisco	Cold	80	3.0	0
64	Shredded Wheat 'n'Bran	Nabisco	Cold	90	4.0	0
65	Shredded Wheat spoon ...	Nabisco	Cold	90	3.0	0
68	Strawberry Fruit Wheats	Nabisco	Cold	90	3.0	5

4. Чтобы вычислить средние значения по столбцам `Calories`, `Fiber` и `Sugars` для каждого производителя, можно вызвать метод `mean` для объекта `manufacturers`.

По умолчанию pandas применяет агрегатные операции ко всем числовым столбцам:

```
In [50] manufacturers.mean()
```

```
Out [50]
```

Manufacturer	Calories	Fiber	Sugars
American Home Food Products	100.000000	0.000000	3.000000
General Mills	111.363636	1.272727	7.954545
Kellogg's	108.695652	2.739130	7.565217
Nabisco	86.666667	4.000000	1.833333
Post	108.888889	2.777778	8.777778
Quaker Oats	95.000000	1.337500	5.250000
Ralston Purina	115.000000	1.875000	6.125000

- Далее нам нужно найти максимальное значение в столбце **Sugars** для каждого производителя. Можно использовать квадратные скобки после объекта **GroupBy** и указать в них, по каким столбцам выполнить агрегирование, а затем вызвать нужный метод, в данном случае метод **max**:

```
In [51] manufacturers["Sugars"].max()
```

```
Out [51] Manufacturer
American Home Food Products    3
General Mills                  14
Kellogg's                     15
Nabisco                        6
Post                          15
Quaker Oats                   12
Ralston Purina                 11
Name: Sugars, dtype: int64
```

- Чтобы найти минимальное значение в столбце **Fiber** для каждого производителя, можно поменять имя столбца-аргумента в квадратных скобках на **Fiber** и вызвать метод **min**:

```
In [52] manufacturers["Fiber"].min()
```

```
Out [52] Manufacturer
American Home Food Products    0.0
General Mills                  0.0
Kellogg's                     0.0
Nabisco                       1.0
Post                          0.0
Quaker Oats                   0.0
Ralston Purina                 0.0
Name: Fiber, dtype: float64
```

- Итак, в последнем упражнении выберем завтраки с наименьшим количеством сахара (столбец **Sugars**) по производителям. Эту задачу можно решить

с помощью метода `apply` и собственной функции. Эта функция — `smallest_sugar_row` — использует метод `nsmallest`, чтобы получить из `DataFrame` запись с наименьшим значением в столбце `Sugars`, а метод `apply` обеспечивает вызов этой функции для каждой группы в `GroupBy`:

```
In [53] def smallest_sugar_row(df):
        return df.nsmallest(1, "Sugars")
In [54] manufacturers.apply(smallest_sugar_row)

Out [54]
```

		Name	Manufacturer	Type	Calories	Fiber	Sugars
Manufacturer							
American H...	43	Maypo	American ...	Hot	100	0.0	3
General Mills	11	Cheerios	General M...	Cold	110	2.0	0
Nabisco	20	Cream of ...	Nabisco	Hot	100	1.0	0
Post	33	Grape-Nuts	Post	Cold	110	3.0	3
Quaker Oats	57	Quaker Oa...	Quaker Oats	Hot	100	2.7	-1
Ralston Pu...	61	Rice Chex	Ralston P...	Cold	110	0.0	2

Поздравляю с завершением выполнения упражнений!

РЕЗЮМЕ

- Объект `GroupBy` является контейнером для группировки записей из `DataFrame`.
- Библиотека `pandas` группирует записи в `GroupBy` по значениям одного или нескольких столбцов.
- Методы `first` и `last` возвращают первую и последнюю записи из каждой группы в `GroupBy`. Порядок записей в каждой группе определяется порядком записей в исходном наборе данных `DataFrame`.
- Методы `head` и `tail` извлекают несколько записей из каждой группы в объекте `GroupBy`, опираясь на их позиции в исходном наборе данных `DataFrame`.
- Метод `nth` извлекает из каждой группы в `GroupBy` запись с указанным порядковым номером.
- `Pandas` может выполнять агрегатные операции, такие как суммирование, вычисление среднего, определение максимального и минимального значения, для каждой группы в объекте `GroupBy`.
- Метод `agg` позволяет применить сразу несколько агрегатных операций к разным столбцам. Он принимает словарь с именами столбцов в качестве ключей и агрегатными операциями в качестве значений.
- Метод `apply` позволяет вызвать функцию для каждой группы в объекте `GroupBy`.

10

Слияние, соединение и конкатенация

В этой главе

- ✓ Конкатенация наборов данных DataFrame по вертикальной и горизонтальной осям.
- ✓ Слияние наборов данных DataFrame с использованием внутреннего, внешнего и левого соединения.
- ✓ Сравнение наборов данных DataFrame на содержание уникальных и общих значений.
- ✓ Соединение наборов данных DataFrame по индексным меткам.

По мере усложнения бизнес-сферы становится все труднее хранить все данные в одной коллекции. Рано или поздно в случае постоянного роста набора данных его объем превысит доступное значение, трудности станут критическими. Чтобы решить эту проблему, администраторы разбивают данные по нескольким таблицам, а затем связывают эти таблицы друг с другом с целью упростить определение отношений между ними.

Возможно, вам уже приходилось работать с базами данных, такими как PostgreSQL, MySQL или Oracle. Системы управления реляционными базами данных (СУРБД) следуют именно такой, описанной в предыдущем абзаце, парадигме. База данных состоит из таблиц. Таблица содержит записи, соответствующие одной модели предметной области. Таблица состоит из строк и столбцов. Каждая строка хранит одну запись, а каждый столбец — один атрибут

этой записи. Таблицы соединяются с помощью столбцов-ключей. Если раньше вы не работали с базами данных, то можете считать, что таблица фактически эквивалентна набору данных `DataFrame` в pandas.

Вот реальный пример. Представьте, что мы создаем сайт электронной коммерции и нам нужна таблица `users` для хранения информации о зарегистрированных пользователях сайта (табл. 10.1). Следуя соглашениям о реляционных базах данных, мы назначаем каждой записи уникальный числовой идентификатор. Значения идентификаторов будут храниться в столбце `id`. Столбец `id` называется *первичным ключом*, потому что его значения являются первичными идентификаторами конкретных строк.

Таблица 10.1

Users				
id	first_name	last_name	email	gender
1	Homer	Simpson	donutfan@simpson.com	Male
2	Bart	Simpson	troublemaker@simpson.com	Male

Представим, что наша следующая цель — слежение за заказами пользователей. Для этого мы создадим таблицу заказов `orders`, в которой будут храниться сведения о заказах, такие как название товара и цена. Но как связать каждый заказ с разместившим его пользователем? Взгляните на табл. 10.2.

Таблица 10.2

Orders				
id	item	price	quantity	user_id
1	Donut Box	4.99	4	1
2	Slingshot	19.99	1	2

Чтобы установить связь между двумя таблицами, администраторы базы данных создают столбцы внешнего ключа. *Внешний ключ* — это ссылка на запись в другой таблице. Ключ называется *внешним*, потому что его значение хранится за пределами текущей таблицы.

В каждой строке в таблице заказов `orders` в столбце `user_id` хранится идентификатор пользователя, разместившего заказ. То есть столбец `user_id` хранит внешние ключи — его значения являются ссылками на записи в другой таблице, в таблице пользователей `users`. Используя установленную связь между двумя таблицами, нетрудно определить, что заказ 1 был размещен пользователем `Homer Simpson` с идентификатором 1.

Использование внешних ключей позволяет свести к минимуму дублирование данных. Например, в таблицу `orders` не нужно копировать имена, фамилии

и адреса электронной почты пользователей. Достаточно сохранить только ссылку на правильную запись в `users`. Бизнес-сущности — информация о пользователях и заказах — хранятся раздельно, но при необходимости их можно связать друг с другом.

Когда придет время объединять таблицы, мы всегда можем обратиться к `pandas`. Библиотека отлично справляется с добавлением, конкатенацией, соединением, слиянием и комбинированием наборов данных `DataFrame` как в вертикальном, так и в горизонтальном направлениях. Она может идентифицировать уникальные и общие записи между наборами `DataFrame`, выполнять операции SQL, такие как внутренние, внешние, левые и правые соединения. В этой главе мы рассмотрим различия между этими соединениями и ситуации, в которых каждое из них может оказаться полезным и уместным.

10.1. ЗНАКОМСТВО С НАБОРАМИ ДАННЫХ

Импортируем библиотеку `pandas` и назовем ее псевдоним `pd`:

```
In [1] import pandas as pd
```

Наборы данных для этой главы взяты из социальной сети Meetup, где пользователи объединяются в группы по интересам, например туризм, литература и настольные игры. Организаторы групп планируют удаленные или очные мероприятия, которые посещают члены группы. Предметная область Meetup имеет несколько моделей данных, включая группы, категории и города.

Все наборы данных для этой главы находятся в каталоге `meetup`. Начнем наше исследование с импорта файлов `groups1.csv` и `groups2.csv`. Эти файлы содержат образцы зарегистрированных групп Meetup. Каждая группа включает числовой идентификатор, название, идентификатор категории и идентификатор города. Вот как выглядит содержимое `groups1`:

```
In [2] groups1 = pd.read_csv("meetup/groups1.csv")
      groups1.head()
```

```
Out [2]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001
3	8940	The New York City Anime Group	29	10001
4	10104	NYC Pit Bull Group	26	10001

Импортируем также и файл `groups2.csv`. Обратите внимание, что оба файла имеют одни и те же четыре столбца. Мы можем предположить, что информация

о группах была каким-то образом разделена и сохранена в двух файлах вместо одного:

```
In [3] groups2 = pd.read_csv("meetup/groups2.csv")
      groups2.head()
```

```
Out [3]
```

	group_id		name	category_id	city_id
0	18879327		BachataMania	5	10001
1	18880221	Photoshoot Chicago - Photography and ...		27	60601
2	18880426	Chicago Adult Push / Kick Scooter Gro...		31	60601
3	18880495	Chicago International Soccer Club		32	60601
4	18880695	Impact.tech San Francisco Meetup		2	94101

Каждая группа имеет внешний ключ `category_id`. Информация о категориях хранится в файле `category.csv`. А уже в этом файле каждая строка хранит числовой идентификатор и название категории:

```
In [4] categories = pd.read_csv("meetup/categories.csv")
      categories.head()
```

```
Out [4]
```

	category_id	category_name
0	1	Arts & Culture
1	3	Cars & Motorcycles
2	4	Community & Environment
3	5	Dancing
4	6	Education & Learning

Кроме того, каждая группа имеет внешний ключ `city_id`. Информация о городах хранится в наборе данных `cities.csv`. В нем каждый город имеет уникальный числовой идентификатор, название, штат и почтовый индекс. Давайте взглянем:

```
In [5] pd.read_csv("meetup/cities.csv").head()
```

```
Out [5]
```

	id	city	state	zip
0	7093	West New York	NJ	7093
1	10001	New York	NY	10001
2	13417	New York Mills	NY	13417
3	46312	East Chicago	IN	46312
4	56567	New York Mills	MN	56567

В наборе данных о городах имеет место небольшая проблема. Посмотрите на значение почтового индекса (`zip`) в первой строке. Число **7093** — это неверный

почтовый индекс; на самом деле в файле CSV хранится значение 07093. Почтовые индексы могут начинаться с нуля. К сожалению, pandas предполагает, что почтовые индексы являются целыми числами, и поэтому удаляет начальные, как ей кажется, незначимые нули. Чтобы решить эту проблему, можно в вызов функции `read_csv` добавить параметр `dtype` и передать в нем словарь, ключи которого соответствуют именам столбцов, а значения определяют соответствующие типы данных. Удостоверимся, что pandas в результате смогла импортировать значения столбца `zip` в виде строк:

```
In [6] cities = pd.read_csv(
        "meetup/cities.csv", dtype = {"zip": "string"})
        cities.head()
```

Out [6]

	id	city state	zip
0	7093	West New York NJ	07093
1	10001	New York NY	10001
2	13417	New York Mills NY	13417
3	46312	East Chicago IN	46312
4	56567	New York Mills MN	56567

Отлично! Можно продолжать. Итак, каждая группа в `groups1` и `groups2` относится к некоторой категории и городу. В столбцах `category_id` и `city_id` хранятся внешние ключи. Значения в столбце `category_id` соответствуют значениям в столбце `category_id` в `categories`. Значения в столбце `city_id` соответствуют значениям в столбце `id` в `cities`. Теперь, когда наши таблицы данных загружены в Jupyter, можно попробовать соединить их.

10.2. КОНКАТЕНАЦИЯ НАБОРОВ ДАННЫХ

Самый простой способ объединить два набора данных — *конкатенация*, добавление одного набора данных `DataFrame` в конец другого.

Группы данных `groups1` и `groups2` имеют одни и те же столбцы с одинаковыми именами. Предположим, что они представляют две половины одного целого. Попробуем объединить их в один `DataFrame`. Pandas имеет для этого удобную функцию `concat`. Ей можно передать параметр `objs` со списком наборов данных `DataFrame`, а pandas объединит объекты в том порядке, в котором они указаны в списке `objs`. В следующем примере строки в группах `groups2` добавляются в конец группы `groups1`:

```
In [7] pd.concat(objs = [groups1, groups2])
```

Out [7]

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001
3	8940	The New York City Anime Group	29	10001
4	10104	NYC Pit Bull Group	26	10001
...
8326	26377464	Shinect	34	94101
8327	26377698	The art of getting what you want [...	14	94101
8328	26378067	Streeterville Running Group	9	60601
8329	26378128	Just Dance NYC	23	10001
8330	26378470	FREE Arabic Chicago Evanston North...	31	60601

16330 rows × 4 columns

Объединенный `DataFrame` содержит 16 330 строк! Как нетрудно догадаться, его длина равна сумме длин наборов данных `groups1` и `groups2`:

```
In [8] len(groups1)
```

```
Out [8] 7999
```

```
In [9] len(groups2)
```

```
Out [9] 8331
```

```
In [10] len(groups1) + len(groups2)
```

```
Out [10] 16330
```

Pandas сохраняет индексные метки из исходных наборов данных `DataFrame`, поэтому в последней строке объединенного набора с количеством строк более 16 000 мы видим индекс **8330**. Индекс **8330** — это индекс последней строки в наборе данных `groups2`. Pandas не волнует, что один и тот же индекс присутствует и в `groups1`, и в `groups2`. В результате в объединенном наборе индексы могут повторяться.

Чтобы избежать такой ситуации, можно передать функции `concat` параметр `ignore_index` со значением `True`, тем самым сгенерировать стандартные числовые индексы. Объединенный `DataFrame` в этом случае будет содержать новые индексы:

```
In [11] pd.concat(objs = [groups1, groups2], ignore_index = True)
```

```
Out [11]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001
3	8940	The New York City Anime Group	29	10001


```

4          10104          NYC Pit Bull Group          26    10001
...          ...          ...          ...    ...
16325  26377464          Shinect          34    94101
16326  26377698  The art of getting what you want ...    14    94101
16327  26378067          Streeterville Running Group          9    60601
16328  26378128          Just Dance NYC          23    10001
16329  26378470  FREE Arabic Chicago Evanston Nort...    31    60601

16330 rows x 4 columns

```

А если мы пожелаем взять лучшее из обоих миров: создать неповторяющиеся индексы, но при этом сохранить информацию о том, из какого набора данных `DataFrame` получена каждая запись? Одно из решений — добавить параметр `keys` и передать ему список строк. `Pandas` свяжет каждую строку в списке `keys` с `DataFrame`, имеющим тот же порядковый номер, что и строка. Списки `keys` и `objs` должны быть одинаковой длины.

В следующем примере записям из набора данных `groups1` назначается ключ "G1", а записям из набора данных `groups2` — ключ "G2". Функция `concat` возвращает `MultiIndex DataFrame`. Первый уровень `MultiIndex` хранит ключи, а второй — индексные метки из соответствующего набора данных `DataFrame`:

```
In [12] pd.concat(objs = [groups1, groups2], keys = ["G1", "G2"])
```

```

Out [12]

```

	group_id	name	category_id	city_id
G1 0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001
3	8940	The New York City Anime Group	29	10001
4	10104	NYC Pit Bull Group	26	10001
...
G2 8326	26377464	Shinect	34	94101
8327	26377698	The art of getting what you wan...	14	94101
8328	26378067	Streeterville Running Group	9	60601
8329	26378128	Just Dance NYC	23	10001
8330	26378470	FREE Arabic Chicago Evanston No...	31	60601

```
16330 rows x 4 columns
```

При необходимости можно извлечь исходные наборы данных из объединенного, обратившись к ключам `G1` или `G2` на первом уровне `MultiIndex`. (Порядок использования метода доступа `loc` объекта `MultiIndex` подробно описывается в главе 7.) Прежде чем продолжить, сохраним объединенный набор данных `DataFrame` в переменной `groups`:

```
In [13] groups = pd.concat(objs = [groups1, groups2], ignore_index = True)
```

Нам предстоит использовать его в разделе 10.4.

10.3. ОТСУТСТВУЮЩИЕ ЗНАЧЕНИЯ В ОБЪЕДИНЕННЫХ DATAFRAME

При объединении двух `DataFrame` библиотека `pandas` подставит значение `NaN` на пересечении строк и столбцов, которые не являются общими для исходных наборов данных. Рассмотрим следующие два набора. Оба имеют столбец `Football`. Набор данных `sports_champions_A` имеет столбец `Baseball`, который отсутствует в наборе `sports_champions_B`, а тот, в свою очередь, имеет столбец `Hockey`, отсутствующий в `sports_champions_A`:

```
In [14] sports_champions_A = pd.DataFrame(
        data = [
            ["New England Patriots", "Houston Astros"],
            ["Philadelphia Eagles", "Boston Red Sox"]
        ],
        columns = ["Football", "Baseball"],
        index = [2017, 2018]
    )

    sports_champions_A
```

Out [14]

	Football	Baseball
2017	New England Patriots	Houston Astros
2018	Philadelphia Eagles	Boston Red Sox

```
In [15] sports_champions_B = pd.DataFrame(
        data = [
            ["New England Patriots", "St. Louis Blues"],
            ["Kansas City Chiefs", "Tampa Bay Lightning"]
        ],
        columns = ["Football", "Hockey"],
        index = [2019, 2020]
    )

    sports_champions_B
```

Out [15]

	Football	Hockey
2019	New England Patriots	St. Louis Blues
2020	Kansas City Chiefs	Tampa Bay Lightning

В процессе конкатенации этих наборов данных будут проставлены `NaN` на месте отсутствующих значений в столбцах `Baseball` и `Hockey`. В части из набора данных `sports_champions_A` нет значений для столбца `Hockey`, а в части из набора данных `sports_champions_B` нет значений для столбца `Baseball`:

```
In [16] pd.concat(objs = [sports_champions_A, sports_champions_B])
```

```
Out [16]
```

	Football	Baseball	Hockey
2017	New England Patriots	Houston Astros	NaN
2018	Philadelphia Eagles	Boston Red Sox	NaN
2019	New England Patriots	NaN	St. Louis Blues
2020	Kansas City Chiefs	NaN	Tampa Bay Lightning

По умолчанию pandas выполняет конкатенацию строк по горизонтали. Но иногда бывает желательно выполнить конкатенацию по вертикали. Определим еще один набор данных `sports_champions_C`, имеющий записи с теми же индексными метками, что и `sports_champions_A` (2017 и 2018), но с совершенно другими, отсутствующими в `sports_champions_A` столбцами Hockey и Basketball:

```
In [17] sports_champions_C = pd.DataFrame(
    data = [
        ["Pittsburgh Penguins", "Golden State Warriors"],
        ["Washington Capitals", "Golden State Warriors"]
    ],
    columns = ["Hockey", "Basketball"],
    index = [2017, 2018]
)

sports_champions_C
```

```
Out [17]
```

	Hockey	Basketball
2017	Pittsburgh Penguins	Golden State Warriors
2018	Washington Capitals	Golden State Warriors

Если выполнить конкатенацию наборов `sports_champions_A` и `sports_champions_C`, то pandas добавит строки из второго DataFrame после строк из первого DataFrame. В результате будут созданы строки с повторяющимися индексными метками 2017 и 2018:

```
In [18] pd.concat(objs = [sports_champions_A, sports_champions_C])
```

```
Out [18]
```

	Football	Baseball	Hockey	Basketball
2017	New England P...	Houston Astros	NaN	NaN
2018	Philadelphia ...	Boston Red Sox	NaN	NaN
2017	NaN	NaN	Pittsburgh Pe...	Golden State ...
2018	NaN	NaN	Washington Ca...	Golden State ...

Это не тот результат, что нам нужен. Нам нужно, чтобы записи с повторяющимися индексными метками (2017 и 2018) объединились и чтобы в столбцах не было отсутствующих значений.

Функция `concat` принимает необязательный параметр `axis`, в котором можно передать значение 1 или "columns", чтобы объединить наборы данных по оси столбцов:

```
In [19] # Следующие два вызова функции concat эквивалентны
        pd.concat(
            objs = [sports_champions_A, sports_champions_C],
            axis = 1
        )
        pd.concat(
            objs = [sports_champions_A, sports_champions_C],
            axis = "columns"
        )
```

Out [19]

	Football	Baseball	Hockey	Basketball
2017	New England P...	Houston Astros	Pittsburgh Pe...	Golden State ...
2018	Philadelphia ...	Boston Red Sox	Washington Ca...	Golden State ...

Вот так намного лучше!

Подытожим: функция `concat` может объединять два набора данных либо по горизонтали, либо по вертикали. Я предпочитаю называть этот процесс «склеиванием» двух наборов данных.

10.4. ЛЕВЫЕ СОЕДИНЕНИЯ

В отличие от конкатенации операция *соединения* (`join`) использует логический критерий, чтобы определить, какие строки или столбцы из двух наборов данных должны объединяться. Например, соединение можно выполнить только по строкам с общими значениями в обоих наборах данных. В следующих далее разделах описываются три вида соединений: левое, внутреннее и внешнее. Давайте исследуем их по очереди.

Левое соединение (`left join`) использует ключи из одного набора данных для извлечения значений из другого. Это эквивалентно операции `VLOOKUP` в Excel. Левое соединение подходит для случаев, когда один набор данных находится в центре внимания, а второй используется для предоставления дополнительной информации, связанной с первым набором. Рассмотрим схему на рис. 10.1. Представьте, что каждый круг на рисунке — это `DataFrame`. `DataFrame` слева находится в фокусе проводимого анализа.

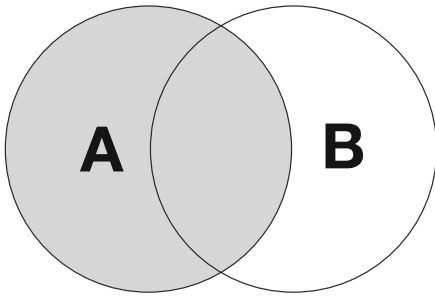


Рис. 10.1. Левое соединение

Напомним, как выглядит набор данных `groups`:

```
In [20] groups.head(3)
```

```
Out [20]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001

Внешние ключи в столбце `category_id` ссылаются на идентификаторы в наборе данных `categories`:

```
In [21] categories.head(3)
```

```
Out [21]
```

	category_id	category_name
0	1	Arts & Culture
1	3	Cars & Motorcycles
2	4	Community & Environment

Выполним левое соединение для набора `groups`, чтобы добавить информацию из `category` для каждой группы. Для слияния одного `DataFrame` с другим используем метод `merge`. В первом параметре, `right`, этот метод принимает `DataFrame`. Терминология взята из предыдущей схемы. Правый набор данных — это кружок справа, он же «второй» набор данных. В параметре `how` можно передать строку, обозначающую тип соединения; в данном случае передадим строку `"left"`. Мы также должны указать, какие столбцы использовать для сопоставления значений в двух наборах данных `DataFrame`. Для этого добавим параметр `on` со значением `"category_id"`. Параметр `on` можно использовать, только когда имя столбца совпадает в обоих `DataFrame`. В нашем случае оба набора данных, `groups` и `categories`, имеют столбец `category_id`:

```
In [22] groups.merge(categories, how = "left", on = "category_id").head()
```

```
Out [22]
```

	group_id	name	category_id	city_id	category_name
0	6388	Alternative Heal...	14	10001	Health & Wellbeing
1	6510	Alternative Ener...	4	10001	Community & Envi...
2	8458	NYC Animal Rights	26	10001	NaN
3	8940	The New York Cit...	29	10001	Sci-Fi & Fantasy
4	10104	NYC Pit Bull Group	26	10001	NaN

Вот и все! Pandas извлекает столбцы из таблицы `categories` всякий раз, когда находит совпадение со значением `category_id` в `groups`. Единственным исключением является столбец `category_id`, который включается в результирующий набор данных только один раз. Обратите внимание: когда библиотека не находит идентификатор категории в `categories`, она вставляет значение `NaN` в столбец `category_name`, полученный из `categories`. Примеры этого можно видеть в строках 2 и 4.

10.5. ВНУТРЕННИЕ СОЕДИНЕНИЯ

Внутреннее соединение (inner join) извлекает значения, существующие в двух наборах данных. Взгляните на рис. 10.2: внутреннее соединение представляет серая область в центре.

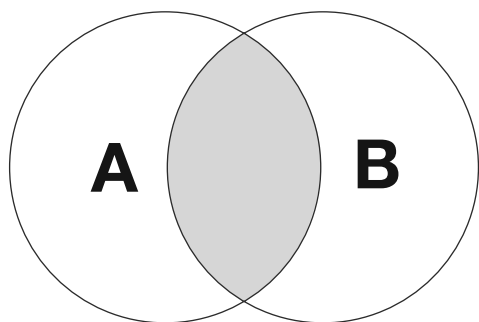


Рис. 10.2. Внутреннее соединение

Выполняя внутреннее соединение, pandas исключает значения, существующие только в каком-то одном наборе данных.

Вспомним, как выглядят наборы данных `groups` и `categories`:

```
In [23] groups.head(3)
```

```
Out [23]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001

In [24] categories.head(3)

Out [24]

	category_id	category_name
0	1	Arts & Culture
1	3	Cars & Motorcycles
2	4	Community & Environment

А теперь определим категории, присутствующие в обоих наборах данных. Выражаясь техническим языком, мы должны извлечь из двух `DataFrame` строки с одинаковыми значениями в столбце `category_id`. В этой ситуации не имеет значения, для какого объекта вызвать метод `merge` — `groups` или `categories`. Внутреннее соединение определяет общие элементы в обоих наборах данных, и результаты будут одинаковыми в любом случае. Вот пример вызова метода `merge` для `groups`:

In [25] groups.merge(categories, how = "inner", on = "category_id")

Out [25]

	group_id	name	category_id	city_id	category_name
0	6388	Alternative He...	14	10001	Health & Wellb...
1	54126	Energy Healers...	14	10001	Health & Wellb...
2	67776	Flourishing Li...	14	10001	Health & Wellb...
3	111855	Hypnosis & NLP...	14	10001	Health & Wellb...
4	129277	The Live Food ...	14	60601	Health & Wellb...
...
8032	25536270	New York Cucko...	17	10001	Lifestyle
8033	25795045	Pagans Paradis...	17	10001	Lifestyle
8034	25856573	Fuck Yeah Femm...	17	94101	Lifestyle
8035	26158102	Chicago Crossd...	17	60601	Lifestyle
8036	26219043	Corporate Goes...	17	10001	Lifestyle

8037 rows x 5 columns

Объединенный `DataFrame` включает все столбцы из `groups` и `categories`. Значения в столбце `category_id` соответствуют значениям как в `groups`, так и в `categories`. Столбец `category_id` включен в результирующий набор данных только один раз. Повторяющийся столбец не нужен, потому что во внутреннем соединении значения в `category_id` одинаковы для `groups` и `categories`.

Теперь немного поразмышляем над тем, что сделала `pandas`. Первые четыре строки в объединенном наборе данных имеют в столбце `category_id` значение 14.

Мы можем отфильтровать `groups` и `categories`, оставив только строки с этим значением:

```
In [26] groups[groups["category_id"] == 14]
```

```
Out [26]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
52	54126	Energy Healers NYC	14	10001
78	67776	Flourishing Life Meetup	14	10001
121	111855	Hypnosis & NLP NYC - Update Your ...	14	10001
136	129277	The Live Food Chicago Community	14	60601
...
16174	26291539	The Transformation Project: Colla...	14	94101
16201	26299876	Cognitive Empathy, How To Transla...	14	10001
16248	26322976	Contemplative Practices Group	14	94101
16314	26366221	The art of getting what you want:...	14	94101
16326	26377698	The art of getting what you want ...	14	94101

```
870 rows x 4 columns
```

```
In [27] categories[categories["category_id"] == 14]
```

```
Out [27]
```

	category_id	category_name
8	14	Health & Wellbeing

Объединенный `DataFrame` содержит по одной строке для каждого совпадения значения `category_id` в двух `DataFrame`. В `groups` и в `categories` имеется 870 и 1 строка соответственно с идентификатором `category_id`, равным 14. Pandas объединила каждую из 870 строк в `groups` с одной строкой в `categories` и создала объединенный набор данных с 870 строками. Поскольку внутреннее соединение создает новую строку для каждого совпадения, объединенный набор данных может оказаться значительно больше исходного. Например, если бы в `categories` имелось три строки со значением 14 в столбце `category_id`, то pandas создала бы набор с 2610 строками (870×3).

10.6. ВНЕШНИЕ СОЕДИНЕНИЯ

Внешнее соединение (outer join) объединяет все записи из двух наборов данных. Исключительное вхождение значений в наборы не имеет никакого значения для внешнего соединения. На рис. 10.3 показан результат внешнего соединения; pandas включает все значения независимо от того, принадлежат ли они к одному набору данных или к обоим.

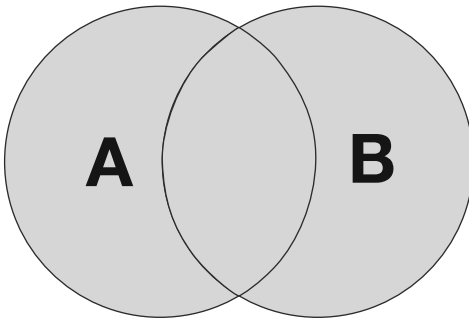


Рис. 10.3. Внешнее соединение

Вспомним, как выглядят наборы данных `groups` и `cities`:

```
In [28] groups.head(3)
```

```
Out [28]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001

```
In [29] cities.head(3)
```

```
Out [29]
```

	id	city	state	zip
0	7093	West New York	NJ	07093
1	10001	New York	NY	10001
2	13417	New York Mills	NY	13417

Выполним внешнее соединение `groups` и `cities`. Извлечем все города: существующие только в `groups`, только в `cities` и в обоих наборах данных.

Притом что сущности `id` одного набора и `city_id` другого совпадают, их имена различаются. До сих пор мы объединяли наборы данных, используя только общие, повторяющиеся в обоих наборах имена столбцов. Если имена столбцов различаются в наборах данных (как в рассматриваемом сейчас примере), то мы должны передать методу `merge` дополнительные параметры. Вместо одного параметра `on` можно передать два параметра: `left_on` и `right_on`. В параметре `left_on` передадим имя столбца в наборе данных слева и в параметре `right_on` — имя соответствующего столбца в наборе данных справа. Вот пример внешнего соединения, добавляющего информацию о городах из набора данных `cities` в набор данных `groups`:

322 Часть II. Библиотека pandas на практике

```
In [30] groups.merge(
        cities, how = "outer", left_on = "city_id", right_on = "id"
    )
```

Out [30]

	group_id	name	category_id	city_id	city	state	zip
0	6388.0	Altern...	14.0	10001.0	New York	NY	10001
1	6510.0	Altern...	4.0	10001.0	New York	NY	10001
2	8458.0	NYC An...	26.0	10001.0	New York	NY	10001
3	8940.0	The Ne...	29.0	10001.0	New York	NY	10001
4	10104.0	NYC Pi...	26.0	10001.0	New York	NY	10001
...
16329	243034...	Midwes...	34.0	60064.0	North ...	IL	60064
16330	NaN	NaN	NaN	NaN	New Yo...	NY	13417
16331	NaN	NaN	NaN	NaN	East C...	IN	46312
16332	NaN	NaN	NaN	NaN	New Yo...	MN	56567
16333	NaN	NaN	NaN	NaN	Chicag...	CA	95712

16334 rows × 8 columns

Окончательный `DataFrame` содержит все идентификаторы городов из обоих наборов данных. Обнаружив совпадение значений между столбцами `city_id` и `id`, pandas объединяет остальные значения столбцов из двух наборов данных в одну строку. Примеры этого можно видеть в первых пяти строках. В столбце `city_id` хранится общий идентификатор для двух наборов данных.

Если в одном `DataFrame` столбец с идентификатором города имеет значение, а в другом — нет, то pandas помещает значение `NaN` в столбец `city_id`. Примеры этого можно видеть в конце объединенного набора данных. Включение таких строк происходит независимо от отсутствия значения в `groups` или в `cities`.

В параметре `indicator` можно передать значение `True` методу `merge`, чтобы тот сообщил, какому `DataFrame` принадлежит каждое значение. Объединенный `DataFrame` будет включать столбец `_merge` со значениями `"both"`, `"left_only"` и `"right_only"`:

```
In [31] groups.merge(
        cities,
        how = "outer",
        left_on = "city_id",
        right_on = "id",
        indicator = True
    )
```

Out [31]

	group_id	name	category_id	city_id	city	state	zip	_merge
0	6388.0	Alt...	14.0	100...	New...	NY	10001	both
1	6510.0	Alt...	4.0	100...	New...	NY	10001	both

2	8458.0	NYC...	26.0	100...	New...	NY	10001	both
3	8940.0	The...	29.0	100...	New...	NY	10001	both
4	101...	NYC...	26.0	100...	New...	NY	10001	both
...
16329	243...	Mid...	34.0	600...	Nor...	IL	60064	both
16330	NaN	NaN	NaN	NaN	New...	NY	13417	rig...
16331	NaN	NaN	NaN	NaN	Eas...	IN	46312	rig...
16332	NaN	NaN	NaN	NaN	New...	MN	56567	rig...
16333	NaN	NaN	NaN	NaN	Chi...	CA	95712	rig...

16334 rows × 9 columns

Столбец `_merge` можно использовать для фильтрации строк, принадлежащих тому или иному `DataFrame`. В следующем примере показано, как извлечь строки со значением "right_only" в столбце `_merge` или, что то же самое, строки с идентификаторами городов, которые присутствуют только в `cities`, то есть в `DataFrame` справа:

```
In [32] outer_join = groups.merge(
        cities,
        how = "outer",
        left_on = "city_id",
        right_on = "id",
        indicator = True
    )

    in_right_only = outer_join["_merge"] == "right_only"

    outer_join[in_right_only].head()
```

Out [32]

	group_id	name	category_id	city_id		city	state	zip	_merge
16330	NaN	NaN	NaN	NaN	New Y...	NY	13417	right...	
16331	NaN	NaN	NaN	NaN	East ...	IN	46312	right...	
16332	NaN	NaN	NaN	NaN	New Y...	MN	56567	right...	
16333	NaN	NaN	NaN	NaN	Chica...	CA	95712	right...	

Теперь из этого набора можно узнать, какие значения присутствуют только в одном из наборов данных, — достаточно написать всего несколько строк кода.

10.7. СЛИЯНИЕ ПО ИНДЕКСНЫМ МЕТКАМ

Представьте, что `DataFrame`, участвующий в соединении, хранит первичные ключи в своем индексе. Давайте смоделируем эту ситуацию. Мы можем вызвать метод `set_index` для `cities`, чтобы назначить его столбец `id` на роль индекса `DataFrame`:

324 Часть II. Библиотека pandas на практике

```
In [33] cities.head(3)
```

```
Out [33]
```

	id	city state	zip
0	7093	West New York NJ	07093
1	10001	New York NY	10001
2	13417	New York Mills NY	13417

```
In [34] cities = cities.set_index("id")
```

```
In [35] cities.head(3)
```

```
Out [35]
```

	id	city state	zip
7093	West New York	NJ	07093
10001	New York	NY	10001
13417	New York Mills	NY	13417

Выполним левое соединение, чтобы снова присоединить `cities` к `groups`. Вспомним, как выглядит `groups`:

```
In [36] groups.head(3)
```

```
Out [36]
```

	group_id	name	category_id	city_id
0	6388	Alternative Health NYC	14	10001
1	6510	Alternative Energy Meetup	4	10001
2	8458	NYC Animal Rights	26	10001

На этот раз мы должны сравнить значения столбца `city_id` в `groups` с индексными метками в `cities`. Для этого, вызывая метод `merge`, следует передать в параметре `how` значение `"left"`, означающее левое соединение. В параметре `left_on` указать, что совпадения в левом наборе данных (`groups`) следует искать в столбце `city_id`. А чтобы поиск совпадений выполнялся в индексе правого `DataFrame`, нужно передать еще один параметр — `right_index` — со значением `True`. Этот аргумент сообщает pandas, что совпадения с `city_id` в правом наборе данных она должна искать в его индексе:

```
In [37] groups.merge(
        cities,
        how = "left",
        left_on = "city_id",
        right_index = True
    )
```

Out [37]

	group_id	name	category_id	city_id	city	state	zip
0	6388	Alterna...	14	10001	New York	NY	10001
1	6510	Alterna...	4	10001	New York	NY	10001
2	8458	NYC Ani...	26	10001	New York	NY	10001
3	8940	The New...	29	10001	New York	NY	10001
4	10104	NYC Pit...	26	10001	New York	NY	10001
...
16325	26377464	Shinect	34	94101	San Fra...	CA	94101
16326	26377698	The art...	14	94101	San Fra...	CA	94101
16327	26378067	Streete...	9	60601	Chicago	IL	60290
16328	26378128	Just Da...	23	10001	New York	NY	10001
16329	26378470	FREE Ar...	31	60601	Chicago	IL	60290

16330 rows × 7 columns

Метод `merge` имеет также дополнительный параметр `left_index`. В нем можно передать значение `True`, чтобы `pandas` искала совпадения в индексе левого `DataFrame`. Левый `DataFrame` — это набор данных, для которого вызывается метод `merge`.

10.8. УПРАЖНЕНИЯ

Мы подошли к концу нашего исследования; спасибо, что присоединились к нам только что (шутка)! Проверим на практике, как вы усвоили понятия, представленные в этой главе.

Наборы данных для упражнений содержат информацию о продажах в вымышленном ресторане. В файлах `week_1_sales.csv` и `week_2_sales.csv` вы найдете списки с данными за две недели. Каждый заказ, сделанный в ресторане, включает идентификатор клиента и идентификатор приобретенного им блюда. Вот как выглядят первые пять строк в `week_1_sales`:

In [38] `pd.read_csv("restaurant/week_1_sales.csv").head()`

Out [38]

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1
3	202	2
4	155	9

326 Часть II. Библиотека pandas на практике

Файл `week_2_sales.csv` имеет аналогичную структуру. Импортируем эти файлы и сохраним наборы данных в переменных `week1` и `week2`:

```
In [39] week1 = pd.read_csv("restaurant/week_1_sales.csv")
        week2 = pd.read_csv("restaurant/week_2_sales.csv")
```

Столбец `Customer ID` содержит внешние ключи, ссылающиеся на значения в столбце `ID` в файле `customers.csv`. Каждая запись в `customers.csv` включает имя, фамилию, пол, название компании и род занятий клиента. Импортируем этот набор данных с помощью функции `read_csv` и назовем столбец `ID` индексом в `DataFrame`, добавив параметр `index_col`:

```
In [40] pd.read_csv("restaurant/customers.csv", index_col = "ID").head()
```

Out [40]

	First Name	Last Name	Gender	Company	Occupation
ID					
1	Joseph	Perkins	Male	Dynazzy	Community Outreach Specialist
2	Jennifer	Alvarez	Female	DabZ	Senior Quality Engineer
3	Roger	Black	Male	Tagfeed	Account Executive
4	Steven	Evans	Male	Fatz	Registered Nurse
5	Judy	Morrison	Female	Demivee	Legal Assistant

```
In [41] customers = pd.read_csv(
        "restaurant/customers.csv", index_col = "ID"
    )
```

В наборах данных `week1` и `week2` есть еще один столбец внешних ключей. Внешний ключ `Food ID` ссылается на значения в столбце `ID` в файле `food.csv`, содержащем список блюд. Для каждого блюда определены идентификатор, название и цена. Импортируя этот набор данных, назовем его столбец `Food ID` индексом в `DataFrame`:

```
In [42] pd.read_csv("restaurant/foods.csv", index_col = "Food ID")
```

Out [42]

Food ID	Food Item	Price
1	Sushi	3.99
2	Burrito	9.99
3	Taco	2.99
4	Quesadilla	4.25
5	Pizza	2.49
6	Pasta	13.99
7	Steak	24.99
8	Salad	11.25

```
9          Donut    0.99
10         Drink    1.75
```

```
In [43] foods = pd.read_csv("restaurant/foods.csv", index_col = "Food ID")
```

После импорта наборов данных можно приступить к выполнению заданий.

10.8.1. Задачи

Предлагаю решить следующие задачи.

1. Объедините данные о продажах за две недели в один **DataFrame**. Назначьте данным из набора **week1** ключ **"Week 1"**, а данным из набора **week2** — ключ **"Week 2"**.
2. Найдите клиентов, которые посещали ресторан на каждой из рассматриваемых недель.
3. Найдите клиентов, которые посещали ресторан на каждой рассматриваемой неделе и каждую неделю заказывали одно и то же блюдо.

СОВЕТ

Выполнить соединение наборов данных по нескольким столбцам можно, передав в параметре **on** список столбцов.

4. Найдите клиентов, посещавших ресторан только на первой неделе и только на второй неделе.
5. Каждая строка в наборе данных **week1** идентифицирует клиента, заказавшего блюдо. Для каждой строки в **week1** извлеките информацию о клиенте из набора данных **customers**.

10.8.2. Решения

Теперь рассмотрим решения.

1. Наше первое задание — объединить сведения о продажах за две недели в один набор данных **DataFrame**. Функция **concat**, доступная на верхнем уровне в библиотеке **pandas**, предлагает идеальное решение. Ей можно передать два набора данных в параметре **objs**. Чтобы добавить уровень **MultiIndex** для каждого из исходных наборов данных, можно передать параметр **keys** со списком меток уровней:

```
In [44] pd.concat(objs = [week1, week2], keys = ["Week 1", "Week 2"])
```

```
Out [44]
```

		Customer ID	Food ID
Week 1	0	537	9
	1	97	4
	2	658	1
	3	202	2
	4	155	9
...
Week 2	245	783	10
	246	556	10
	247	547	9
	248	252	9
	249	249	6

500 rows × 2 columns

2. Во втором задании предлагалось определить клиентов, которые посещали ресторан обе недели. Выражаясь техническим языком, мы должны найти идентификаторы клиентов, присутствующие в обоих наборах данных, `week1` и `week2`. Внутреннее соединение — вот что нам нужно. Для этого вызовем метод `merge` для `week1` и передадим ему `week2` в качестве правого набора данных. Укажем тип соединения `"inner"` и предложим pandas отыскать общие значения в столбце `Customer ID`:

```
In [45] week1.merge(
        right = week2, how = "inner", on = "Customer ID"
    ).head()
```

Out [45]

	Customer ID	Food ID_x	Food ID_y
0	537	9	5
1	155	9	3
2	155	1	3
3	503	5	8
4	503	5	9

Как вы наверняка помните, внутреннее соединение выбирает все совпадения идентификаторов клиентов в наборах данных `week1` и `week2`. Как следствие, в результате присутствуют дубликаты (клиенты 155 и 503). Если понадобится удалить дубликаты, то можно вызвать метод `drop_duplicates`, представленный в главе 5:

```
In [46] week1.merge(
        right = week2, how = "inner", on = "Customer ID"
    ).drop_duplicates(subset = ["Customer ID"]).head()
```

Out [46]

	Customer ID	Food ID_x	Food ID_y
0	537	9	5
1	155	9	3
3	503	5	8
5	550	6	7
6	101	7	4

3. В третьей задаче надо было найти клиентов, посещавших ресторан обе недели и заказывавших одно и то же блюдо. И снова правильным выбором является внутреннее соединение, выбирающее значения, которые присутствуют и в левом, и в правом наборах данных. Но на этот раз мы должны передать в параметре `on` список с именами двух столбцов, потому что требуется выбрать из наборов данных `week1` и `week2` записи с совпадающими значениями в столбцах `Customer ID` и `Food ID`:

```
In [47] week1.merge(
        right = week2,
        how = "inner",
        on = ["Customer ID", "Food ID"]
    )
```

Out [47]

	Customer ID	Food ID
0	304	3
1	540	3
2	937	10
3	233	3
4	21	4
5	21	4
6	922	1
7	578	5
8	578	5s

4. Одно из решений задачи поиска клиентов, посещавших ресторан только на одной из недель, — использовать внешнее соединение. Можем сопоставить записи в двух наборах данных, используя значения в столбце `Customer ID`, и передать в параметре `indicator` значение `True`, чтобы добавить столбец `_merge`. Pandas укажет, существует ли идентификатор клиента только в левом ("`left_only`"), только в правом ("`right_only`") или в обоих наборах ("`both`"):

```
In [48] week1.merge(
        right = week2,
        how = "outer",
        on = "Customer ID",
        indicator = True
    ).head()
```

Out [48]

	Customer ID	Food ID_x	Food ID_y	_merge
0	537	9.0	5.0	both
1	97	4.0	NaN	left_only
2	658	1.0	NaN	left_only
3	202	2.0	NaN	left_only
4	155	9.0	3.0	both

5. Для решения последней задачи, чтобы добавить информацию о клиентах к строкам в наборе данных `week1`, обратимся к левому соединению, и это будет оптимальным решением. Для этого можно вызвать метод `merge` набора данных `week1`, передав ему таблицу `customers` в качестве правого набора данных, и параметр `how` со значением `"left"`.

Сложность в том, что в наборе данных `week1` идентификаторы клиентов хранятся в столбце `Customer ID`, а в наборе данных `customers` — в индексных метках. Чтобы решить эту проблему, достаточно передать методу `merge` параметр `left_on` с именем столбца из набора данных `week1` и параметр `right_index` со значением `True`:

```
In [49] week1.merge(
        right = customers,
        how = "left",
        left_on = "Customer ID",
        right_index = True
    ).head()
```

Out [49]

	Customer ID	Food ID	First Name	Last Name	Gender	Company	Occupation
0	537	9	Cheryl	Carroll	Female	Zoombeat	Regist...
1	97	4	Amanda	Watkins	Female	Ozu	Accoun...
2	658	1	Patrick	Webb	Male	Browsebug	Commun...
3	202	2	Louis	Campbell	Male	Rhynoodle	Accoun...
4	155	9	Carolyn	Diaz	Female	Gigazoom	Databa...

Поздравляю с успешным решением задач!

РЕЗЮМЕ

- *Первичный ключ* — это уникальный идентификатор записи в наборе данных.
- *Внешний ключ* — это ссылка на запись в другом наборе данных.
- Функция `concat` объединяет наборы данных `DataFrame` либо по горизонтали, либо по вертикали.

- Метод `merge` соединяет два набора данных `DataFrame` на основе некоторого логического критерия.
- *Внутреннее соединение* определяет общие значения в двух наборах данных `DataFrame`. Обнаружив любое совпадение, `pandas` извлекает все столбцы из правого и левого `DataFrame`.
- *Внешнее соединение* объединяет два набора данных `DataFrame`. `Pandas` включает значения независимо от того, присутствуют ли они только в каком-то одном или в обоих наборах данных.
- *Левое соединение* извлекает столбцы из правого набора данных `DataFrame`, если значения в них присутствуют в левом наборе данных. Эта операция эквивалентна операции `VLOOKUP` в Excel.
- Левое соединение идеально подходит для случаев, когда второй набор данных содержит дополнительную информацию, которую желательно присоединить к основному набору данных.

11

Дата и время

В этой главе

- ✓ Преобразование объекта `Series` с текстовыми значениями в дату и время.
- ✓ Извлечение даты и времени из объектов `datetime`.
- ✓ Округление дат до конца недели, месяца и квартала.
- ✓ Сложение и вычитание дат и времени.

`datetime` — это тип данных, предназначенный для хранения даты и времени. Он может моделировать конкретную дату (например, 4 октября 2021 года), определенное время (например, 11:50) или и то, и другое (например, 4 октября 2021 года, 11:50). Объекты `datetime`, представляющие дату и время, играют важную роль, потому что позволяют выявлять тенденции с течением времени. Анализируя дату и время, финансовый аналитик может узнать, в какие дни недели акции приносят наибольшую прибыль, владелец ресторана может определить часы наибольшего наплыва клиентов, а технолог — выяснить, какие этапы в технологических процессах являются узкими местами. Ответ на вопрос «когда» в наборе данных часто может помочь с ответом на вопрос «почему».

В этой главе мы рассмотрим встроенные в Python объекты `datetime` и выясним, как `pandas` усовершенствует их с помощью своих объектов `Timestamp` и `Timedelta`. Вы также узнаете, как использовать библиотеку для преобразования строк

в даты, прибавления и вычитания отрезков времени, вычисления длительности и многого другого. Но не будем терять время и приступим к изучению.

11.1. ЗНАКОМСТВО С ОБЪЕКТОМ `TIMESTAMP`

Модуль — это файл с кодом на языке Python. Стандартная библиотека Python включает более 250 модулей, предоставляющих проверенные решения типовых задач, таких как подключение к базе данных, математические вычисления и тестирование. Стандартная библиотека существует для того, чтобы разработчики могли писать программное обеспечение, используя имеющиеся функции основного языка, без установки дополнительных средств. О Python часто говорят, что он поставляется с «батареями в комплекте»; подобно игрушке, этот язык готов к использованию прямо из коробки.

11.1.1. Как Python работает с датой и временем

Чтобы уменьшить потребление памяти, Python по умолчанию не загружает модули из стандартной библиотеки. Поэтому необходимо явно импортировать любые модули, используемые в нашем проекте. Так же как в случае с внешним пакетом (например, `pandas`), мы можем импортировать модуль с помощью ключевого слова `import` и назначить ему псевдоним с помощью ключевого слова `as`. Нашей целью является модуль `datetime` из стандартной библиотеки; в нем хранятся классы для работы с датами и временем. `dt` — популярный псевдоним для модуля `datetime`. Запустим новый блокнот Jupyter и импортируем `datetime` вместе с библиотекой `pandas`:

```
In [1] import datetime as dt
      import pandas as pd
```

Рассмотрим четыре класса, объявленные в модуле: `date`, `time`, `datetime` и `time-delta`. (Дополнительные подробности о классах и объектах вы найдете в приложении Б.)

Класс `data` представляет одну конкретную дату. Объект этого класса не хранит значений времени. Конструктор класса `data` принимает параметры `year`, `month` и `day`. Во всех параметрах ожидаются целые числа. В следующем примере создается объект `data` с датой моего дня рождения — 12 апреля 1991 года:

```
In [2] # Следующие две строки эквивалентны
      birthday = dt.date(1991, 4, 12)
      birthday = dt.date(year = 1991, month = 4, day = 12)
      birthday
```

```
Out [2] datetime.date(1991, 4, 12)
```

334 Часть II. Библиотека pandas на практике

Объект `date` сохраняет аргументы конструктора в своих атрибутах. Мы можем получить доступ к их значениям, обратившись к атрибутам `year`, `month` и `day`:

```
In [3] birthday.year
```

```
Out [3] 1991
```

```
In [4] birthday.month
```

```
Out [4] 4
```

```
In [5] birthday.day
```

```
Out [5] 12
```

Объект `date` — *неизменяемый*. Вы не сможете изменить его внутреннее состояние после создания. При попытке изменить какой-либо атрибут Python сгенерирует исключение `AttributeError`:

```
In [6] birthday.month = 10
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-15-2690a31d7b19> in <module>  
----> 1 birthday.month = 10  
AttributeError: attribute 'month' of 'datetime.date' objects is not writable
```

Класс `date` дополняется классом `time`, который моделирует конкретное время суток безотносительно к дате. В первых трех параметрах — `hour`, `minute` и `second` — конструктор `time` принимает целочисленные аргументы, обозначающие часы, минуты и секунды соответственно. Подобно объекту `date`, объект `time` — *неизменяемый*. В следующем примере создается экземпляр объекта `time`, моделирующий время 6:43:25 утра:

```
In [7] # Следующие две строки эквивалентны  
alarm_clock = dt.time(6, 43, 25)  
alarm_clock = dt.time(hour = 6, minute = 43, second = 25)  
alarm_clock
```

```
Out [7] datetime.time(6, 43, 25)
```

По умолчанию все три параметра принимают значение `0`. Если вызвать конструктор `time` без аргументов, то он создаст объект, представляющий полночь (`00:00:00`). Полночь — это 0 часов 0 минут и 0 секунд:

```
In [8] dt.time()
```

```
Out [8] datetime.time(0, 0)
```

В следующем примере конструктору `time` передается значение `9` в параметре `hour` и `42` — в параметре `second`, а параметр `minute` опущен. В этом случае атрибут

`minute` в объекте `time` получит значение по умолчанию 0. В итоге получится время 9:00:42:

```
In [9] dt.time(hour = 9, second = 42)
```

```
Out [9] datetime.time(9, 0, 42)
```

Конструктор `time` использует 24-часовой формат представления времени, значит, чтобы представить время после полудня, в параметре `hour` ему нужно передать значение, большее или равное 12. Следующий пример моделирует время 19:43:22, или, что то же самое, 7:43:22 пополудни:

```
In [10] dt.time(hour = 19, minute = 43, second = 22)
```

```
Out [10] datetime.time(19, 43, 22)
```

Объект `time` сохраняет аргументы конструктора в атрибутах объекта. Получить их значения можно, обратившись к атрибутам `hour`, `minute` и `second`:

```
In [11] alarm_clock.hour
```

```
Out [11] 6
```

```
In [12] alarm_clock.minute
```

```
Out [12] 43
```

```
In [13] alarm_clock.second
```

```
Out [13] 25
```

Следующий на очереди объект — `datetime`, содержащий и дату, и время. В первых шести параметрах (`year`, `month`, `day`, `hour`, `minute` и `second`) он принимает год, месяц, день, часы, минуты и секунды:

```
In [14] # The two lines below are equivalent
moon_landing = dt.datetime(1969, 7, 20, 22, 56, 20)
moon_landing = dt.datetime(
    year = 1969,
    month = 7,
    day = 20,
    hour = 22,
    minute = 56,
    second = 20
)
moon_landing
```

```
Out [14] datetime.datetime(1969, 7, 20, 22, 56, 20)
```

Параметры `year`, `month` и `day` являются обязательными. А параметры, определяющие время, можно опустить, в таком случае они получают значение по

умолчанию 0. Следующий пример моделирует полночь 1 января 2020 года (00:00:00). Здесь конструктору явно передаются параметры `year`, `month` и `day`; а параметры `hour`, `minute` и `second` опущены и получают значение по умолчанию 0:

```
In [15] dt.datetime(2020, 1, 1)
```

```
Out [15] datetime.datetime(2020, 1, 1, 0, 0)
```

И наконец, последний объект в модуле `datetime` — это `timedelta`. Он моделирует длительность — продолжительность во времени. Его конструктор принимает такие параметры, как `weeks`, `days` и `hours`, обозначающие количество недель, дней и часов соответственно. Все параметры являются необязательными и по умолчанию принимают значение 0. Конструктор складывает значения параметров, чтобы получить общую продолжительность. В следующем примере мы складываем 8 недель и 6 дней, что в сумме дает 62 дня (8 недель × 7 дней + 6 дней). В этом примере также складываются 3 часа, 58 минут и 12 секунд, что в сумме дает 14 292 секунды (238 минут × 60 секунд + 12 секунд):

```
In [16] dt.timedelta(
    weeks = 8,
    days = 6,
    hours = 3,
    minutes = 58,
    seconds = 12
)
```

```
Out [16] datetime.timedelta(days=62, seconds=14292)
```

Теперь, получив представление, как Python моделирует даты, время и продолжительность, посмотрим, как эти понятия используются в `pandas`.

11.1.2. Как pandas работает с датой и временем

Многие пользователи критикуют модуль `datetime` в Python. Вот некоторые их аргументы:

- слишком много модулей, о которых нужно помнить; дело в том, что в этой главе представлены только `datetime`, но доступны также дополнительные модули с поддержкой календарей, преобразованиями времени, служебными функциями и многим другим;
- слишком много классов, о которых нужно помнить;
- сложный программный интерфейс объектов, обслуживающих логику часовых поясов.

`Pandas` представляет объект `Timestamp`, который может служить заменой объекта `datetime`. Объекты `Timestamp` и `datetime` можно считать братьями; в экосистеме `pandas` их часто можно использовать взаимозаменяемо, например при передаче

методам в виде аргументов. Подобно тому как `Series` расширяет списки Python, `Timestamp` расширяет функциональные возможности более примитивного объекта `datetime`. Далее в этой главе мы детально рассмотрим некоторые из этих расширений.

Конструктор `Timestamp` доступен на верхнем уровне `pandas`; он принимает те же параметры, что и конструктор `datetime`. Первые три параметра, задающие дату (`year`, `month` и `day`), являются обязательными. Параметры, задающие время, — необязательные и по умолчанию принимают значение `0`. Следующий пример снова моделирует замечательную дату 12 апреля 1991 года:

```
In [17] # Следующие две строки эквивалентны
        pd.Timestamp(1991, 4, 12)
        pd.Timestamp(year = 1991, month = 4, day = 12)
```

```
Out [17] Timestamp('1991-04-12 00:00:00')
```

В `pandas` объект `Timestamp` считается равным объекту `date/datetime`, если они оба хранят одну и ту же информацию. Чтобы определить, равны ли эти объекты фактически, можно использовать оператор `==`:

```
In [18] (pd.Timestamp(year = 1991, month = 4, day = 12)
        == dt.date(year = 1991, month = 4, day = 12))
```

```
Out [18] True
```

```
In [19] (pd.Timestamp(year = 1991, month = 4, day = 12, minute = 2)
        == dt.datetime(year = 1991, month = 4, day = 12, minute = 2))
```

```
Out [19] True
```

Два объекта будут считаться неравными, если их значения различаются в дате или времени. В следующем примере создается экземпляр `Timestamp` со значением 2 в параметре `minute` и объект `datetime` со значением 1 в том же параметре. Проверка их равенства дает `False`:

```
In [20] (pd.Timestamp(year = 1991, month = 4, day = 12, minute = 2)
        == dt.datetime(year = 1991, month = 4, day = 12, minute = 1))
```

```
Out [20] False
```

Конструктор `Timestamp` отличается удивительной гибкостью и может принимать самые разные входные значения. Приведу пример, когда конструктору вместо последовательности целых чисел передается строка с датой в обычном формате ГГГГ-ММ-ДД (четыре цифры года, две цифры месяца и две цифры числа в месяце). И `pandas`, к всеобщему восхищению, правильно расшифровывает эту дату:

```
In [21] pd.Timestamp("2015-03-31")
```

```
Out [21] Timestamp('2015-03-31 00:00:00')
```

338 Часть II. Библиотека pandas на практике

На самом деле pandas распознает множество стандартных форматов представления даты и времени. В следующем примере дефисы в представлении даты заменены слешами:

```
In [22] pd.Timestamp("2015/03/31")
```

```
Out [22] Timestamp('2015-03-31 00:00:00')
```

А ниже конструктору передается строка в формате ММ/ДД/ГГГГ, и снова pandas правильно расшифровывает ее:

```
In [23] pd.Timestamp("03/31/2015")
```

```
Out [23] Timestamp('2015-03-31 00:00:00')
```

В строку можно включить также и значение времени в разных форматах:

```
In [24] pd.Timestamp("2021-03-08 08:35:15")
```

```
Out [24] Timestamp('2021-03-08 08:35:15')
```

```
In [25] pd.Timestamp("2021-03-08 6:13:29 PM")
```

```
Out [25] Timestamp('2021-03-08 18:13:29')
```

Наконец, конструктор `Timestamp` может принимать в качестве аргументов объекты `date`, `time` и `datetime`. Вот пример передачи ему объекта `datetime`:

```
In [26] pd.Timestamp(dt.datetime(2000, 2, 3, 21, 35, 22))
```

```
Out [26] Timestamp('2000-02-03 21:35:22')
```

Объект `Timestamp` реализует все атрибуты `datetime`, такие как `hour`, `minute` и `second`, и дает к ним доступ. Следующий пример сохраняет в переменной объект `Timestamp`, созданный выше, а затем выводит значения некоторых отдельных атрибутов:

```
In [27] my_time = pd.Timestamp(dt.datetime(2000, 2, 3, 21, 35, 22))
        print(my_time.year)
        print(my_time.month)
        print(my_time.day)
        print(my_time.hour)
        print(my_time.minute)
        print(my_time.second)
```

```
Out [27] 2000
         2
         3
        21
        35
        22
```

Разработчики pandas приложили все силы, чтобы их объекты, представляющие дату и время, работали так же, как стандартные объекты `datetime`. Таким образом, можно считать, что эти объекты полностью взаимозаменяемы в операциях pandas.

11.2. ХРАНЕНИЕ НЕСКОЛЬКИХ ОТМЕТОК ВРЕМЕНИ В DATETIMEINDEX

Индекс — это набор идентифицирующих меток, прикрепленных к структуре данных pandas. Наиболее распространенным видом индексов из числа рассмотренных нами до сих пор является `RangeIndex` — последовательность возрастающих или убывающих числовых значений. Получить доступ к индексу `Series` или `DataFrame` можно через атрибут `index`:

```
In [28] pd.Series([1, 2, 3]).index
```

```
Out [28] RangeIndex(start=0, stop=3, step=1)
```

Для хранения коллекции строковых меток pandas использует объект `Index`. Как показывает следующий пример, pandas выбирает тип объекта индекса для присоединения к `Series`, исходя из его содержимого:

```
In [29] pd.Series([1, 2, 3], index = ["A", "B", "C"]).index
```

```
Out [29] Index(['A', 'B', 'C'], dtype='object')
```

`DatetimeIndex` — это индекс, хранящий объекты `Timestamp`. Если в параметре `index` передать конструктору `Series` список отметок времени, то pandas присоединит к `Series` индекс `DatetimeIndex`:

```
In [30] timestamps = [
        pd.Timestamp("2020-01-01"),
        pd.Timestamp("2020-02-01"),
        pd.Timestamp("2020-03-01"),
    ]
    pd.Series([1, 2, 3], index = timestamps).index
```

```
Out [30] DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01'],
                        dtype='datetime64[ns]', freq=None)
```

Точно так же pandas будет использовать `DatetimeIndex`, если передать конструктору список объектов `datetime`:

```
In [31] datetimes = [
        dt.datetime(2020, 1, 1),
        dt.datetime(2020, 2, 1),
    ]
```

```

        dt.datetime(2020, 3, 1),
    ]
    pd.Series([1, 2, 3], index = datetimes).index

```

```

Out [31] DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01'],
                        dtype='datetime64[ns]', freq=None)

```

Мы можем создать `DatetimeIndex` с нуля. Его конструктор доступен на верхнем уровне `pandas`. В параметре `data` конструктору можно передать любую итерируемую коллекцию с датами. Даты могут быть представлены строками, объектами `datetime`, объектами `Timestamp` и даже их комбинациями. `Pandas` преобразует все значения в эквивалентные объекты `Timestamp` и сохранит их в индексе:

```

In [32] string_dates = ["2018/01/02", "2016/04/12", "2009/09/07"]
        pd.DatetimeIndex(data = string_dates)

```

```

Out [32] DatetimeIndex(['2018-01-02', '2016-04-12', '2009-09-07'],
                        dtype='datetime64[ns]', freq=None)

```

```

In [33] mixed_dates = [
        dt.date(2018, 1, 2),
        "2016/04/12",
        pd.Timestamp(2009, 9, 7)
    ]
    dt_index = pd.DatetimeIndex(mixed_dates)
    dt_index

```

```

Out [33] DatetimeIndex(['2018-01-02', '2016-04-12', '2009-09-07'],
                        dtype='datetime64[ns]', freq=None)

```

Теперь, когда у нас есть объект `DatetimeIndex` в переменной `dt_index`, присоединим его к структуре данных `pandas`. Рассмотрим на примере, как индекс присоединяется к выборке `Series`:

```

In [34] s = pd.Series(data = [100, 200, 300], index = dt_index)
        s

```

```

Out [34] 2018-01-02    100
         2016-04-12    200
         2009-09-07    300
         dtype: int64

```

`Pandas` поддерживает операции с датой и временем в объектах `Timestamp`, но не поддерживает этот функционал для значений в текстовых форматах. `Pandas` не может определить день недели, например, по текстовой строке "2018-01-02", потому что рассматривает ее как набор цифр и дефисов, но не как фактическую дату. Вот почему, импортируя набор данных, необходимо преобразовать все соответствующие строковые столбцы в дату и время.

Для сортировки содержимого структуры по индексу `DatetimeIndex` в порядке возрастания или убывания можно использовать метод `sort_index`. Следующий

пример демонстрирует сортировку в порядке возрастания (от самых ранних дат к более поздним):

```
In [35] s.sort_index()
Out [35] 2009-09-07    300
         2016-04-12    200
         2018-01-02    100
         dtype: int64
```

При сортировке или сравнении pandas учитывает и дату, и время. Даже если два объекта `Timestamp` представляют одну и ту же дату, pandas сравнит в том числе их атрибуты `hour`, `minute`, `second` и т. д.

Для `Timestamp` поддерживаются различные операции сортировки и сравнения. Например, оператор «меньше» (`<`) проверяет, представляет ли один объект `Timestamp` более ранний момент во времени, чем другой:

```
In [36] morning = pd.Timestamp("2020-01-01 11:23:22 AM")
         evening = pd.Timestamp("2020-01-01 11:23:22 PM")
         morning < evening
```

```
Out [36] True
```

В разделе 11.7 вы увидите, как применять эти типы сравнений ко всем значениям в `Series`.

11.3. ПРЕОБРАЗОВАНИЕ ЗНАЧЕНИЙ СТОЛБЦОВ ИЛИ ИНДЕКСОВ В ДАТУ И ВРЕМЯ

Наш первый набор данных для этой главы, `disney.csv`, содержит информацию почти за 60 лет о ценах на акции компании Walt Disney Company — одной из самых известных в мире в индустрии развлечений. Каждая строка включает дату, максимальную и минимальную стоимость акции за этот день, а также ее цену на момент открытия и закрытия торгов:

```
In [37] disney = pd.read_csv("disney.csv")
         disney.head()
```

```
Out [37]
```

	Date	High	Low	Open	Close
0	1962-01-02	0.096026	0.092908	0.092908	0.092908
1	1962-01-03	0.094467	0.092908	0.092908	0.094155
2	1962-01-04	0.094467	0.093532	0.094155	0.094155
3	1962-01-05	0.094779	0.093844	0.094155	0.094467
4	1962-01-08	0.095714	0.092285	0.094467	0.094155

По умолчанию функция `read_csv` импортирует все значения нечисловых столбцов в виде строк. Узнать типы столбцов можно с помощью атрибута `dtypes` объекта `DataFrame`. Обратите внимание, что столбец `Date` имеет тип данных `object`, так в pandas обозначаются строки:

```
In [38] disney.dtypes
```

```
Out [38] Date      object
         High      float64
         Low       float64
         Open      float64
         Close     float64
         dtype: object
```

Мы должны явно сообщить pandas, значения каких столбцов нужно преобразовать в `datetime`. Один из возможных способов сделать это, который мы видели выше, в главе 3, — добавить параметр `parse_dates` функции `read_csv`. В этом параметре можно передать список столбцов, значения которых должны быть преобразованы в `datetime`:

```
In [39] disney = pd.read_csv("disney.csv", parse_dates = ["Date"])
```

Альтернативным решением является функция преобразования `to_datetime`, доступная на верхнем уровне pandas. Она принимает итерируемый объект (например, список, кортеж, `Series` или индекс), преобразует его значения в объекты `datetime` и возвращает новые значения в виде `DatetimeIndex`. Вот небольшой пример:

```
In [40] string_dates = ["2015-01-01", "2016-02-02", "2017-03-03"]
         dt_index = pd.to_datetime(string_dates)
         dt_index
```

```
Out [40] DatetimeIndex(['2015-01-01', '2016-02-02', '2017-03-03'],
                        dtype='datetime64[ns]', freq=None)
```

Передадим столбец `Date` из набора данных `disney` в функцию `to_datetime`:

```
In [41] pd.to_datetime(disney["Date"]).head()
```

```
Out [41] 0    1962-01-02
         1    1962-01-03
         2    1962-01-04
         3    1962-01-05
         4    1962-01-08
         Name: Date, dtype: datetime64[ns]
```

Теперь, получив `Series` с объектами `datetime`, запишем его в исходный набор данных. Следующий пример замещает исходный столбец `Date` новой

последовательностью объектов `datetime`. Напомню, что Python сначала вычисляет правую часть выражения присваивания:

```
In [42] disney["Date"] = pd.to_datetime(disney["Date"])
```

Проверим еще раз столбец `Date`, обратившись к атрибуту `dtypes`:

```
In [43] disney.dtypes
Out [43] Date      datetime64[ns]
         High      float64
         Low       float64
         Open      float64
         Close     float64
         dtype: object
```

Отлично, у нас есть столбец с объектами `datetime`! Теперь, имея столбец `Date` со значениями в правильном формате, мы можем приступить к изучению мощных возможностей обработки даты и времени, имеющихся в `pandas`.

11.4. ИСПОЛЬЗОВАНИЕ ОБЪЕКТА DATETIMEPROPERTIES

Последовательность `Series` со значениями `datetime` имеет особый атрибут `dt` с объектом `DatetimeProperties`:

```
In [44] disney["Date"].dt

Out [44] <pandas.core.indexes.accessors.DatetimeProperties object at
         0x116247950>
```

Мы можем обращаться к атрибутам и методам объекта `DatetimeProperties` и извлекать с их помощью информацию из значений `datetime` в столбце. Атрибут `dt` играет ту же роль для значений `datetime`, что и атрибут `str` для строк. (Краткий обзор `str` вы найдете в главе 6.) Оба атрибута специализируются на манипуляциях с определенным типом данных.

Начнем изучение объекта `DatetimeProperties` с атрибута `day`, который извлекает день месяца из каждой даты. `Pandas` возвращает значения в виде новой последовательности `Series`:

```
In [45] disney["Date"].head(3)

Out [45] 0    1962-01-02
         1    1962-01-03
```

344 Часть II. Библиотека pandas на практике

```
2    1962-01-04
Name: Date, dtype: datetime64[ns]
```

```
In [46] disney["Date"].dt.day.head(3)
```

```
Out [46] 0     2
         1     3
         2     4
         Name: Date, dtype: int64
```

Атрибут `month` возвращает последовательность `Series` с номерами месяцев. Январю соответствует значение 1, февралю — значение 2 и т. д. Важно отметить, что такой порядок нумерации отличается от принятого в Python/pandas, где первому элементу присваивается порядковый номер 0:

```
In [47] disney["Date"].dt.month.head(3)
```

```
Out [47] 0     1
         1     1
         2     1
         Name: Date, dtype: int64
```

Атрибут `year` возвращает последовательность `Series` с годами:

```
In [48] disney["Date"].dt.year.head(3)
```

```
Out [48] 0    1962
         1    1962
         2    1962
         Name: Date, dtype: int64
```

Предыдущие атрибуты довольно просты. Однако pandas имеет атрибуты, возвращающие и более интересную информацию. Одним из них является атрибут `dayofweek`, возвращающий последовательность `Series` чисел, представляющих дни недели. 0 обозначает понедельник, 1 — вторник и далее до 6 — воскресенье. В следующем примере значение 1 в позиции индекса 0 указывает, что 2 января 1962 года приходится на вторник:

```
In [49] disney["Date"].dt.dayofweek.head()
```

```
Out [49] 0     1
         1     2
         2     3
         3     4
         4     0
         Name: Date, dtype: int64
```

А если нам понадобится название дня недели вместо его номера? В этом нам поможет метод `day_name`. Но будьте внимательны: метод должен вызываться для объекта `dt`, а не для самой последовательности `Series`:


```
In [50] disney["Date"].dt.day_name().head()
```

```
Out [50] 0    Tuesday
         1    Wednesday
         2    Thursday
         3    Friday
         4    Monday
         Name: Date, dtype: object
```

Эти атрибуты и методы `dt` можно использовать в сочетании с другими функциями `pandas`. Например, посчитаем среднюю доходность акций Disney по дням недели. Для начала прикрепим последовательность `Series`, возвращаемую методом `dt.day_name`, к набору данных `disney`:

```
In [51] disney["Day of Week"] = disney["Date"].dt.day_name()
```

После этого можно сгруппировать строки по значениям в новом столбце `Day of Week` (этот прием был представлен в главе 7):

```
In [52] group = disney.groupby("Day of Week")
```

И далее вызвать метод `mean` объекта `GroupBy`, чтобы вычислить среднее значение для каждой группы:

```
In [53] group.mean()
```

```
Out [53]
```

Day of Week	High	Low	Open	Close
Friday	23.767304	23.318898	23.552872	23.554498
Monday	23.377271	22.930606	23.161392	23.162543
Thursday	23.770234	23.288687	23.534561	23.540359
Tuesday	23.791234	23.335267	23.571755	23.562907
Wednesday	23.842743	23.355419	23.605618	23.609873

Всего тремя строками кода мы рассчитали среднюю доходность акций по дням недели.

Но вернемся к методам объекта `dt`. Метод `month_name` возвращает `Series` с названиями месяцев:

```
In [54] disney["Date"].dt.month_name().head()
```

```
Out [54] 0    January
         1    January
         2    January
         3    January
         4    January
         Name: Date, dtype: object
```

Некоторые атрибуты объекта `dt` возвращают логические значения. Предположим, мы решили исследовать динамику изменения стоимости акций компании Disney в начале каждого квартала за всю имеющуюся историю. Кварталы начинаются 1 января, 1 апреля, 1 июля и 1 октября. Атрибут `is_quarter_start` возвращает `Series` с логическими значениями, где `True` означает, что дата в строке приходится на день начала квартала:

```
In [55] disney["Date"].dt.is_quarter_start.tail()
Out [55] 14722    False
          14723    False
          14724    False
          14725     True
          14726    False
          Name: Date, dtype: bool
```

Последовательность `Series` с логическими значениями можно использовать для извлечения строк, приходящихся на начало каждого квартала. Следующий пример использует уже знакомый синтаксис квадратных скобок для извлечения строк:

```
In [56] disney[disney["Date"].dt.is_quarter_start].head()
```

```
Out [56]
```

	Date	High	Low	Open	Close	Day of Week
189	1962-10-01	0.064849	0.062355	0.063913	0.062355	Monday
314	1963-04-01	0.087989	0.086704	0.087025	0.086704	Monday
377	1963-07-01	0.096338	0.095053	0.096338	0.095696	Monday
441	1963-10-01	0.110467	0.107898	0.107898	0.110467	Tuesday
565	1964-04-01	0.116248	0.112394	0.112394	0.116248	Wednesday

Для извлечения строк, приходящихся на конец квартала, можно использовать атрибут `is_quarter_end`:

```
In [57] disney[disney["Date"].dt.is_quarter_end].head()
```

```
Out [57]
```

	Date	High	Low	Open	Close	Day of Week
251	1962-12-31	0.074501	0.071290	0.074501	0.072253	Monday
440	1963-09-30	0.109825	0.105972	0.108541	0.107577	Monday
502	1963-12-31	0.101476	0.096980	0.097622	0.101476	Tuesday
564	1964-03-31	0.115605	0.112394	0.114963	0.112394	Tuesday
628	1964-06-30	0.101476	0.100191	0.101476	0.100834	Tuesday

Атрибуты `is_month_start` и `is_month_end` позволяют выбрать строки с датами, приходящимися на начало и конец месяца соответственно:

```
In [58] disney[disney["Date"].dt.is_month_start].head()
```

```
Out [58]
```

	Date	High	Low	Open	Close	Day of Week
22	1962-02-01	0.096338	0.093532	0.093532	0.094779	Thursday
41	1962-03-01	0.095714	0.093532	0.093532	0.095714	Thursday
83	1962-05-01	0.087296	0.085426	0.085738	0.086673	Tuesday
105	1962-06-01	0.079814	0.077943	0.079814	0.079814	Friday
147	1962-08-01	0.068590	0.068278	0.068590	0.068590	Wednesday

```
In [59] disney[disney["Date"].dt.is_month_end].head()
```

```
Out [59]
```

	Date	High	Low	Open	Close	Day of Week
21	1962-01-31	0.093844	0.092908	0.093532	0.093532	Wednesday
40	1962-02-28	0.094779	0.093220	0.094155	0.093220	Wednesday
82	1962-04-30	0.087608	0.085738	0.087608	0.085738	Monday
104	1962-05-31	0.082308	0.079814	0.079814	0.079814	Thursday
146	1962-07-31	0.069214	0.068278	0.068278	0.068590	Tuesday

Атрибут `is_year_start` возвращает значение `True`, если дата приходится на начало года. Следующий пример возвращает пустой `DataFrame`: фондовый рынок закрыт в первый день Нового года, поэтому в наборе данных нет ни одной записи, соответствующей заданным критериям:

```
In [60] disney[disney["Date"].dt.is_year_start].head()
```

```
Out [60]
```

	Date	High	Low	Open	Close	Day of Week
--	------	------	-----	------	-------	-------------

Атрибут `is_year_end` возвращает значение `True`, если дата приходится на конец года:

```
In [61] disney[disney["Date"].dt.is_year_end].head()
```

```
Out [61]
```

	Date	High	Low	Open	Close	Day of Week
251	1962-12-31	0.074501	0.071290	0.074501	0.072253	Monday
502	1963-12-31	0.101476	0.096980	0.097622	0.101476	Tuesday
755	1964-12-31	0.117853	0.116890	0.116890	0.116890	Thursday
1007	1965-12-31	0.154141	0.150929	0.153498	0.152214	Friday
1736	1968-12-31	0.439301	0.431594	0.434163	0.436732	Tuesday

Процесс фильтрации не зависит от используемого атрибута: в любом случае сначала создается последовательность логических значений, а затем она передается в квадратных скобках после `DataFrame`.

11.5. СЛОЖЕНИЕ И ВЫЧИТАНИЕ ИНТЕРВАЛОВ ВРЕМЕНИ

Складывать и вычитать согласованные интервалы времени можно с помощью объекта `DateOffset`. Его конструктор доступен на верхнем уровне библиотеки pandas, принимает параметры `years`, `months`, `days` и др. Следующий пример моделирует интервал времени протяженностью три года четыре месяца и три дня:

```
In [62] pd.DateOffset(years = 3, months = 4, days = 5)
```

```
Out [62] <DateOffset: days=5, months=4, years=3>
```

Для напоминания, вот как выглядят первые пять строк в наборе данных `disney`:

```
In [63] disney["Date"].head()
```

```
Out [63] 0    1962-01-02
          1    1962-01-03
          2    1962-01-04
          3    1962-01-05
          4    1962-01-08
          Name: Date, dtype: datetime64[ns]
```

Теперь представим, что в нашей системе учета произошел сбой и даты в столбце `Date` оказались смещены на пять дней. Мы можем прибавить постоянное количество времени к каждой дате в последовательности объектов `datetime` с помощью оператора сложения (+) и объекта `DateOffset`. Сложение предполагает «смещение вперед», или «в будущее». В следующем примере к каждой дате в столбце `Date` прибавляется пять дней:

```
In [64] (disney["Date"] + pd.DateOffset(days = 5)).head()
```

```
Out [64] 0    1962-01-07
          1    1962-01-08
          2    1962-01-09
          3    1962-01-10
          4    1962-01-13
          Name: Date, dtype: datetime64[ns]
```

Оператор вычитания (-) в сочетании с `DateOffset` вычитает указанное количество времени из каждой даты в последовательности объектов `datetime`. Вычитание

предполагает «смещение назад», или «в прошлое». В следующем примере каждая дата смещается на три дня назад:

```
In [65] (disney["Date"] - pd.DateOffset(days = 3)).head()
```

```
Out [65] 0    1961-12-30
         1    1961-12-31
         2    1962-01-01
         3    1962-01-02
         4    1962-01-05
         Name: Date, dtype: datetime64[ns]
```

Из приведенного результата неочевидно, но вообще время внутри, за видимым фасадом, хранится в виде объекта `Timestamp`. Когда мы преобразовывали строки в столбце `Date` в значения `datetime`, pandas предположила, что имеется в виду полночь каждой даты. В следующем примере конструктору `DateOffset` передается дополнительный параметр `hours` с заданным значением 6, чтобы прибавить еще некоторое время к каждому значению `datetime` в столбце `Date`. Если теперь вывести содержимое получившейся последовательности `Series`, вы увидите и дату, и время:

```
In [66] (disney["Date"] + pd.DateOffset(days = 10, hours = 6)).head()
```

```
Out [66] 0    1962-01-12 06:00:00
         1    1962-01-13 06:00:00
         2    1962-01-14 06:00:00
         3    1962-01-15 06:00:00
         4    1962-01-18 06:00:00
         Name: Date, dtype: datetime64[ns]
```

Ту же логику pandas применяет при вычитании интервала времени. В следующем примере из каждой даты вычитается один год три месяца десять дней шесть часов и три минуты:

```
In [67] (
        disney["Date"]
        - pd.DateOffset(
            years = 1, months = 3, days = 10, hours = 6, minutes = 3
        )
    ).head()
Out [67] 0    1960-09-21 17:57:00
         1    1960-09-22 17:57:00
         2    1960-09-23 17:57:00
         3    1960-09-24 17:57:00
         4    1960-09-27 17:57:00
         Name: Date, dtype: datetime64[ns]
```

Конструктор `DateOffset` поддерживает, кроме описанных, именованные параметры для передачи секунд, микросекунд и наносекунд. За дополнительной информацией обращайтесь к документации на pandas.

11.6. СМЕЩЕНИЕ ДАТ

Объект `DateOffset` оптимально подходит для прибавления некоторого количества времени к каждой дате или вычитания из нее. Но на практике анализ часто требует более динамичных и разнообразных расчетов. Допустим, что нам нужно округлить каждую дату в наборе до конца текущего месяца. Даты разнятся тем, что каждую из них отделяет разное количество дней от конца месяца, поэтому простого прибавления `DateOffset` недостаточно.

Pandas предлагает готовые объекты, представляющие смещения, для динамических расчетов на основе времени. Они определены в модуле `offsets.py`. Обращения к этим смещениям в нашем коде мы должны предварять префиксом с полным путем: `pd.offsets`.

Одним из примеров подобных трансформаций может служить смещение `MonthEnd`, оно округляет каждую дату до конца следующего месяца. Для напомнимания: вот как выглядят последние пять значений в столбце `Date`:

```
In [68] disney["Date"].tail()

Out [68] 14722    2020-06-26
          14723    2020-06-29
          14724    2020-06-30
          14725    2020-07-01
          14726    2020-07-02
          Name: Date, dtype: datetime64[ns]
```

Объекты смещений в pandas поддерживают синтаксис сложения и вычитания, представленный в разделе 11.5. В следующем примере создается новая последовательность `Series` с датами и временем, округленными до конца месяца. Оператор сложения выполняет смещение вперед во времени — к концу месяца в будущем:

```
In [69] (disney["Date"] + pd.offsets.MonthEnd()).tail()

Out [69] 14722    2020-06-30
          14723    2020-06-30
          14724    2020-07-31
          14725    2020-07-31
          14726    2020-07-31
          Name: Date, dtype: datetime64[ns]
```

При округлении действует непреложное правило: затребованное смещение всегда должно произойти. Pandas не может округлить дату до той же даты, то есть если округляемая дата уже приходится на конец месяца, библиотека округляет ее до конца следующего месяца. Так, например, pandas округлила дату `2020-06-30` в позиции 14724 до `2020-07-31` — конца следующего месяца.

Оператор вычитания выполняет смещение назад во времени. В следующем примере используется смещение `MonthEnd` для округления дат до конца предыдущего месяца. Pandas округляет первые три даты (2020-06-26, 2020-06-29 и 2020-06-30) до 2020-05-31, последнего дня мая, а последние две даты (2020-07-01 и 2020-07-02) — до 2020-06-30, последнего дня июня:

```
In [70] (disney["Date"] - pd.offsets.MonthEnd()).tail()
```

```
Out [70] 14722    2020-05-31
          14723    2020-05-31
          14724    2020-05-31
          14725    2020-06-30
          14726    2020-06-30
          Name: Date, dtype: datetime64[ns]
```

Другой объект смещения, `MonthBegin`, выполняет округление до первой даты месяца. В следующем примере используется оператор `+` для округления каждой даты до начала следующего месяца. Как можно видеть в результатах, pandas округляет первые три даты (2020-06-26, 2020-06-29 и 2020-06-30) до 2020-07-01, начала июля, а последние две даты (2020-07-01 и 2020-07-02) — до 2020-08-01 первого дня августа:

```
In [71] (disney["Date"] + pd.offsets.MonthBegin()).tail()
```

```
Out [71] 14722    2020-07-01
          14723    2020-07-01
          14724    2020-07-01
          14725    2020-08-01
          14726    2020-08-01
          Name: Date, dtype: datetime64[ns]
```

Применение оператора вычитания к смещению `MonthBegin` округляет дату до начала месяца. В следующем примере pandas округляет первые три даты (2020-06-26, 2020-06-29 и 2020-06-30) до начала июня, 2020-06-01, а последнюю дату (2020-07-02) — до начала июля, 2020-07-01. Но обратите внимание на округление даты 2020-07-01 в позиции 14725. Как отмечалось выше, pandas не может округлить дату до той же даты — затребованное смещение всегда должно произойти, поэтому pandas выполняет округление до начала предыдущего месяца, 2020-06-01:

```
In [72] (disney["Date"] - pd.offsets.MonthBegin()).tail()
```

```
Out [72] 14722    2020-06-01
          14723    2020-06-01
          14724    2020-06-01
          14725    2020-06-01
          14726    2020-07-01
          Name: Date, dtype: datetime64[ns]
```

Для расчета рабочего времени доступна специальная группа смещений; их имена начинаются с заглавной буквы **B**. Например, смещение до конца рабочего месяца (**BMonthEnd**) округляет заданную дату до последнего рабочего дня месяца. Рабочими днями считаются понедельник, вторник, среда, четверг и пятница.

Рассмотрим последовательность **Series** с тремя объектами **datetime**. Эти три даты приходятся на четверг, пятницу и субботу соответственно:

```
In [73] may_dates = ["2020-05-28", "2020-05-29", "2020-05-30"]
        end_of_may = pd.Series(pd.to_datetime(may_dates))
        end_of_may
```

```
Out [73] 0    2020-05-28
         1    2020-05-29
         2    2020-05-30
         dtype: datetime64[ns]
```

Сравним смещения **MonthEnd** и **BMonthEnd**. Когда мы прибавляем смещение **MonthEnd**, pandas округляет все три даты до последнего дня мая, **2020-05-31**, независимо от того, попадает ли эта дата на рабочий или на выходной день:

```
In [74] end_of_may + pd.offsets.MonthEnd()
Out [74] 0    2020-05-31
         1    2020-05-31
         2    2020-05-31
         dtype: datetime64[ns]
```

Смещение **BMonthEnd** возвращает другой набор результатов. Последний рабочий день мая 2020 года — пятница, 29 мая. Pandas округляет первую дату, **2020-05-28**, до 29-го числа. Следующая дата, **2020-05-29**, приходится на последний рабочий день месяца. Но pandas не может округлить дату до нее же самой, поэтому она округляет **2020-05-29** до последнего рабочего дня в июне, вторника **2020-06-30**. Последняя дата, **2020-05-30**, — это суббота. В мае не осталось рабочих дней, поэтому pandas округляет эту дату до последнего рабочего дня июня, **2020-06-30**:

```
In [75] end_of_may + pd.offsets.BMonthEnd()

Out [75] 0    2020-05-29
         1    2020-06-30
         2    2020-06-30
         dtype: datetime64[ns]
```

Модуль **pd.offsets** включает также смещения для округления до первого и последнего календарного и рабочего дня в квартале, в году и т. д. Не постесняйтесь исследовать их самостоятельно.

11.7. ОБЪЕКТ TIMEDELTA

Вспомним встроенный объект `timedelta`, упоминавшийся выше в этой главе. Объект `timedelta` моделирует временные интервалы — количество времени, отделяющее два момента. Интервал, например один час, представляет протяженность во времени и не связан ни с какой конкретной датой или временем. Pandas моделирует интервалы с помощью собственного объекта `Timedelta`.

ПРИМЕЧАНИЕ

Эти два объекта легко спутать. `timedelta` встроен в Python, а `Timedelta` — в pandas. Они взаимозаменяемы при использовании с операциями pandas.

Конструктор `Timedelta` доступен на верхнем уровне pandas и принимает именованные параметры, обозначающие единицы времени, такие как `days`, `hours`, `minutes` и `seconds`. В следующем примере создается экземпляр `Timedelta`, моделирующий интервал протяженностью восемь дней семь часов шесть минут и пять секунд:

```
In [76] duration = pd.Timedelta(
        days = 8,
        hours = 7,
        minutes = 6,
        seconds = 5
    )
    duration
```

```
Out [76] Timedelta('8 days 07:06:05')
```

Функция `to_timedelta`, доступная на верхнем уровне pandas, преобразует свой аргумент в объект `Timedelta`. Ей можно передать строку, как это проделано в примере ниже:

```
In [77] duration = pd.to_timedelta("3 hours, 5 minutes, 12 seconds")
```

```
Out [77] Timedelta('0 days 03:05:12')
```

Или же в функцию передается целое число с параметром `unit`. Параметр `unit` объявляет единицу времени, в которой измеряется промежуток времени и которую представляет число. К числу допустимых аргументов относятся `"hour"`, `"day"` и `"minute"`. Объект `Timedelta`, созданный в следующем примере, моделирует пятичасовой интервал:

```
In [78] pd.to_timedelta(5, unit = "hour")
```

```
Out [78] Timedelta('0 days 05:00:00')
```

В вызов функции `to_timedelta` можно передать итерируемый объект, такой как список, чтобы преобразовать его значения в объекты `Timedelta`. Результат в этом случае будет возвращен в виде `TimedeltaIndex` — индекса для хранения интервалов:

```
In [79] pd.to_timedelta([5, 10, 15], unit = "day")
```

```
Out [79] TimedeltaIndex(['5 days', '10 days', '15 days'],
                        dtype='timedelta64[ns]', freq=None)
```

Обычно объекты `Timedelta` получаются в результате вычислений и редко создаются с нуля. Вычитание одной отметки времени из другой, например, автоматически возвращает `Timedelta`:

```
In [80] pd.Timestamp("1999-02-05") - pd.Timestamp("1998-05-24")
```

```
Out [80] Timedelta('257 days 00:00:00')
```

Теперь, познакоившись с объектами `Timedelta`, импортируем наш второй набор данных для этой главы — `deliveries.csv` с информацией о доставке товаров для вымышленной компании. Каждая строка включает дату заказа и дату доставки:

```
In [81] deliveries = pd.read_csv("deliveries.csv")
        deliveries.head()
```

```
Out [81]
```

	order_date	delivery_date
0	5/24/98	2/5/99
1	4/22/92	3/6/98
2	2/10/91	8/26/92
3	7/21/92	11/20/97
4	9/2/93	6/10/98

Потренируемся в преобразовании значений в двух столбцах в объекты `datetime`. Мы, конечно, можем использовать параметр `parse_dates`, но опробуем другой подход. Один из возможных вариантов — вызвать функцию `to_datetime` дважды: для столбца `order_date` и для столбца `delivery_date`, и перезаписать затем результатами существующие столбцы в `DataFrame`:

```
In [82] deliveries["order_date"] = pd.to_datetime(
        deliveries["order_date"]
    )
        deliveries["delivery_date"] = pd.to_datetime(
        deliveries["delivery_date"]
    )
```

Более масштабируемое решение — выполнить итерации по именам столбцов с помощью цикла `for`. Мы можем динамически сослаться на столбец в `deliveries`,

использовать `to_datetime` для создания из него индекса `DatetimeIndex` с отметками времени `Timestamp`, а затем заменить исходный столбец в аргументах функции:

```
In [83] for column in ["order_date", "delivery_date"]:
        deliveries[column] = pd.to_datetime(deliveries[column])
```

Посмотрим, как теперь выглядит содержимое набора данных `deliveries`. Новый формат столбца подтверждает, что мы преобразовали строки в объекты `datetime`:

```
In [84] deliveries.head()
```

```
Out [84]
```

	order_date	delivery_date
0	1998-05-24	1999-02-05
1	1992-04-22	1998-03-06
2	1991-02-10	1992-08-26
3	1992-07-21	1997-11-20
4	1993-09-02	1998-06-10

Рассчитаем продолжительность каждой доставки. Для этого нужно вычесть столбец `order_date` из столбца `delivery_date`, что легко реализуется с `pandas`:

```
In [85] (deliveries["delivery_date"] - deliveries["order_date"]).head()
```

```
Out [85] 0    257 days
         1   2144 days
         2    563 days
         3   1948 days
         4   1742 days
         dtype: timedelta64[ns]
```

`Pandas` возвращает последовательность `Series` объектов `timedelta`. Добавим новую последовательность `Series` в набор данных `deliveries` в виде столбца `duration`:

```
In [86] deliveries["duration"] = (
        deliveries["delivery_date"] - deliveries["order_date"]
    )
        deliveries.head()
```

```
Out [86]
```

	order_date	delivery_date	duration
0	1998-05-24	1999-02-05	257 days
1	1992-04-22	1998-03-06	2144 days
2	1991-02-10	1992-08-26	563 days
3	1992-07-21	1997-11-20	1948 days
4	1993-09-02	1998-06-10	1742 days

Теперь у нас есть два столбца со значениями `Timestamp` и один столбец со значениями `Timedelta`:

```
In [87] deliveries.dtypes
```

```
Out [87] order_date      datetime64[ns]
         delivery_date    datetime64[ns]
         duration         timedelta64[ns]
         dtype: object
```

Объекты `Timedelta` можно прибавлять или вычитать из объектов `Timestamp`. В следующем примере продолжительность доставки в каждой строке вычитается из столбца `delivery_date`. Как и ожидалось, значения в получившейся последовательности `Series` идентичны значениям в столбце `order_date`:

```
In [88] (deliveries["delivery_date"] - deliveries["duration"]).head()
```

```
Out [88] 0    1998-05-24
         1    1992-04-22
         2    1991-02-10
         3    1992-07-21
         4    1993-09-02
         dtype: datetime64[ns]
```

Оператор сложения прибавляет значение `Timedelta` к значению `Timestamp`. Допустим, нужно вычислить, какие даты доставки получились бы, если бы товары доставлялись в два раза дольше. Для этого можно прибавить значения `Timedelta` из столбца `duration` к значениям `Timestamp` в столбце `delivery_date`:

```
In [89] (deliveries["delivery_date"] + deliveries["duration"]).head()
```

```
Out [89] 0    1999-10-20
         1    2004-01-18
         2    1994-03-12
         3    2003-03-22
         4    2003-03-18
         dtype: datetime64[ns]
```

Для сортировки последовательностей `Series` объектов `Timedelta` можно использовать метод `sort_values`. В следующем примере столбец `duration` сортируется в порядке возрастания: от самой короткой продолжительности доставки до самой долгой:

```
In [90] deliveries.sort_values("duration")
```

```
Out [90]
```

	order_date	delivery_date	duration
454	1990-05-24	1990-06-01	8 days
294	1994-08-11	1994-08-20	9 days

```

10 1998-05-10    1998-05-19    9 days
499 1993-06-03   1993-06-13   10 days
143 1997-09-20   1997-10-06   16 days
...      ...      ...      ...
152 1990-09-18   1999-12-19 3379 days
62  1990-04-02   1999-08-16 3423 days
458 1990-02-13   1999-11-15 3562 days
145 1990-03-07   1999-12-25 3580 days
448 1990-01-20   1999-11-12 3583 days

```

```
501 rows x 3 columns
```

К последовательностям `Series` объектов `Timedelta` можно также применять математические методы. Ниже показаны примеры применения трех методов, которые мы использовали на протяжении всей книги: `max` для выбора наибольшего значения, `min` для выбора наименьшего значения и `mean` для вычисления среднего:

```
In [91] deliveries["duration"].max()
```

```
Out [91] Timedelta('3583 days 00:00:00')
```

```
In [92] deliveries["duration"].min()
```

```
Out [92] Timedelta('8 days 00:00:00')
```

```
In [93] deliveries["duration"].mean()
```

```
Out [93] Timedelta('1217 days 22:53:53.532934')
```

А вот еще одна задача. Выберем из набора данных `DataFrame` записи, в которых продолжительность доставки составила больше года. Чтобы сравнить каждое значение в столбце `duration` с фиксированной продолжительностью, можно использовать оператор «больше» (`>`). Фиксированную продолжительность можно указать в виде строки или объекта `Timedelta`. В следующем примере используется строка `"365 days"`:

```
In [94] # Следующие две строки эквивалентны
        (deliveries["duration"] > pd.Timedelta(days = 365)).head()
        (deliveries["duration"] > "365 days").head()
```

```
Out [94] 0      False
         1       True
         2       True
         3       True
         4       True
         Name: Delivery Time, dtype: bool
```

Используем полученную последовательность `Series` логических значений для фильтрации строк с длительностью доставки более 365 дней:

```
In [95] deliveries[deliveries["duration"] > "365 days"].head()
```

```
Out [95]
```

	order_date	delivery_date	duration
1	1992-04-22	1998-03-06	2144 days
2	1991-02-10	1992-08-26	563 days
3	1992-07-21	1997-11-20	1948 days
4	1993-09-02	1998-06-10	1742 days
6	1990-01-25	1994-10-02	1711 days

В операции сравнения можно использовать любую детализацию продолжительности, если необходимо. Следующий пример включает в строку дни, часы и минуты, перечисляя единицы измерения через запятую:

```
In [96] long_time = (
        deliveries["duration"] > "2000 days, 8 hours, 4 minutes"
    )
    deliveries[long_time].head()
```

```
Out [96]
```

	order_date	delivery_date	duration
1	1992-04-22	1998-03-06	2144 days
7	1992-02-23	1998-12-30	2502 days
11	1992-10-17	1998-10-06	2180 days
12	1992-05-30	1999-08-15	2633 days
15	1990-01-20	1998-07-24	3107 days

Напомню, что pandas поддерживает возможность сортировки столбцов со значениями `Timedelta`. Поэтому, чтобы найти самую длинную или самую короткую продолжительность, достаточно вызвать метод `sort_values` последовательности `Series` с продолжительностями.

11.8. УПРАЖНЕНИЯ

А теперь воспользуйтесь счастливой возможностью попрактиковаться в применении идей, представленных в этой главе.

11.8.1. Задачи

Citi Bike NYC — официальная программа проката велосипедов в Нью-Йорке. Жители и туристы могут брать и оставлять велосипеды в сотнях мест по всему городу. Данные о поездках общедоступны и ежемесячно публикуются городом

на странице <https://www.citibikenyc.com/system-data>. Файл `citibike.csv` содержит информацию примерно о 1,9 миллиона поездок на велосипедах, совершенных в июне 2020 года. Чтобы нам с вами не отвлекаться на неважные детали, я упростил структуру набора данных, оставив только два столбца: время начала и конца каждой поездки. Импортируем этот набор и сохраним его в переменной `citi_bike`:

```
In [97] citi_bike = pd.read_csv("citibike.csv")
        citi_bike.head()
```

Out [97]

	start_time	stop_time
0	2020-06-01 00:00:03.3720	2020-06-01 00:17:46.2080
1	2020-06-01 00:00:03.5530	2020-06-01 01:03:33.9360
2	2020-06-01 00:00:09.6140	2020-06-01 00:17:06.8330
3	2020-06-01 00:00:12.1780	2020-06-01 00:03:58.8640
4	2020-06-01 00:00:21.2550	2020-06-01 00:24:18.9650

Значения даты и времени в столбцах `start_time` и `stop_time` включают год, месяц, день, час, минуту, секунду и микросекунду. (*Микросекунда* — это единица времени, равная одной миллионной доле секунды.)

Для вывода сводной информации, включая длину `DataFrame`, типы данных столбцов и занимаемый объем памяти, можно использовать метод `info`. Обратите внимание, что pandas импортирует значения двух столбцов в виде строк:

```
In [98] citi_bike.info()
```

Out [98]

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1882273 entries, 0 to 1882272
Data columns (total 2 columns):
#   Column      Dtype
---  ---
0   start_time  object
1   stop_time   object
dtypes: object(2)
memory usage: 28.7+ MB
```

Вот задачи для этой главы.

1. Преобразуйте строковые значения в столбцах `start_time` и `stop_time` в значения типа `Timestamp`.
2. Подсчитайте количество поездок, совершаемых по дням недели (понедельник, вторник и т. д.). В какой будний день совершается больше всего велопоездки? Используйте столбец `start_time` в качестве отправной точки.

3. Подсчитайте количество поездок за каждую неделю в течение месяца. Для этого округлите все даты в столбце `start_time` до предыдущего или текущего понедельника. Предположим, что каждая неделя начинается в понедельник и заканчивается в воскресенье. Соответственно, первая неделя июня будет начинаться в понедельник 1 июня и заканчиваться в воскресенье 7 июня.
4. Рассчитайте продолжительность каждой поездки и сохраните результаты в новый столбец `duration`.
5. Найдите среднюю продолжительность поездки.
6. Извлеките из набора данных пять самых долгих поездок.

11.8.2. Решения

Пришло время привести решения задач.

1. С преобразованием значений столбцов `start_time` и `end_time` в отметки времени прекрасно справится функция `to_datetime`. Следующий пример кода перебирает список с именами столбцов в цикле `for`, передает каждый столбец функции `to_datetime` и замещает существующий столбец со строками новой последовательностью `Series` с объектами `datetime`:

```
In [99] for column in ["start_time", "stop_time"]:
        citi_bike[column] = pd.to_datetime(citi_bike[column])
```

Давайте снова вызовем метод `info`, чтобы убедиться, что в двух столбцах хранятся объекты `datetime`:

```
In [100] citi_bike.info()
```

```
Out [100]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1882273 entries, 0 to 1882272
Data columns (total 2 columns):
#   Column      Dtype
---  ---
0   start_time  datetime64[ns]
1   stop_time   datetime64[ns]
dtypes: datetime64[ns](2)
memory usage: 28.7 MB
```

2. Подсчет количества поездок в будние дни выполняется в два этапа. Сначала из каждого объекта `datetime` в столбце `start_time` нужно извлечь день недели, а затем подсчитать количество событий, приходящееся на каждый будний день. Метод `dt.day_name` возвращает `Series` с названиями дней недели для каждой даты:


```
In [101] citi_bike["start_time"].dt.day_name().head()
```

```
Out [101] 0    Monday
          1    Monday
          2    Monday
          3    Monday
          4    Monday
          Name: start_time, dtype: object
```

Затем, чтобы подсчитать количество вхождений дней недели, можно вызвать проверенный метод `value_counts` полученной последовательности `Series`. В июне 2020 года самым популярным днем для велопрогулок был вторник:

```
In [102] citi_bike["start_time"].dt.day_name().value_counts()
```

```
Out [102] Tuesday      305833
          Sunday       301482
          Monday       292690
          Saturday     285966
          Friday       258479
          Wednesday    222647
          Thursday     215176
          Name: start_time, dtype: int64
```

3. В следующем задании предлагается сгруппировать даты по неделям. Это легко сделать, округлив дату до предыдущего или текущего понедельника. Вот рациональное решение: использовать атрибут `dayofweek`, чтобы получить последовательность `Series` чисел, где `0` обозначает понедельник, `1` — вторник, `6` — воскресенье и т. д.:

```
In [103] citi_bike["start_time"].dt.dayofweek.head()
```

```
Out [103] 0    0
          1    0
          2    0
          3    0
          4    0
          Name: start_time, dtype: int64
```

Кроме всего прочего, номер дня недели определяет удаленность в днях от ближайшего понедельника. Понедельник, 1 июня, например, имеет значение дня недели, равное `0`. Дата отстоит от ближайшего понедельника на `0` дней. Точно так же вторник, 2 июня, имеет значение `dayofweek`, равное `1`. Дата находится на расстоянии одного дня от ближайшего понедельника (1 июня). Сохраним эту последовательность `Series` в переменной `days_away_from_monday`:

```
In [104] days_away_from_monday = citi_bike["start_time"].dt.dayofweek
```

Вычитая значение `dayofweek` из самой даты, мы эффективно округлим каждую дату до предыдущего понедельника. Полученную последовательность

дней недели можно передать в функцию `to_timedelta` и преобразовать ее в последовательность продолжительностей. Чтобы числовые значения продолжительностей обрабатывались как количество дней, передадим параметр `unit` со значением `"day"`:

```
In [105] citi_bike["start_time"] - pd.to_timedelta(
        days_away_from_monday, unit = "day"
    )

Out [105] 0          2020-06-01 00:00:03.372
          1          2020-06-01 00:00:03.553
          2          2020-06-01 00:00:09.614
          3          2020-06-01 00:00:12.178
          4          2020-06-01 00:00:21.255
          ...
        1882268      2020-06-29 23:59:41.116
        1882269      2020-06-29 23:59:46.426
        1882270      2020-06-29 23:59:47.477
        1882271      2020-06-29 23:59:53.395
        1882272      2020-06-29 23:59:53.901
        Name: start_time, Length: 1882273, dtype: datetime64[ns]
```

Сохраним новую последовательность `Series` в переменной `date_rounded_to_monday`:

```
In [106] dates_rounded_to_monday = citi_bike[
        "start_time"
    ] - pd.to_timedelta(days_away_from_monday, unit = "day")
```

Мы уже прошли половину пути к результату. Теперь округлим даты до понедельников, но метод `value_counts` пока не даст желаемого результата, потому что ненулевые значения времени заставляют pandas считать даты неравными:

```
In [107] dates_rounded_to_monday.value_counts().head()

Out [107] 2020-06-22 20:13:36.208    3
          2020-06-08 17:17:26.335    3
          2020-06-08 16:50:44.596    3
          2020-06-15 19:24:26.737    3
          2020-06-08 19:49:21.686    3
          Name: start_time, dtype: int64
```

Воспользуемся атрибутом `dt.date`, чтобы вычленить последовательность `Series` с датами из каждого значения `datetime`:

```
In [108] dates_rounded_to_monday.dt.date.head()

Out [108] 0          2020-06-01
          1          2020-06-01
```

```

2    2020-06-01
3    2020-06-01
4    2020-06-01
Name: start_time, dtype: object

```

Итак, отделив даты, можно вызвать метод `value_counts` для подсчета вхождений каждого значения. В июне наибольшее количество поездок в неделю было зафиксировано с понедельника 15 июня по воскресенье 21 июня:

```

In [109] dates_rounded_to_monday.dt.date.value_counts()

Out [109] 2020-06-15    481211
          2020-06-08    471384
          2020-06-22    465412
          2020-06-01    337590
          2020-06-29    126676
          Name: start_time, dtype: int64

```

4. Чтобы вычислить продолжительность каждой поездки, можно вычесть столбец `start_time` из столбца `stop_time`. Pandas вернет последовательность с объектами `Timedelta`. Эта последовательность понадобится нам далее, поэтому прикрепим ее к `DataFrame` в виде нового столбца с именем `duration`:

```

In [110] citi_bike["duration"] = (
          citi_bike["stop_time"] - citi_bike["start_time"]
        )
          citi_bike.head()

Out [110]

```

	start_time	stop_time	duration
0	2020-06-01 00:00:03.372	2020-06-01 00:17:46.208	0 days 00:17:42.836000
1	2020-06-01 00:00:03.553	2020-06-01 01:03:33.936	0 days 01:03:30.383000
2	2020-06-01 00:00:09.614	2020-06-01 00:17:06.833	0 days 00:16:57.219000
3	2020-06-01 00:00:12.178	2020-06-01 00:03:58.864	0 days 00:03:46.686000
4	2020-06-01 00:00:21.255	2020-06-01 00:24:18.965	0 days 00:23:57.710000

Обратите внимание, что предыдущая операция вычитания вызвала бы ошибку, если бы в столбцах хранились строки; вот почему необходимо сначала преобразовать их в `datetime`.

5. Пришло время найти среднюю продолжительность всех поездок. Это делается просто: достаточно вызвать метод `mean` для нового столбца `duration`. Средняя продолжительность поездки составила 27 минут 19 секунд:

```

In [111] citi_bike["duration"].mean()

Out [111] Timedelta('0 days 00:27:19.590506853')

```

6. В заключительном задании надо определить пять самых длинных поездок из имеющихся в наборе данных. Одно из возможных решений: отсортировать значения в столбце `duration` в порядке убывания вызовом метода `sort_values`, а затем вызвать метод `head`, чтобы получить первые пять строк. По всей видимости, люди, совершившие эти поездки, забыли зарегистрировать время окончания пользования велосипедом:

```
In [112] citi_bike["duration"].sort_values(ascending = False).head()
```

```
Out [112] 50593    32 days 15:01:54.940000
          98339    31 days 01:47:20.632000
          52306    30 days 19:32:20.696000
          15171    30 days 04:26:48.424000
          149761   28 days 09:24:50.696000
          Name: duration, dtype: timedelta64[ns]
```

Другое возможное решение предлагает метод `nlargest`. Его можно вызвать для столбца `duration` или для всего набора данных `DataFrame`. Давайте попробуем последний вариант:

```
In [113] citi_bike.nlargest(n = 5, columns = "duration")
```

```
Out [113]
```

	start_time	stop_time	duration
50593	2020-06-01 21:30:17...	2020-07-04 12:32:12...	32 days 15:01:54.94...
98339	2020-06-02 19:41:39...	2020-07-03 21:29:00...	31 days 01:47:20.63...
52306	2020-06-01 22:17:10...	2020-07-02 17:49:31...	30 days 19:32:20.69...
15171	2020-06-01 13:01:41...	2020-07-01 17:28:30...	30 days 04:26:48.42...
149761	2020-06-04 14:36:53...	2020-07-03 00:01:44...	28 days 09:24:50.69...

Вот и все: определены пять самых продолжительных поездок из имеющихся в наборе данных. Поздравляю с успешным выполнением заданий!

РЕЗЮМЕ

- Объект `Timestamp` в pandas — гибкая и мощная замена встроенного в Python объекта `datetime`.
- Атрибут `dt` последовательности `Series` объектов `datetime` открывает доступ к объекту `DatetimeProperties` с атрибутами и методами для извлечения дня, месяца, названия дня недели и т. д.
- Объект `Timedelta` моделирует продолжительность.
- При вычитании одного объекта `Timestamp` из другого pandas создает объект `Timedelta`.

- Смещения в пакете `pd.offsets` динамически округляют даты до ближайшей недели, месяца, квартала и т. д. Мы можем округлить вперед, применив операцию сложения, или назад, применив операцию вычитания.
- `DatetimeIndex` — это контейнер для значений `Timestamp`. Его можно добавить в структуру данных `pandas` как индекс или столбец.
- `TimedeltaIndex` служит контейнером для объектов `Timedelta`.
- Функция верхнего уровня `to_datetime` преобразует значения из итерируемого объекта в `DatetimeIndex` с отметками времени `Timestamps`.

12

Импорт и экспорт данных

В этой главе

- ✓ Импорт данных в формате JSON.
- ✓ Преобразование иерархии вложенных записей в плоскую последовательность.
- ✓ Загрузка файла CSV из Интернета.
- ✓ Чтение и запись из книги/в книгу Excel.

Наборы данных могут распространяться в самых разных форматах: в виде значений, разделенных запятыми (Comma-Separated Values, CSV), значений, разделенных табуляцией (Tab-Separated Values, TSV), книг Excel (XLSX) и др. В некоторых форматах данные хранятся не в табличном виде, а в форме наборов пар данных «ключ/значение». Рассмотрим следующие два примера. На рис. 12.1 показаны данные, хранимые в таблице, а на рис. 12.2 — те же данные, но в словаре Python.

Year	Award	Winner
2000	Best Actor	Russel Crows
2000	Best Actress	Julia Roberts
2001	Best Actor	Denzel Washington
2001	Best Actress	Halle Berry

Рис. 12.1. Победители премии «Оскар»

Как и обещал, привожу словарь Python, который иллюстрирует ту же структуру данных в виде набора пар «ключ/значение»:

```
{
    2000: [
        {
            "Award": "Best Actor",
            "Winner": "Russell Crowe"
        },
        {
            "Award": "Best Actress",
            "Winner": "Julia Roberts"
        }
    ],
    2001: [
        {
            "Award": "Best Actor",
            "Winner": "Denzel Washington"
        },
        {
            "Award": "Best Actress",
            "Winner": "Halle Berry"
        }
    ]
}
```

Рис. 12.2. Словарь Python (хранилище пар «ключ/значение») с теми же данными

Pandas включает вспомогательные функции для преобразования данных «ключ/значение» в табличную форму и наоборот. К данным из набора `DataFrame` мы можем применить все доступные методы. Но преобразование данных в подходящую для анализа форму часто оказывается самой сложной частью анализа. В этой главе вы узнаете, как решать типичные проблемы, возникающие при импорте данных, мы также рассмотрим другую сторону медали: экспорт наборов данных `DataFrame` в различные типы файлов и структуры данных.

12.1. ЧТЕНИЕ И ЗАПИСЬ ФАЙЛОВ JSON

Начнем наш разговор с обсуждения JSON, пожалуй, самого популярного формата хранения пар «ключ/значение» из доступных на сегодняшний день. *Форма записи объектов JavaScript* (JavaScript Object Notation, JSON) — это текстовый формат для хранения и передачи данных. Несмотря на то что синтаксис формата JSON заимствован из языка программирования JavaScript, он никак не зависит от языка. В настоящее время большинство языков программирования, включая Python, имеют поддержку чтения и записи данных в формате JSON.

Данные в формате JSON состоят из пар «ключ/значение», в которых ключ служит уникальным идентификатором значения. Символ двоеточия (:) связывает ключ со значением:

```
"name": "Harry Potter"
```

Ключи должны быть текстовыми строками. Значения могут быть данными любого типа, включая строки, числа и логические значения. JSON похож на объект словаря в Python.

JSON — это популярный формат обмена данными, он используется во многих современных прикладных программных интерфейсах (Application Programming Interfaces, API), таких, например, как серверы сайтов. Ответ API в формате JSON выглядит как простая строка, например:

```
{"name": "Harry Potter", "age": 17, "wizard": true}
```

Программы статического анализа кода, называемые *линтерами* (linter), форматируют ответы JSON, помещая каждую пару «ключ/значение» в отдельную строку. Одним из популярных примеров такого ПО является JSONLint (<https://jsonlint.com>). Если строку JSON из предыдущего примера передать JSONLint, то на выходе получится такой результат:

```
{
  "name": "Harry Potter",
  "age": 17,
  "wizard": true,
}
```

Технически это и предыдущее представления эквивалентны друг другу, но последнее проще читается.

Ответ в формате JSON содержит три пары «ключ/значение»:

- ключ "name" имеет строковое значение "Harry Potter";
- ключ "age" имеет целочисленное значение 17;
- ключ "wizard" имеет логическое значение true.

В JSON логические значения записываются строчными буквами, но концептуально они идентичны логическим значениям в Python.

Ключ также может указывать на *массив*, упорядоченный набор элементов, эквивалентный списку в Python. Так, ключу "friends" в следующем примере соответствует массив из двух строк:

```
{
  "name": "Harry Potter",
  "age": 17,
```



```

    "wizard": true,
    "friends": ["Ron Weasley", "Hermione Granger"],
}

```

JSON может хранить дополнительные пары «ключ/значение» во вложенных объектах, как, например, в ключе "address" в следующем примере. Мы, программисты на Python, можем рассматривать "address" как словарь, вложенный в другой словарь:

```

{
    "name": "Harry Potter",
    "age": 17,
    "wizard": true,
    "friends": ["Ron Weasley", "Hermione Granger"],
    "address": {
        "street": "4 Privet Drive",
        "town": "Little Whinging"
    }
}

```

Вложенные наборы пар «ключ/значение» помогают упростить данные, консолидируя в себе связанные поля.

12.1.1. Загрузка файла JSON в DataFrame

А теперь создадим новый блокнот Jupyter и импортируем библиотеку pandas. Создайте блокнот в том же каталоге, где находятся файлы с данными для этой главы:

```
In [1] import pandas as pd
```

Данные в формате JSON можно хранить в текстовом файле с расширением .json. Файл prizes.json в этой главе — это ответ в формате JSON, полученный от Nobel Prize API. Этот API сообщает имена лауреатов Нобелевской премии начиная с 1901 года. Вы можете просмотреть исходный ответ API в формате JSON в своем браузере, перейдя по адресу <http://api.nobelprize.org/v1/prize.json>. Вот как выглядят данные JSON в форме предварительного просмотра:

```

{
  "prizes": [
    {
      "year": "2019",
      "category": "chemistry",
      "laureates": [
        {
          "id": "976",
          "firstname": "John",
          "surname": "Goodenough",
          "motivation": "\"for the development of lithium-ion batteries\"",
          "share": "3"
        }
      ]
    }
  ]
}

```

```

{
  "id": "977",
  "firstname": "M. Stanley",
  "surname": "Whittingham",
  "motivation": "\"for the development of lithium-ion batteries\"",
  "share": "3"
},
{
  "id": "978",
  "firstname": "Akira",
  "surname": "Yoshino",
  "motivation": "\"for the development of lithium-ion batteries\"",
  "share": "3"
}
]
},

```

На верхнем уровне эта структура JSON содержит ключ "prizes", которому соответствует массив словарей, по одному для каждой комбинации «год/категория» ("chemistry", "physics", "literature" и т. д.). Ключи "year" и "category" указаны для всех лауреатов, тогда как ключи "laureates" и "overallMotivation" — только для некоторых. Вот пример словаря с ключом "overallMotivation":

```

{
  year: "1972",
  category: "peace",
  overallMotivation: "No Nobel Prize was awarded this year. The prize
    money for 1972 was allocated to the Main Fund."
}

```

Ключу "laureates" соответствует массив словарей, каждый со своими ключами "id", "firstname", "surname", "motivation" и "share". Ключ "laureates" позволяет хранить массив для тех лет, когда несколько человек были удостоены Нобелевской премии в одной и той же категории. Значением ключа "laureates" является массив, даже если в данном году был выбран только один победитель. Например:

```

{
  year: "2019",
  category: "literature",
  laureates: [
    {
      id: "980",
      firstname: "Peter",
      surname: "Handke",
      motivation: "for an influential work that with linguistic
        ingenuity has explored the periphery and the specificity of
        human experience",
      share: "1"
    }
  ]
},

```

Имена функций импорта в pandas следуют согласованной схеме именования; каждое имя начинается с префикса `read`, за которым следует тип файла. Например, выше мы неоднократно использовали функцию `read_csv` для импорта файлов `.csv`. Чтобы импортировать файл JSON, мы будем использовать функцию `read_json`. В первом параметре она принимает в качестве аргумента путь к файлу. В следующем примере в эту функцию передается файл `nobel.json`. В ответ она возвращает набор данных `DataFrame` с единственным столбцом `prizes`:

```
In [2] nobel = pd.read_json("nobel.json")
      nobel.head()
```

```
Out [2]
```

```

                                     prizes
-----
0  {'year': '2019', 'category': 'chemistry', 'laureates': [{'id': '97...
1  {'year': '2019', 'category': 'economics', 'laureates': [{'id': '98...
2  {'year': '2019', 'category': 'literature', 'laureates': [{'id': '9...
3  {'year': '2019', 'category': 'peace', 'laureates': [{'id': '981', ...
4  {'year': '2019', 'category': 'physics', 'overallMotivation': '"for...
```

Мы благополучно импортировали файл в pandas, но, к сожалению, анализировать набор данных в этом формате очень неудобно. Pandas создала столбец `prizes` для ключа верхнего уровня и для каждого его значения сгенерировала словарь с парами «ключ/значение», извлеченными из файла JSON. Вот пример одной записи в получившемся наборе данных:

```
In [3] nobel.loc[2, "prizes"]
```

```
Out [3] {'year': '2019',
        'category': 'literature',
        'laureates': [{'id': '980',
                        'firstname': 'Peter',
                        'surname': 'Handke',
                        'motivation': '"for an influential work that with linguistic
                                ingenuity has explored the periphery and the specificity of
                                human experience"',
                        'share': '1'}]}
```

А в следующем примере значение этой записи передается встроенной функции `type`. Таким образом, фактически мы получаем последовательность `Series` словарей:

```
In [4] type(nobel.loc[2, "prizes"])
```

```
Out [4] dict
```

Наша следующая цель — преобразовать данные в табличный формат. Для этого извлечем пары «ключ/значение» на верхнем уровне JSON (`year`, `category`), чтобы разделить данные по столбцам в `DataFrame`. Нужно также проанализировать каждый словарь в списке `"laureates"` и извлечь из него вложенную информацию, чтобы в конечном итоге получить по одной строке, включающей год и категорию,

для каждого нобелевского лауреата. Вот как выглядит `DataFrame`, который мы должны получить:

	id	firstname	surname	motivation	share	year	category
0	976	John	Goodenough	"for the develop...	3	2019	chemistry
1	977	M. Stanley	Whittingham	"for the develop...	3	2019	chemistry
2	978	Akira	Yoshino	"for the develop...	3	2019	chemistry

Процесс перемещения вложенных записей в единый одномерный список называется *преобразованием в плоскую последовательность* или *нормализацией*. Библиотека pandas включает встроенную функцию `json_normalize`, которая возьмет на себя всю тяжелую работу. Давайте опробуем ее на небольшом примере — на словаре из набора данных `nobel`. Извлечем словарь из первой строки с помощью метода `loc` и сохраним его в переменной `chemistry_2019`:

```
In [5] chemistry_2019 = nobel.loc[0, "prizes"]
      chemistry_2019
```

```
Out [5] {'year': '2019',
        'category': 'chemistry',
        'laureates': [{'id': '976',
                        'firstname': 'John',
                        'surname': 'Goodenough',
                        'motivation': '"for the development of lithium-ion batteries"',
                        'share': '3'},
                      {'id': '977',
                        'firstname': 'M. Stanley',
                        'surname': 'Whittingham',
                        'motivation': '"for the development of lithium-ion batteries"',
                        'share': '3'},
                      {'id': '978',
                        'firstname': 'Akira',
                        'surname': 'Yoshino',
                        'motivation': '"for the development of lithium-ion batteries"',
                        'share': '3'}]}
```

Передадим словарь `chemistry_2019` функции `json_normalize` в параметр `data`. Хорошая новость: pandas извлекает из словаря три ключа верхнего уровня ("`year`", "`category`" и "`laureates`") для разделения столбцов в новом наборе данных. Но, к сожалению, она все так же единой строкой сохраняет вложенные словари из списка "`laureates`", тогда как нам хотелось бы разделить данные по отдельным столбцам.

```
In [6] pd.json_normalize(data = chemistry_2019)
```

```
Out [6]
```

	year	category	laureates
0	2019	chemistry	[{'id': '976', 'firstname': 'John', 'surname': '...

Для нормализации вложенных записей в "laureates" можно использовать параметр `record_path` функции `json_normalize`. В этом параметре должна передаваться строка с ключом в словаре, причем ключ должен содержать вложенные записи. Передадим "laureates":

```
In [7] pd.json_normalize(data = chemistry_2019, record_path = "laureates")
```

```
Out [7]
```

	id	firstname	surname	motivation	share
0	976	John	Goodenough	"for the development of li...	3
1	977	M. Stanley	Whittingham	"for the development of li...	3
2	978	Akira	Yoshino	"for the development of li...	3

Хорошо, да не очень. Pandas развернула вложенные словари "laureates" в новые столбцы, но теперь мы потеряли исходные столбцы "year" и "category". Чтобы сохранить эти пары «ключ/значение» верхнего уровня, передадим список с их именами в параметре `meta`:

```
In [8] pd.json_normalize(
        data = chemistry_2019,
        record_path = "laureates",
        meta = ["year", "category"],
    )
```

```
Out [8]
```

	id	firstname	surname	motivation	share	year	category
0	976	John	Goodenough	"for the develop...	3	2019	chemistry
1	977	M. Stanley	Whittingham	"for the develop...	3	2019	chemistry
2	978	Akira	Yoshino	"for the develop...	3	2019	chemistry

Вот теперь получен именно тот **DataFrame**, который нам нужен. Наша стратегия нормализации успешно справилась с единым словарем из столбца `prizes`. К счастью, функция `json_normalize` достаточно интеллектуальна, чтобы принять последовательность словарей и повторить логику извлечения для каждого элемента. Посмотрим, что получится, если передать функции последовательность `prizes`:

```
In [9] pd.json_normalize(
        data = nobel["prizes"],
        record_path = "laureates",
        meta = ["year", "category"]
    )
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-49-e09a24c19e5b> in <module>
      2     data = nobel["prizes"],
```

```

3     record_path = "laureates",
----> 4     meta = ["year", "category"]
5 )

```

```
KeyError: 'laureates'
```

Увы! Pandas сгенерировала исключение `KeyError`. В некоторых словарях в последовательности `prizes` отсутствует ключ `"laureates"`, в результате функция `json_normalize` не смогла извлечь информацию о лауреатах из несуществующего списка. Один из способов решить эту проблему — идентифицировать словари с отсутствующим ключом `"laureates"` и добавить этот ключ вручную. В таких случаях значение ключа `"laureates"` можно инициализировать пустым списком.

На секунду отвлечемся и рассмотрим метод `setdefault` словарей в языке Python. Возьмем для примера такой словарь:

```

In [10] cheese_consumption = {
        "France": 57.9,
        "Germany": 53.2,
        "Luxembourg": 53.2
    }

```

Метод `setdefault` добавляет в словарь пару «ключ/значение», но только если в словаре нет указанного ключа. Если ключ существует, метод возвращает его существующее значение. В первом аргументе метод принимает ключ, а во втором — значение.

Код следующего примера пытается добавить в словарь `cheese_consumption` ключ `"France"` со значением `100`. Данный ключ уже существует, поэтому ничего не меняется. Python сохраняет исходное значение `57.9`:

```
In [11] cheese_consumption.setdefault("France", 100)
```

```
Out [11] 57.9
```

```
In [12] cheese_consumption["France"]
```

```
Out [12] 57.9
```

Для сравнения: пример ниже вызывает `setdefault` с аргументом `"Italy"`. Ключа `"Italy"` в словаре нет, поэтому Python добавляет его и присваивает ему значение `48`:

```
In [13] cheese_consumption.setdefault("Italy", 48)
```

```
Out [13] 48
```

```
In [14] cheese_consumption
```

```
Out [14] {'France': 57.9, 'Germany': 53.2, 'Luxembourg': 53.2, 'Italy': 48}
```

Применим этот прием к каждому словарию внутри последовательности `prizes`. Вызовем метод `setdefault` для каждого словаря, чтобы добавить ключ `"laureates"` с пустым списком в качестве значения, если этот ключ отсутствует. Напомню, что для обхода элементов последовательности `Series` можно использовать метод `apply`. Этот метод, представленный в главе 3, принимает функцию и последовательно передает ей каждый элемент в `Series`. В следующем примере определяется функция `add_laureates_key`, обновляющая один словарь, затем она же передается методу `apply` в качестве аргумента:

```
In [15] def add_laureates_key(entry):
        entry.setdefault("laureates", [])

        nobel["prizes"].apply(add_laureates_key)

Out [15] 0      [{'id': '976', 'firstname': 'John', 'surname':...
1      [{'id': '982', 'firstname': 'Abhijit', 'surnam...
2      [{'id': '980', 'firstname': 'Peter', 'surname'...
3      [{'id': '981', 'firstname': 'Abiy', 'surname':...
4      [{'id': '973', 'firstname': 'James', 'surname'...
        ...
641     [{'id': '160', 'firstname': 'Jacobus H.', 'sur...
642     [{'id': '569', 'firstname': 'Sully', 'surname'...
643     [{'id': '462', 'firstname': 'Henry', 'surname'...
644     [{'id': '1', 'firstname': 'Wilhelm Conrad', 's...
645     [{'id': '293', 'firstname': 'Emil', 'surname':...
Name: prizes, Length: 646, dtype: object
```

Метод `setdefault` изменяет словари прямо внутри `prizes`, поэтому нет необходимости перезаписывать исходную последовательность.

Теперь, когда все словари имеют ключ `"laureates"`, можно вновь вызвать функцию `json_normalize`. Снова передадим ей в параметре `meta` список с двумя ключами верхнего уровня, которые хотим сохранить. Кроме того, используем параметр `record_path`, чтобы указать атрибут верхнего уровня с вложенным списком записей:

```
In [16] winners = pd.json_normalize(
        data = nobel["prizes"],
        record_path = "laureates",
        meta = ["year", "category"]
    )

    winners
```

```
Out [16]
```

	id	firstname	surname	motivation	share	year	category
0	976	John	Goodenough	"for the de...	3	2019	chemistry
1	977	M. Stanley	Whittingham	"for the de...	3	2019	chemistry

2	978	Akira	Yoshino	"for the de...	3	2019	chemistry
3	982	Abhijit	Banerjee	"for their ...	3	2019	economics
4	983	Esther	Duflo	"for their ...	3	2019	economics
...
945	569	Sully	Prudhomme	"in special...	1	1901	literature
946	462	Henry	Dunant	"for his hu...	2	1901	peace
947	463	Frédéric	Passy	"for his li...	2	1901	peace
948	1	Wilhelm Con...	Röntgen	"in recogni...	1	1901	physics
949	293	Emil	von Behring	"for his wo...	1	1901	medicine

950 rows × 7 columns

Ура! Мы нормализовали данные, полученные в формате JSON, преобразовав их в табличный формат и сохранив в двумерном наборе данных `DataFrame`.

12.1.2. Экспорт содержимого `DataFrame` в файл JSON

Теперь попробуем выполнить обратный процесс: преобразовать содержимое `DataFrame` в формат JSON и записать его в файл. Метод `to_json` принимает структуру данных pandas и создает строку в формате JSON; его параметр `orient` задает формат, в котором pandas должна вернуть данные. В следующем примере в этом параметре передается аргумент `"records"`, чтобы получить массив JSON-объектов с парами «ключ/значение». Pandas хранит имена столбцов в виде ключей словаря с соответствующими значениями. Вот пример преобразования первых двух строк в наборе данных `winners`, который мы создали в подразделе 12.1.1:

In [17] `winners.head(2)`

Out [17]

	id	firstname	surname	motivation	share	year	category
0	976	John	Goodenough	"for the develop...	3	2019	chemistry
1	977	M. Stanley	Whittingham	"for the develop...	3	2019	chemistry

In [18] `winners.head(2).to_json(orient = "records")`

Out [18]

```
'[{"id": "976", "firstname": "John", "surname": "Goodenough", "motivation": "\\nfor the development of lithium-ion batteries\\n", "share": "3", "year": "2019", "category": "chemistry"}, {"id": "977", "firstname": "M. Stanley", "surname": "Whittingham", "motivation": "\\nfor the development of lithium-ion batteries\\n", "share": "3", "year": "2019", "category": "chemistry"}]'
```


Для сравнения попробуем передать аргумент `"split"`, чтобы вернуть словарь с отдельными ключами `columns`, `index` и `data`. Этот аргумент предотвращает дублирование имен столбцов в каждой записи:

```
In [19] winners.head(2).to_json(orient = "split")
```

```
Out [19]
```

```
'{"columns":["id","firstname","surname","motivation","share","year","category"],"index":[0,1],"data":[["976","John","Goodenough","\\"for the development of lithium-ion batteries\\"","3","2019","chemistry"],["977","M. Stanley","Whittingham","\\"for the development of lithium-ion batteries\\"","3","2019","chemistry"]}]'
```

В параметре `orient` также можно передать аргументы `"index"`, `"columns"`, `"values"` и `"table"`.

Если получившийся формат соответствует вашим ожиданиям, то просто передайте методу `to_json` имя файла в качестве первого аргумента. Pandas запишет строку в файл JSON в том же каталоге, где был открыт блокнот Jupyter:

```
In [20] winners.to_json("winners.json", orient = "records")
```

ВНИМАНИЕ

Будьте осторожны, выполняя одну и ту же ячейку блокнота дважды. Если в каталоге существует файл `winners.json`, то pandas перезапишет его, когда повторно выполнится предыдущая ячейка. Библиотека не предупредит о замене файла. По этой причине я настоятельно рекомендую давать выходным файлам имена, отличные от имен входных файлов.

12.2. ЧТЕНИЕ И ЗАПИСЬ ФАЙЛОВ CSV

Наш следующий набор данных содержит имена детей в Нью-Йорке. Каждая строка включает имя, год рождения, пол, этническую принадлежность, количество (сколько раз выбиралось это имя) и рейтинг популярности. Файл CSV, который мы используем, доступен на веб-сайте правительства Нью-Йорка <http://mng.bz/MgzQ>.

Вы можете открыть указанную веб-страницу в браузере и загрузить файл CSV на свой компьютер. С другой стороны, можно передать URL файла в качестве первого аргумента функции `read_csv`. Pandas автоматически извлечет набор данных и импортирует его в `DataFrame`. Жестко заданные URL удобны, когда требуется извлекать часто меняющиеся данные в реальном времени, потому что они избавляют от ручной работы по загрузке набора данных каждый раз, когда требуется повторно запустить анализ:

```
In [21] url = "https://data.cityofnewyork.us/api/views/25th-nujf/rows.csv"
        baby_names = pd.read_csv(url)
        baby_names.head()
```

```
Out [21]
```

	Year of Birth	Gender	Ethnicity	Child's First Name	Count	Rank
0	2011	FEMALE	HISPANIC	GERALDINE	13	75
1	2011	FEMALE	HISPANIC	GIA	21	67
2	2011	FEMALE	HISPANIC	GIANNA	49	42
3	2011	FEMALE	HISPANIC	GISELLE	38	51
4	2011	FEMALE	HISPANIC	GRACE	36	53

Обратите внимание, что если pandas передать недействительную ссылку, то она сгенерирует исключение `HTTPError`.

Попробуем записать набор данных `baby_names` в простой файл CSV вызовом метода `to_csv`. Если этот метод вызвать без аргумента, то он выведет строку CSV непосредственно в блокнот Jupyter. Следуя соглашениям CSV, pandas разделяет строки символами перевода строки, а значения в строках — запятыми. Напомню, что роль символа перевода строки в Python играет символ `\n`. Вот небольшой пример вывода метода с первыми десятью строками из `baby_names`:

```
In [22] baby_names.head(10).to_csv()
```

```
Out [22]
```

```
",Year of Birth,Gender,Ethnicity,Child's First
Name,Count,Rank\n0,2011,FEMALE,HISPANIC,GERALDINE,13,75\n1,2011,FEMALE,H
ISPANIC,GIA,21,67\n2,2011,FEMALE,HISPANIC,GIANNA,49,42\n3,2011,FEMALE,HI
SPANIC,GISELLE,38,51\n4,2011,FEMALE,HISPANIC,GRACE,36,53\n5,2011,FEMALE,
HISPANIC,GUADALUPE,26,62\n6,2011,FEMALE,HISPANIC,HAILEY,126,8\n7,2011,FE
MALE,HISPANIC,HALEY,14,74\n8,2011,FEMALE,HISPANIC,HANNAH,17,71\n9,2011,F
EMALE,HISPANIC,HAYLEE,17,71\n"
```

По умолчанию pandas включает в строку CSV индекс `DataFrame`. Обратите внимание на запятую в начале строки и числовые значения (0, 1, 2 и т. д.) после каждого символа `\n`. На рис. 12.3 стрелками выделены запятые в выводе метода `to_csv`.



```
",Year of Birth,Gender,Ethnicity,Child's First
Name,Count,Rank\n0,2011,FEMALE,HISPANIC,GERALDINE,13,75\n1,2011,FEMALE,HISP
ANIC,GIA,21,67\n2,2011,FEMALE,HISPANIC,GIANNA,49,42\n3,2011,FEMALE,HISPANIC
,GISELLE,38,51\n4,2011,FEMALE,HISPANIC,GRACE,36,53\n5,2011,FEMALE,HISPANIC,
GUADALUPE,26,62\n6,2011,FEMALE,HISPANIC,HAILEY,126,8\n7,2011,FEMALE,HISPANI
C,HALEY,14,74\n8,2011,FEMALE,HISPANIC,HANNAH,17,71\n9,2011,FEMALE,HISPANIC,
HAYLEE,17,71\n"
```

Рис. 12.3. Вывод в формате CSV со стрелками, указывающими на индексные метки

Индекс можно исключить из вывода, если передать в параметре `index` аргумент `False`:

```
In [23] baby_names.head(10).to_csv(index = False)
```

```
Out [23]
```

```
"Year of Birth,Gender,Ethnicity,Child's First
Name,Count,Rank\n2011,FEMALE,HISPANIC,GERALDINE,13,75\n2011,FEMALE,HISPA
NIC,GIA,21,67\n2011,FEMALE,HISPANIC,GIANNA,49,42\n2011,FEMALE,HISPANIC,G
ISELLE,38,51\n2011,FEMALE,HISPANIC,GRACE,36,53\n2011,FEMALE,HISPANIC,GUA
DALUPE,26,62\n2011,FEMALE,HISPANIC,HAILEY,126,8\n2011,FEMALE,HISPANIC,HA
LEY,14,74\n2011,FEMALE,HISPANIC,HANNAH,17,71\n2011,FEMALE,HISPANIC,HAYLE
E,17,71\n"
```

Чтобы записать строку в файл CSV, нужно передать методу `to_csv` желаемое имя файла в качестве первого аргумента. Обязательно включите в имя файла расширение `.csv`. Если путь к файлу не указан, то pandas сохранит файл в тот же каталог, где открыт блокнот Jupyter:

```
In [24] baby_names.to_csv("NYC_Baby_Names.csv", index = False)
```

В этом случае метод ничего не выводит в следующую ячейку блокнота. Однако если перейти в интерфейс навигации Jupyter Notebook, то можно увидеть, что pandas создала файл CSV. На рис. 12.4 показан сохраненный файл `NYC_Baby_Names.csv`.



Рис. 12.4. Файл `NYC_Baby_Names.csv`, сохраненный в том же каталоге, где был открыт блокнот Jupyter

По умолчанию pandas записывает в файл CSV все столбцы из `DataFrame`. Но есть также возможность и явно указать, какие столбцы экспортировать, передав список их имен в параметр `columns`. В примере ниже создается CSV-файл, включающий только столбцы `Gender`, `Child's First Name` и `Count`:

```
In [25] baby_names.to_csv(
        "NYC_Baby_Names.csv",
        index = False,
        columns = ["Gender", "Child's First Name", "Count"]
    )
```

Обратите внимание, что если в каталоге уже существует файл `NYC_Baby_Names.csv`, то pandas перезапишет его.

12.3. ЧТЕНИЕ КНИГ EXCEL И ЗАПИСЬ В НИХ

В настоящее время Excel — самое популярное приложение для работы с электронными таблицами. Pandas упрощает чтение книг Excel и запись в них и даже на определенные их листы. Но эта операция требует небольшой предварительной подготовки, в частности, необходимо интегрировать два программных компонента.

12.3.1. Установка библиотек xlrd и openpyxl в среде Anaconda

Для взаимодействия с Excel нужны библиотеки `xlrd` и `openpyxl`. Они служат связующим звеном между Python и Excel.

Вот краткое напоминание порядка установки пакета в среде Anaconda. Более подробный обзор вы найдете в приложении А. Если вы уже установили эти библиотеки у себя, то смело переходите к подразделу 12.3.2.

1. Запустите приложение Terminal (в macOS) или Anaconda Prompt (в Windows).
2. Выполните команду `conda info --envs`, чтобы просмотреть доступное окружение Anaconda:

```
$ conda info --envs

# conda environments:
#
base                  *  /opt/anaconda3
pandas_in_action      /opt/anaconda3/envs/pandas_in_action
```

3. Активируйте среду Anaconda, в которую вы предполагаете установить библиотеки. В приложении А показано, как создать среду `pandas_in_action` для этой книги. Если вы выбрали другое имя для своего окружения, то подставьте его вместо `pandas_in_action` в следующей команде:

```
$ conda activate pandas_in_action
```

4. Установите библиотеки `xlrd` и `openpyxl` с помощью команды `conda install`:

```
(pandas_in_action) $ conda install xlrd openpyxl
```

5. Когда Anaconda закончит перечислять необходимые зависимости пакетов, введите `Y` и нажмите `Enter`, чтобы начать установку.
6. По завершении установки выполните команду `jupyter notebook`, чтобы снова запустить сервер Jupyter, и вернитесь в блокнот Jupyter для этой главы.

Не забудьте выполнить ячейку с командой `import pandas as pd` вверху.

12.3.2. Импорт книг Excel

Функция `read_excel`, доступная на верхнем уровне `pandas`, импортирует книгу Excel в `DataFrame`. В первом параметре `io` она принимает строковый аргумент — путь к книге. Обязательно включите расширение `.xlsx` в имя файла. По умолчанию `pandas` импортирует только первый лист из книги.

Отличной отправной точкой нам послужит книга `Excel Single Worksheet.xlsx`, потому что она содержит только один лист `Data`:

```
In [26] pd.read_excel("Single Worksheet.xlsx")
```

```
Out [26]
```

	First Name	Last Name	City	Gender
0	Brandon	James	Miami	M
1	Sean	Hawkins	Denver	M
2	Judy	Day	Los Angeles	F
3	Ashley	Ruiz	San Francisco	F
4	Stephanie	Gomez	Portland	F

Функция `read_excel` поддерживает множество тех же параметров, что и `read_csv`, в том числе `index_col` для выбора столбцов индекса, `usecols` для выбора столбцов и `squeeze` для преобразования `DataFrame` с одним столбцом в объект `Series`. В следующем примере на роль индекса выбирается столбец `City` и сохраняются только три из четырех столбцов. Обратите внимание, что если некоторый столбец передается в параметре `index_col`, то он также должен передаваться в списке `usecols`:

```
In [27] pd.read_excel(
        io = "Single Worksheet.xlsx",
        usecols = ["City", "First Name", "Last Name"],
        index_col = "City"
    )
```

```
Out [27]
```

City	First Name	Last Name
Miami	Brandon	James
Denver	Sean	Hawkins
Los Angeles	Judy	Day
San Francisco	Ashley	Ruiz
Portland	Stephanie	Gomez

Сложность немного увеличивается, когда книга содержит несколько листов. Книга `Multiple Worksheets.xlsx`, например, содержит три листа: `Data 1`, `Data 2` и `Data 3`. По умолчанию `pandas` импортирует из книги только первый лист:

382 Часть II. Библиотека pandas на практике

```
In [28] pd.read_excel("Multiple Worksheets.xlsx")
```

```
Out [28]
```

	First Name	Last Name	City	Gender
0	Brandon	James	Miami	M
1	Sean	Hawkins	Denver	M
2	Judy	Day	Los Angeles	F
3	Ashley	Ruiz	San Francisco	F
4	Stephanie	Gomez	Portland	F

В процессе импорта данных pandas присваивает каждому листу индекс, начиная с 0. Мы можем импортировать конкретный лист, передав индекс листа или его имя в параметре `sheet_name`. По умолчанию этот параметр получает значение 0 (первый лист). Поэтому следующие две инструкции вернут один и тот же `DataFrame`:

```
In [29] # The two lines below are equivalent
pd.read_excel("Multiple Worksheets.xlsx", sheet_name = 0)
pd.read_excel("Multiple Worksheets.xlsx", sheet_name = "Data 1")
```

```
Out [29]
```

	First Name	Last Name	City	Gender
0	Brandon	James	Miami	M
1	Sean	Hawkins	Denver	M
2	Judy	Day	Los Angeles	F
3	Ashley	Ruiz	San Francisco	F
4	Stephanie	Gomez	Portland	F

Чтобы импортировать все листы, можно передать в параметре `sheet_name` аргумент `None`. Pandas сохранит каждый рабочий лист в отдельном `DataFrame`. Функция `read_excel` возвращает словарь с именами листов в качестве ключей и соответствующими наборами данных `DataFrame` в качестве значений:

```
In [30] workbook = pd.read_excel(
    "Multiple Worksheets.xlsx", sheet_name = None
)

workbook
```

```
Out [30] {'Data 1':   First Name Last Name      City Gender
0      Brandon    James      Miami      M
1        Sean  Hawkins    Denver      M
2        Judy     Day  Los Angeles      F
3     Ashley    Ruiz  San Francisco      F
4  Stephanie    Gomez    Portland      F,
```

```
'Data 2':  First Name Last Name      City Gender
0    Parker    Power      Raleigh    F
1    Preston  Prescott  Philadelphia  F
2    Ronaldo  Donaldo      Bangor      M
3    Megan    Stiller   San Francisco  M
4    Bustin   Jieber    Austin      F,
'Data 3':  First Name Last Name      City Gender
0    Robert   Miller   Seattle     M
1    Tara     Garcia  Phoenix     F
2    Raphael  Rodriguez Orlando    M}
```

```
In [31] type(workbook)
```

```
Out [31] dict
```

Чтобы получить доступ к `DataFrame` с листом, нужно обратиться к ключу в этом словаре. Итак, извлекаем `DataFrame` с листом `Data 2`:

```
In [32] workbook["Data 2"]
```

```
Out [32]
```

	First Name	Last Name	City	Gender
0	Parker	Power	Raleigh	F
1	Preston	Prescott	Philadelphia	F
2	Ronaldo	Donaldo	Bangor	M
3	Megan	Stiller	San Francisco	M
4	Bustin	Jieber	Austin	F

Чтобы импортировать подмножество листов, можно передать в параметре `sheet_name` список индексов или имен листов. В этом случае `pandas` тоже вернет словарь. Ключи словаря будут соответствовать строкам в списке `sheet_name`. В следующем примере импортируются только листы `Data 1` и `Data 3`:

```
In [33] pd.read_excel(
        "Multiple Worksheets.xlsx",
        sheet_name = ["Data 1", "Data 3"]
    )
```

```
Out [33] {'Data 1':  First Name Last Name      City Gender
0    Brandon    James      Miami    M
1     Sean    Hawkins      Denver    M
2     Judy      Day    Los Angeles    F
3    Ashley    Ruiz  San Francisco    F
4    Stephanie  Gomez    Portland    F,
'Data 3':  First Name Last Name      City Gender
0    Robert   Miller   Seattle     M
1    Tara     Garcia  Phoenix     F
2    Raphael  Rodriguez Orlando    M}
```

А этот пример извлекает листы с индексами 1 и 2, или, то есть опять же второй и третий листы:

```
In [34] pd.read_excel("Multiple Worksheets.xlsx", sheet_name = [1, 2])
```

```
Out [34] {1:   First Name Last Name      City Gender
0   Parker      Power    Raleigh      F
1  Preston  Prescott  Philadelphia      F
2  Ronaldo   Donaldo    Bangor        M
3   Megan    Stiller  San Francisco      M
4   Bustin    Jieber    Austin        F,
2:   First Name Last Name      City Gender
0   Robert    Miller   Seattle        M
1    Tara    Garcia   Phoenix        F
2  Raphael  Rodriguez  Orlando        M}
```

Импортировав листы в наборы данных `DataFrame`, можно вызывать любые методы, присущие таким наборам. Тип источника данных не влияет на доступные нам операции.

12.3.3. Экспорт книг Excel

Вернемся к набору данных `baby_names`, который скачан с сайта правительства Нью-Йорка. Напомню, как выглядят действия и результаты:

```
In [35] baby_names.head()
```

```
Out [35]
```

	Year of Birth	Gender	Ethnicity	Child's First Name	Count	Rank
0	2011	FEMALE	HISPANIC	GERALDINE	13	75
1	2011	FEMALE	HISPANIC	GIA	21	67
2	2011	FEMALE	HISPANIC	GIANNA	49	42
3	2011	FEMALE	HISPANIC	GISELLE	38	51
4	2011	FEMALE	HISPANIC	GRACE	36	53

Допустим, мы решили разделить этот набор данных на две части по половому признаку и записать каждый `DataFrame` в отдельный лист в новой книге Excel. Начнем с фильтрации набора данных `baby_names` по значениям в столбце `Gender`. В главе 5 для этого был предложен такой синтаксис:

```
In [36] girls = baby_names[baby_names["Gender"] == "FEMALE"]
        boys = baby_names[baby_names["Gender"] == "MALE"]
```

Чтобы записать наборы данных в книгу Excel, требуется выполнить больше шагов, чем для записи в CSV. Во-первых, нужно создать объект `ExcelWriter`, который служит основой будущей книги, а затем прикрепить к нему отдельные листы.

Конструктор `ExcelWriter` доступен на верхнем уровне библиотеки `pandas`. В первом параметре `path` он принимает имя файла новой книги в виде строки. Если имя файла указать без пути к нему, то `pandas` создаст файл Excel в том же каталоге, где был открыт блокнот Jupyter. Обязательно сохраните объект `ExcelWriter` в переменной. В примере ниже для этого используется переменная `excel_file`:

```
In [37] excel_file = pd.ExcelWriter("Baby_Names.xlsx")
        excel_file
```

```
Out [37] <pandas.io.excel._openpyxl._OpenpyxlWriter at 0x118a7bf90>
```

На следующем шаге нужно подключить наборы данных `girls` и `boys` к отдельным листам в книге. Начнем работать с первым набором.

Объекты `DataFrame` имеют метод `to_excel` для записи в книгу Excel. В первом параметре `excel_writer` он принимает объект `ExcelWriter`, подобный тому, что мы создали в предыдущем примере. В параметре `sheet_name` метод принимает имя листа в виде строки. Наконец, в параметре `index` можно передать значение `False`, чтобы исключить индекс `DataFrame`:

```
In [38] girls.to_excel(
        excel_writer = excel_file, sheet_name = "Girls", index = False
    )
```

Обратите внимание, что на данный момент мы еще не создали книгу Excel, а просто создали объект `ExcelWriter`, к которому подключили набор данных `girls`, чтобы создать книгу.

Теперь подключим набор данных `boys`. Для этого вызовем метод `to_excel` для `boys`, передав в параметре `excel_writer` тот же объект `ExcelWriter`. Теперь `pandas` знает, что она должна записать оба набора данных в одну и ту же книгу. Изменим также строковый аргумент в параметре `sheet_name`. Чтобы экспортировать только часть столбцов, передадим их список в параметре `columns`. Следующий пример сообщает библиотеке `pandas`, что при записи набора данных `boys` в лист `Boys` она должна включить только столбцы `Child's First Name`, `Count` и `Rank`:

```
In [39] boys.to_excel(
        excel_file,
        sheet_name = "Boys",
        index = False,
        columns = ["Child's First Name", "Count", "Rank"]
    )
```

Теперь, выполнив подготовительные операции, можно записать книгу Excel на диск. Для этого нужно вызвать метод `save` объекта `excel_file`:

```
In [40] excel_file.save()
```

Проверьте интерфейс навигации Jupyter Notebook, чтобы увидеть результат. На рис. 12.5 показан новый файл `Baby_Names.xlsx` в папке с блокнотом.



Рис. 12.5. Файл Excel, сохраненный в том же каталоге, что и блокнот Jupyter

Вот и все. Теперь вы знаете, как с помощью pandas экспортировать данные в файлы JSON, CSV и XLSX. Ну и не забудьте, что библиотека предлагает множество других функций для экспорта своих структур данных в файлы других форматов.

12.4. УПРАЖНЕНИЯ

Опробуем на практике идеи, представленные в этой главе. Файл `tv_shows.json` содержит сводный список с названиями серий из телесериалов, извлеченный из Episodate.com API (<https://www.episodate.com/api>). В данном случае он включает информацию о трех телесериалах: *The X-Files* («Секретные материалы»), *Lost* («Остаться в живых») и *Buffy the Vampire Slayer* («Баффи — истребительница вампиров»).

```
In [41] tv_shows_json = pd.read_json("tv_shows.json")
        tv_shows_json
```

Out [41]

```

                                     shows
-----
0  {'show': 'The X-Files', 'runtime': 60, 'network': 'FOX',...
1  {'show': 'Lost', 'runtime': 60, 'network': 'ABC', 'episo...
2  {'show': 'Buffy the Vampire Slayer', 'runtime': 60, 'net...
```

Данные в формате JSON состоят из ключа `shows` верхнего уровня, которому соответствует список из трех словарей, по одному для каждого из трех сериалов:

```
{
  "shows": [{}, {}, {}]
}
```

Каждый вложенный словарь содержит ключи `"show"`, `"runtime"`, `"network"` и `"episodes"`. Вот как выглядит первый усеченный словарь:

```
In [42] tv_shows_json.loc[0, "shows"]
```

Out [42] {'show': 'The X-Files',

```
'runtime': 60,
'network': 'FOX',
'episodes': [{ 'season': 1,
               'episode': 1,
               'name': 'Pilot',
               'air_date': '1993-09-11 01:00:00'},
              { 'season': 1,
               'episode': 2,
               'name': 'Deep Throat',
               'air_date': '1993-09-18 01:00:00'},
```

Ключ "episodes" хранит список словарей. Каждый словарь содержит информацию об одной серии. В примере выше по тексту выведены в результате данные по первым двум сериям из первого сезона *The X-Files*.

12.4.1. Задачи

Решите следующие задачи.

1. Нормализуйте данные в каждом словаре в столбце "shows". В результате должен получиться `DataFrame`, в котором каждая серия хранится в отдельной записи. Каждая запись должна включать соответствующие метаданные серии (season, episode, name и air_date), а также информацию верхнего уровня о сериале (show, runtime и network).
2. Разбейте нормализованный набор данных на три `DataFrame`, по одному для каждого сериала ("The X-Files", "Lost" и "Buffy the Vampire Slayer").
3. Сохраните три `DataFrame` в книгу Excel `episodes.xlsx`, каждый в своем листе (имена листов выберите по своему вкусу).

12.4.2. Решения

Рассмотрим решения задач.

1. Для извлечения информации о сериях в каждом сериале можно использовать функцию `json_normalize`. Серии доступны по ключу "episodes", и его можно передать в параметре `record_path`. Чтобы извлечь общие данные для сериала в целом, можно в параметре `meta` передать список соответствующих ключей верхнего уровня:

```
In [43] tv_shows = pd.json_normalize(
        data = tv_shows_json["shows"],
        record_path = "episodes",
        meta = ["show", "runtime", "network"]
    )
tv_shows
```

Out [43]

	season	episode	name	air_date	show	runtime	network
0	1	1	Pilot	1993-09-1...	The X-Files	60	FOX
1	1	2	Deep Throat	1993-09-1...	The X-Files	60	FOX
2	1	3	Squeeze	1993-09-2...	The X-Files	60	FOX
3	1	4	Conduit	1993-10-0...	The X-Files	60	FOX
4	1	5	The Jerse...	1993-10-0...	The X-Files	60	FOX
...
477	7	18	Dirty Girls	2003-04-1...	Buffy the...	60	UPN
478	7	19	Empty Places	2003-04-3...	Buffy the...	60	UPN
479	7	20	Touched	2003-05-0...	Buffy the...	60	UPN
480	7	21	End of Days	2003-05-1...	Buffy the...	60	UPN
481	7	22	Chosen	2003-05-2...	Buffy the...	60	UPN

482 rows × 7 columns

2. Очередная задача, напомним, требует разбить весь набор данных на три `DataFrame`, по одному для каждого телесериала. Для ее решения можно отфильтровать строки в `tv_shows` по значениям в столбце `show`:

```
In [44] xfiles = tv_shows[tv_shows["show"] == "The X-Files"]
        lost = tv_shows[tv_shows["show"] == "Lost"]
        buffy = tv_shows[tv_shows["show"] == "Buffy the Vampire Slayer"]
```

3. Наконец, запишем три `DataFrame` в книгу Excel. Для начала создадим экземпляр объекта `ExcelWriter` и сохраним его в переменной. В первом параметре конструктору можно передать имя книги. Я решил назвать ее `episodes.xlsx`:

```
In [45] episodes = pd.ExcelWriter("episodes.xlsx")
        episodes
```

Out [45] <pandas.io.excel._openpyxl._OpenpyxlWriter at 0x11e5cd3d0>

Затем нужно вызвать метод `to_excel` трех `DataFrame`, чтобы связать их с отдельными листами в книге. В каждый вызов мы передадим один и тот же объект `episodes` в параметре `excel_writer`. Кроме того, в каждый вызов нужно передать уникальное имя для каждого листа в параметре `sheet_name`. Наконец, в параметре `index` передадим значение `False`, чтобы исключить сохранение индекса `DataFrame`:

```
In [46] xfiles.to_excel(
        excel_writer = episodes, sheet_name = "X-Files", index = False
    )
```

```
In [47] lost.to_excel(
        excel_writer = episodes, sheet_name = "Lost", index = False
    )
```

```
In [48] buffy.to_excel(
        excel_writer = episodes,
        sheet_name = "Buffy the Vampire Slayer",
        index = False
    )
```

После связывания `DataFrame` с листами можно вызвать метод `save` объекта `episodes`, чтобы создать книгу `episodes.xlsx`:

```
In [49] episodes.save()
```

Поздравляю, упражнения выполнены!

РЕЗЮМЕ

- Функция `read_json` преобразует файл JSON в `DataFrame`.
- Функция `json_normalize` преобразует вложенные данные JSON в табличный `DataFrame`.
- В функции импорта, такие как `read_csv`, `read_json` и `read_excel`, можно вместо ссылки на локальный файл передать URL. Pandas загрузит набор данных, доступный по указанной ссылке.
- Функция `read_excel` осуществляет импорт книги Excel. Параметр `sheet_name` функции задает листы для импорта. Если импортируется несколько листов, pandas сохраняет полученные объекты `DataFrame` в словаре.
- Чтобы записать один или несколько `DataFrame` в книгу Excel, нужно создать экземпляр `ExcelWriter`, связать с ним объекты `DataFrame` вызовом их методов `to_excel`, а затем вызвать метод `save` объекта `ExcelWriter`.

13

Настройка pandas

В этой главе

- ✓ Настройка параметров отображения pandas как для всего блокнота Jupyter, так и для отдельных ячеек.
- ✓ Ограничение количества печатаемых строк и столбцов в DataFrame.
- ✓ Изменение точности представления чисел с десятичной точкой.
- ✓ Усечение текстового содержимого ячейки.
- ✓ Округление числовых значений, если они опускаются ниже нижнего предела.

Работая с наборами данных в этой книге, мы не раз наблюдали, как pandas старается повысить нашу эффективность, предлагая разумные решения, связанные с представлением данных. Например, когда мы выводим набор данных `DataFrame`, содержащий 1000 строк, библиотека предполагает, что для нас предпочтительнее было бы увидеть 30 строк в начале и в конце, а не весь набор данных, который определенно не уместится на экране. Но иногда бывает желательно переопределить настройки pandas по умолчанию и привести их в соответствие со своими потребностями. К счастью, библиотека позволяет изменить многие из своих параметров. В этой главе я расскажу, как установить в соответствии со своими предпочтениями количество отображаемых строк и столбцов, точность вычислений с плавающей точкой и округление значений. Итак, засучим рукава и посмотрим, как можно воздействовать на ситуацию с настройками.

13.1. ПОЛУЧЕНИЕ И ИЗМЕНЕНИЕ ПАРАМЕТРОВ НАСТРОЙКИ PANDAS

Для начала импортируем библиотеку pandas и присвоим ей псевдоним `pd`:

```
In [1] import pandas as pd
```

Набор данных для этой главы, `happiness.csv`, содержит оценки уровня счастья в разных странах мира. Эти данные собраны службой социологических опросов Gallup при поддержке Организации Объединенных Наций. Каждая запись в наборе данных включает совокупный показатель счастья страны и величину валового внутреннего продукта (ВВП), приходящегося на душу населения, величины социальной поддержки и ожидаемой продолжительности жизни и оценку щедрости. Всего набор данных содержит шесть столбцов и 156 строк:

```
In [2] happiness = pd.read_csv("happiness.csv")
      happiness.head()
```

```
Out [2]
```

	Country	Score	GDP per cap...	Social sup...	Life expect...	Generosity
0	Finland	7.769	1.340	1.587	0.986	0.153
1	Denmark	7.600	1.383	1.573	0.996	0.252
2	Norway	7.554	1.488	1.582	1.028	0.271
3	Iceland	7.494	1.380	1.624	1.026	0.354
4	Netherlands	7.488	1.396	1.522	0.999	0.322

Pandas хранит свои настройки в единственном объекте `options`, доступном на верхнем уровне библиотеки. Каждый его параметр принадлежит некоторой категории. Начнем с категории `display`, содержащей настройки отображения структуры данных pandas.

Функция `describe_option` возвращает описание параметра, название которого можно передать функции в виде строки. Рассмотрим параметр `max_rows` в категории `display`. Он определяет максимальное количество строк, которое выведет pandas, прежде чем та усечет отображаемую часть `DataFrame`¹:

¹ При превышении `max_rows` выполняется переключение на отображение в усеченном виде. В зависимости от значения `large_repr` объекты либо усекаются с удалением центральной части, либо выводятся в виде сводного представления.

Значение `None` означает «без ограничений».

Если Python/IPython работает в терминале и параметр `large_repr` имеет значение `truncate`, то параметру `max_rows` можно присвоить значение 0, и тогда pandas автоматически будет определять высоту окна терминала и выводить усеченный объект, соответствующий высоте окна. Блокнот IPython, IPython qtconsole или IDLE имеют графический интерфейс, поэтому правильно определить количество строк для вывода автоматически невозможно.

[по умолчанию: 60] [текущее значение: 60]. — *Примеч. пер.*

```
In [3] pd.describe_option("display.max_rows")
```

```
Out [3]
```

```
display.max_rows : int
    If max_rows is exceeded, switch to truncate view. Depending on
    `large_repr`, objects are either centrally truncated or printed
    as a summary view. 'None' value means unlimited.
    In case python/IPython is running in a terminal and
    `large_repr` equals 'truncate' this can be set to 0 and pandas
    will auto-detect the height of the terminal and print a
    truncated object which fits the screen height. The IPython
    notebook, IPython qtconsole, or IDLE do not run in a terminal
    and hence it is not possible to do correct auto-detection.
    [default: 60] [currently: 60]
```

Обратите внимание, что в конце описания выводится значение параметра по умолчанию и текущее значение.

Pandas выводит описания всех параметров библиотеки, соответствующих строковому аргументу. Библиотека обычно использует регулярные выражения для сравнения аргумента `describe_option` с названиями имеющихся параметров. Напомню, что *регулярное выражение* — это шаблон искомого текста (подробный обзор регулярных выражений вы найдете в приложении Д). В следующем примере функции передается аргумент `"max_col"`. Pandas выводит описание двух параметров, соответствующих указанному аргументу:

```
In [4] pd.describe_option("max_col")
```

```
Out [4]
```

```
display.max_columns : int
    If max_cols is exceeded, switch to truncate view. Depending on
    `large_repr`, objects are either centrally truncated or printed as
    a summary view. 'None' value means unlimited.
    In case python/IPython is running in a terminal and `large_repr`
    equals 'truncate' this can be set to 0 and pandas will auto-detect
    the width of the terminal and print a truncated object which fits
    the screen width. The IPython notebook, IPython qtconsole, or IDLE
    do not run in a terminal and hence it is not possible to do
    correct auto-detection.
    [default: 20] [currently: 5]
display.max_colwidth : int or None
    The maximum width in characters of a column in the repr of
    a pandas data structure. When the column overflows, a "..."
    placeholder is embedded in the output. A 'None' value means unlimited.
    [default: 50] [currently: 9]
```

Несмотря на привлекательность использования регулярных выражений, я все же рекомендую писать полное имя параметра, включая его категорию. Явный код, как правило, менее подвержен ошибкам.

Получить текущее значение параметра можно двумя способами. Первый — это функция `get_option`, доступная на верхнем уровне pandas; как и `describe_option`, она принимает строковый аргумент с именем искомого параметра. Второй способ заключается в обращении к атрибутам объекта `pd.options` с именами, совпадающими с именем категории и параметра.

В следующем примере показаны оба способа. Обе строки кода возвращают 60 для параметра `max_rows`, согласно этому значению pandas будет обрезать вывод любого `DataFrame` длиной более 60 строк:

```
In [5] # Следующие две строки эквивалентны
      pd.get_option("display.max_rows")
      pd.options.display.max_rows
```

```
Out [5] 60
```

Точно так же изменить значение параметра настройки можно двумя способами. Первый способ — функция `set_option`, доступная на верхнем уровне pandas, которая принимает имя параметра в качестве первого аргумента и новое значение в качестве второго. Второй способ — операция присваивания нового значения через атрибуты объекта `pd.options`:

```
In [6] # Следующие две строки эквивалентны
      pd.set_option("display.max_rows", 6)
      pd.options.display.max_rows = 6
```

Здесь мы настроили усечение вывода `DataFrame`, если он содержит больше шести строк:

```
In [7] pd.options.display.max_rows
```

```
Out [7] 6
```

Посмотрим, как отразились проведенные изменения. Следующий пример выводит первые шесть строк из набора данных `happiness`. Установленный нами предел в шесть строк не превышен, поэтому pandas выводит `DataFrame` без усечения:

```
In [8] happiness.head(6)
```

```
Out [8]
```

	Country	Score	GDP per cap...	Social sup...	Life expect...	Generosity
0	Finland	7.769	1.340	1.587	0.986	0.153
1	Denmark	7.600	1.383	1.573	0.996	0.252
2	Norway	7.554	1.488	1.582	1.028	0.271
3	Iceland	7.494	1.380	1.624	1.026	0.354
4	Netherlands	7.488	1.396	1.522	0.999	0.322
5	Switzerland	7.480	1.452	1.526	1.052	0.263

Теперь попробуем превысить пороговое значение и выведем первые семь строк из `happiness`. Библиотека всегда стремится выводить одинаковое количество строк до и после усеечения. В измененном запросе она вывела три первые и три последние строки, отбросив среднюю строку (с индексом 3):

```
In [9] happiness.head(7)
```

```
Out [9]
```

	Country	Score	GDP per cap...	Social sup...	Life expect...	Generosity
0	Finland	7.769	1.340	1.587	0.986	0.153
1	Denmark	7.600	1.383	1.573	0.996	0.252
2	Norway	7.554	1.488	1.582	1.028	0.271
...
4	Netherlands	7.488	1.396	1.522	0.999	0.322
5	Switzerland	7.480	1.452	1.526	1.052	0.263
6	Sweden	7.343	1.387	1.487	1.009	0.267

```
7 rows x 6 columns
```

Параметр `max_rows` задает максимальное количество печатаемых строк. Дополняющий его параметр `display.max_columns` устанавливает максимальное количество печатаемых столбцов. Он имеет значение по умолчанию 20:

```
In [10] # Следующие две строки эквивалентны
pd.get_option("display.max_columns")
pd.options.display.max_columns
```

```
Out [10] 20
```

И снова изменить параметр можно с помощью функции `set_option` или прямым присваиванием значения вложенному атрибуту `max_columns`:

```
In [11] # Следующие две строки эквивалентны
pd.set_option("display.max_columns", 2)
pd.options.display.max_columns = 2
```

Если присвоить параметру `max_columns` четное число, то pandas не будет включать в число подсчета выводимых столбцов столбец с заполнителями. Набор данных `happiness` имеет шесть столбцов, но в следующем примере выводятся, как и требовалось, только два из них. Pandas вывела первый и последний столбцы — `Country` и `Generosity` — и добавила между ними столбец с заполнителями:

```
In [12] happiness.head(7)
```

```
Out [12]
```

	Country	...	Generosity
0	Finland	...	0.153
1	Denmark	...	0.252

```

2      Norway ...    0.271
...      ... ...    ...
4 Netherlands ...    0.322
5 Switzerland ...    0.263
6      Sweden ...    0.267

```

```
7 rows x 6 columns
```

Если присвоить параметру `max_columns` нечетное число, то pandas включит в число подсчета выводимых столбцов столбец с заполнителями. Нечетное число гарантирует, что pandas выведет одинаковое количество столбцов по обе стороны. В следующем примере параметру `max_columns` присваивается значение 5. В качестве результата в выводе набора данных `happiness` отображаются два столбца слева (`Country` и `Score`), столбец с заполнителями и два столбца справа (`Life expectancy` и `Generosity`). Pandas выводит четыре из шести имеющихся столбцов:

```

In [13] # Следующие две строки эквивалентны
pd.set_option("display.max_columns", 5)
pd.options.display.max_columns = 5

```

```
In [14] happiness.head(7)
```

```
Out [14]
```

	Country	Score	...	Life expectancy	Generosity
0	Finland	7.769	...	0.986	0.153
1	Denmark	7.600	...	0.996	0.252
2	Norway	7.554	...	1.028	0.271
...
4	Netherlands	7.488	...	0.999	0.322
5	Switzerland	7.480	...	1.052	0.263
6	Sweden	7.343	...	1.009	0.267

```
5 rows x 6 columns
```

Чтобы вернуть параметр в исходное состояние, просто передайте его имя функции `reset_option`. Следующий пример восстанавливает значение по умолчанию в параметре `max_rows`:

```
In [15] pd.reset_option("display.max_rows")
```

Проверить проведенное изменение можно с помощью все той же функции `get_option`:

```
In [16] pd.get_option("display.max_rows")
```

```
Out [16] 60
```

Как видите, pandas вернула параметру `max_rows` его значение по умолчанию — 60.

13.2. ТОЧНОСТЬ

Теперь, освоившись с интерфейсом изменения настроек в pandas, рассмотрим несколько наиболее популярных настроечных параметров.

Параметр `display.precision` устанавливает количество цифр, отображаемых после десятичной точки при выводе вещественных чисел. Значение по умолчанию равно 6:

```
In [17] pd.describe_option("display.precision")
```

```
Out [17]
```

```
display.precision : int
Floating point output precision (number of significant
digits). This is only a suggestion
[default: 6] [currently: 6]
```

Следующий пример присваивает параметру `precision` значение 2. Этот параметр — точность — влияет на вывод значений во всех четырех столбцах в наборе данных `happiness`, которые содержат числа с плавающей точкой:

```
In [18] # Следующие две строки эквивалентны
pd.set_option("display.precision", 2)
pd.options.display.precision = 2
```

```
In [19] happiness.head()
```

```
Out [19]
```

	Country	Score	...	Life expectancy	Generosity
0	Finland	7.77	...	1.34	0.15
1	Denmark	7.60	...	1.38	0.25
2	Norway	7.55	...	1.49	0.27
3	Iceland	7.49	...	1.38	0.35
4	Netherlands	7.49	...	1.40	0.32

```
5 rows x 6 columns
```

Параметр `precision`, как уже указывалось, влияет только на представление чисел с плавающей точкой. Исходные значения в `DataFrame` остаются в неприкосновенности, что легко доказать, выбрав произвольное значение с плавающей точкой с помощью метода `loc`, например, из столбца `Score`:

```
In [20] happiness.loc[0, "Score"]
```

```
Out [20] 7.769
```

Исходное значение в столбце `Score`, равное 7.769, никак не изменилось. Pandas изменила лишь его представление, округлив до 7.77 при выводе `DataFrame`.

13.3. МАКСИМАЛЬНАЯ ШИРИНА СТОЛБЦА

Параметр `display.max_colwidth` определяет максимальное количество символов, которые pandas может вывести, прежде чем обрезать вывод содержимого текстовой ячейки:

```
In [21] pd.describe_option("display.max_colwidth")
```

```
Out [21]
```

```
display.max_colwidth : int or None
    The maximum width in characters of a column in the repr of
    a pandas data structure. When the column overflows, a "..."
    placeholder is embedded in the output. A 'None' value means
    unlimited.
    [default: 50] [currently: 50]
```

Следующий пример настраивает усечение текста, если его длина превышает девять символов:

```
In [22] # Следующие две строки эквивалентны
pd.set_option("display.max_colwidth", 9)
pd.options.display.max_colwidth = 9
```

Посмотрим, как теперь изменится вывод набора данных `happiness`:

```
In [23] happiness.tail()
```

```
Out [23]
```

	Country	Score	...	Life expectancy	Generosity
151	Rwanda	3.33	...	0.61	0.22
152	Tanzania	3.23	...	0.50	0.28
153	Afgha...	3.20	...	0.36	0.16
154	Central Afr...	3.08	...	0.10	0.23
155	South...	2.85	...	0.29	0.20

```
5 rows x 6 columns
```

Pandas обрезала последние три значения в столбце `Country` (Afghanistan, Central African Republic и South Sudan). Первые два значения — Rwanda (шесть символов) и Tanzania (восемь символов) — не изменились при выводе.

13.4. ПОРОГ ОКРУГЛЕНИЯ ДО НУЛЯ

В некоторых случаях значения сущностей в наборах можно смело считать незначительными, если они достаточно близки к 0. Скажем, имеет место ситуация, когда для вас значение 0.10 может рассматриваться как «равное 0» или

«фактически 0». Параметр `display.chop_threshold` устанавливает предел, ниже которого числа с плавающей точкой будут отображаться как 0:

```
In [24] pd.describe_option("display.chop_threshold")
```

```
Out [24]
```

```
display.chop_threshold : float or None
    if set to a float value, all float values smaller then the
    given threshold will be displayed as exactly 0 by repr and
    friends.
    [default: None] [currently: None]
```

Следующий пример устанавливает порог округления до нуля равным 0.25:

```
In [25] pd.set_option("display.chop_threshold", 0.25)
```

Обратите внимание, что в примере ниже в строке с индексом 154 и в столбцах `Life expectancy` и `Generosity` pandas вывела значение 0.00 (округлив фактические значения 0.105 и 0.235 соответственно):

```
In [26] happiness.tail()
```

```
Out [26]
```

	Country	Score	...	Life expectancy	Generosity
151	Rwanda	3.33	...	0.61	0.00
152	Tanzania	3.23	...	0.50	0.28
153	Afghanistan	3.20	...	0.36	0.00
154	Central Afr...	3.08	...	0.00	0.00
155	South Sudan	2.85	...	0.29	0.00

```
5 rows x 6 columns
```

Подобно параметру `precision`, `chop_threshold` не изменяет исходные значения в `DataFrame` и влияет только на их представление при выводе.

13.5. ПАРАМЕТРЫ КОНТЕКСТА

До сих пор мы изменяли глобальные настройки. При их изменении меняется формат вывода во всех последующих ячейках блокнота Jupyter. Глобальные настройки продолжают действовать, пока им не будут присвоены новые значения. Например, если присвоить параметру `display.max_columns` значение 6, то Jupyter будет выводить не более шести столбцов из `DataFrame` во всех последующих ячейках.

Но иногда бывает желательно настроить параметры отображения только для одной ячейки. Этого можно добиться с помощью функции `option_context`,

доступной на верхнем уровне pandas, объединив ее со встроенным ключевым словом `with` для создания *контекстного блока*. Контекстный блок можно рассматривать как временную среду выполнения. Функция `option_context` устанавливает временные значения для параметров pandas, действующие, пока выполняется код внутри блока; глобальные настройки при этом не затрагиваются.

Настройки передаются функции `option_context` в виде последовательности аргументов. Следующий пример выводит набор данных `happiness` с параметрами:

- `display.max_columns`, равным 5;
- `display.max_rows`, равным 10;
- `display.precision`, равным 3.

Jupyter не распознает содержимое блока `with` как последний оператор ячейки блокнота, поэтому нужно вызвать функцию блокнота с именем `display`, чтобы вручную отобразить содержимое `DataFrame`:

```
In [27] with pd.option_context(
        "display.max_columns", 5,
        "display.max_rows", 10,
        "display.precision", 3
    ):
        display(happiness)
```

Out [27]

	Country	Score	...	Life expectancy	Generosity
0	Finland	7.769	...	0.986	0.153
1	Denmark	7.600	...	0.996	0.252
2	Norway	7.554	...	1.028	0.271
3	Iceland	7.494	...	1.026	0.354
4	Netherlands	7.488	...	0.999	0.322
...
151	Rwanda	3.334	...	0.614	0.217
152	Tanzania	3.231	...	0.499	0.276
153	Afghanistan	3.203	...	0.361	0.158
154	Central Afr...	3.083	...	0.105	0.235
155	South Sudan	2.853	...	0.295	0.202

156 rows × 6 columns

Благодаря использованию ключевого слова `with` мы не изменили глобальные настройки этих трех параметров, они сохранили свои первоначальные значения.

Функцию `option_context` удобно использовать для настройки параметров по-разному в разных ячейках. Если необходимо обеспечить единообразие в отображении всех выходных данных, я все-таки рекомендую установить параметры глобально один раз в верхней ячейке вашего блокнота Jupyter.

РЕЗЮМЕ

- Функция `describe_option` возвращает описание указанных параметров настройки pandas.
- Функция `set_option` устанавливает новое значение в параметре.
- Изменить значение параметра настройки можно также присваиванием значений атрибутам объекта `pd.options`.
- Функция `reset_option` присваивает указанному параметру настройки pandas его значение по умолчанию.
- Параметры `display.max_rows` и `display.max_columns` определяют максимальное количество строк/столбцов при выводе наборов данных `DataFrame`.
- Параметр `display.precision` изменяет количество десятичных знаков, отображаемых после запятой.
- Параметр `display.max_colwidth` определяет максимальную ширину столбцов в символах при выводе.
- Параметр `display.chop_threshold` устанавливает предел, ниже которого числа с плавающей точкой будут отображаться как нули.
- Объединение функции `option_context` с ключевым словом `with` позволяет определить настройки, действующие только в пределах контекстного блока.

14

Визуализация

В этой главе

- ✓ Установка библиотеки Matplotlib для визуализации данных.
- ✓ Визуализация графиков и диаграмм с помощью pandas и Matplotlib.
- ✓ Применение цветовых шаблонов.

Вывод содержимого наборов данных `DataFrame` имеет определенную ценность, но во многих случаях информация лучше воспринимается, когда она представлена в виде графиков и диаграмм. Линейный график позволяет быстро оценить развитие тенденций с течением времени; гистограмма дает возможность четко идентифицировать уникальные категории и численность; круговая диаграмма помогает увидеть пропорции и соотношения и т. д. К счастью, pandas легко интегрируется со многими популярными библиотеками визуализации данных для Python, включая Matplotlib, seaborn и ggplot. В этой главе вы узнаете, как использовать Matplotlib для отображения динамических диаграмм на основе последовательностей `Series` и наборов данных `DataFrame`. Я надеюсь, что эти знания помогут вам добавить изюминку в ваши презентации.

14.1. УСТАНОВКА MATPLOTLIB

По умолчанию для отображения диаграмм и графиков pandas использует пакет Matplotlib, распространяемый с открытым исходным кодом. Установим его в нашу среду Anaconda.

Начнем с запуска приложения Terminal (macOS) или Anaconda Prompt (Windows). Слева в скобках должна быть указана базовая среда Anaconda base — текущая активная среда.

В процессе установки Anaconda (описан в приложении А) мы создали среду с именем `pandas_in_action`. Выполните команду `conda activate`, чтобы активировать ее. Если вы выбрали для среды другое название, замените `pandas_in_action` в команде (ниже по тексту) этим названием:

```
(base) ~$ conda activate pandas_in_action
```

В скобках должно отобразиться имя активированной среды. Выполните команду `conda install matplotlib`, чтобы установить библиотеку Matplotlib в среде `pandas_in_action`:

```
(pandas_in_action) ~$ conda install matplotlib
```

Когда появится запрос на подтверждение, введите Y, чтобы ответить утвердительно, и нажмите клавишу Enter. Когда установка завершится, запустите Jupyter Notebook и создайте новый блокнот.

14.2. ЛИНЕЙНЫЕ ГРАФИКИ

Как всегда, начнем с импорта библиотеки pandas. Также импортируем пакет `pyplot` из библиотеки Matplotlib. В данном случае под *пакетом* понимается вложенная папка в каталоге библиотеки. Сослаться на пакет `pyplot` можно с помощью точечного синтаксиса, который мы используем для ссылки на любые атрибуты библиотек. В сообществе pandas пакет `pyplot` обычно импортируется под псевдонимом `plt`.

По умолчанию каждая диаграмма Matplotlib отображается в Jupyter Notebook в отдельном окне браузера, подобно всплывающему окну на сайте. Эта особенность может раздражать, особенно когда блокнот выводит много графиков. Чтобы заставить Jupyter отображать графики непосредственно под кодом, в следующей ячейке, можно добавить дополнительную строку `%matplotlib inline.%matplotlib inline` — это пример «магической» функции, краткого способа установки конфигурационного параметра в блокноте:

```
In [1] import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```

А теперь приступим к работе с данными! Набор данных для этой главы, `space_missions.csv`, включает сведения о более чем 100 космических полетах, совершенных в течение 2019 и 2020 годов. Каждая запись включает дату полета,

название компании-спонсора, страну, стоимость и признак успешности полета ("Success" или "Failure"):

```
In [2] pd.read_csv("space_missions.csv").head()
```

```
Out [2]
```

	Date	Company Name	Location	Cost	Status
0	2/5/19	Arianespace	France	200.00	Success
1	2/22/19	SpaceX	USA	50.00	Success
2	3/2/19	SpaceX	USA	50.00	Success
3	3/9/19	CASC	China	29.15	Success
4	3/22/19	Arianespace	France	37.00	Success

Настроим два параметра, прежде чем сохранить импортированный `DataFrame` в переменной `space`. Для этого используем параметр `parse_dates`, чтобы импортировать значения в столбце `Date` в виде даты и времени. Затем назначим столбец `Date` на роль индекса `DataFrame`:

```
In [3] space = pd.read_csv(
        "space_missions.csv",
        parse_dates = ["Date"],
        index_col = "Date"
    )

    space.head()
```

```
Out [3]
```

Date	Company Name	Location	Cost	Status
2019-02-05	Arianespace	France	200.00	Success
2019-02-22	SpaceX	USA	50.00	Success
2019-03-02	SpaceX	USA	50.00	Success
2019-03-09	CASC	China	29.15	Success
2019-03-22	Arianespace	France	37.00	Success

Предположим, мы решили графически отобразить стоимость полетов за два года. Оптимальным средством визуализации изменения тенденций с течением времени является график *временной последовательности*. Для его построения надо отложить время по оси *X* и оцениваемые значения по оси *Y*. Приступим: извлечем столбец `Cost` из набора данных `space`. В результате получится последовательность `Series` с числовыми значениями и с датами и временем в индексе:

```
In [4] space["Cost"].head()
```

```
Out [4] Date
        2019-02-05    200.00
        2019-02-22     50.00
```

```

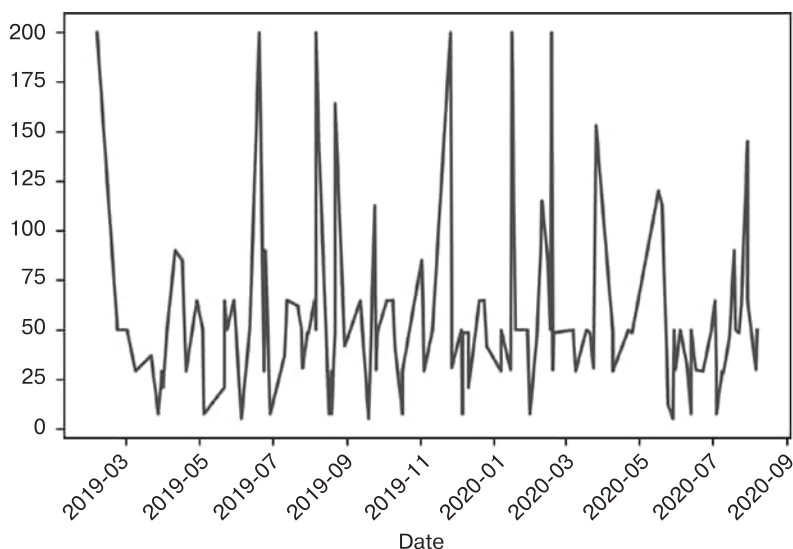
2019-03-02    50.00
2019-03-09    29.15
2019-03-22    37.00
Name: Cost, dtype: float64

```

Чтобы отобразить эти данные в форме графика, вызовем метод `plot` структуры данных. По умолчанию Matplotlib рисует линейный график. Jupyter также выводит местоположение объекта графика в памяти компьютера. Местоположение будет меняться при каждом выполнении ячейки, поэтому его можно смело игнорировать:

```
In [5] space["Cost"].plot()
```

```
Out [5] <matplotlib.axes._subplots.AxesSubplot at 0x11e1c4650>322
```

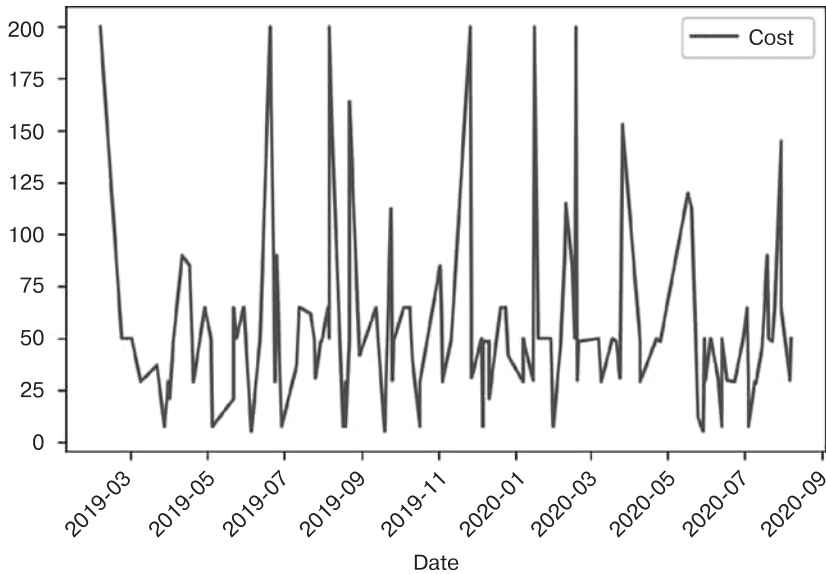


Прекрасно! Мы вывели линейный график с помощью Matplotlib, используя значения из pandas. По умолчанию вдоль оси *X* библиотека отображает индексные метки (в данном случае дату и время), а вдоль оси *Y* — значения в *Series*. Matplotlib сама вычисляет разумные интервалы, исходя из диапазона значений по обеим осям.

Можно было бы вызвать метод `plot` самого набора данных `space`. И в этом случае pandas нарисует тот же график, что и было сделано, как видно из следующего рисунка. Но полной тождественности графиков — первого и второго — мы добились только потому, что набор данных имеет лишь один числовой столбец:

```
In [6] space.plot()
```

```
Out [6] <matplotlib.axes._subplots.AxesSubplot at 0x11ea18790>
```



Если в `DataFrame` окажется несколько числовых столбцов, то Matplotlib нарисует на графике отдельную кривую для каждого из них. Будьте осторожны: если между столбцами существует большой разрыв в величине значений (например, если один числовой столбец имеет значения, исчисляемые миллионами, а другой — сотнями), то изменения в маленьких значениях могут стать незаметными на фоне больших значений. Рассмотрим следующий `DataFrame`:

```
In [7] data = [
        [2000, 3000000],
        [5000, 5000000]
      ]

      df = pd.DataFrame(data = data, columns = ["Small", "Large"])
      df
```

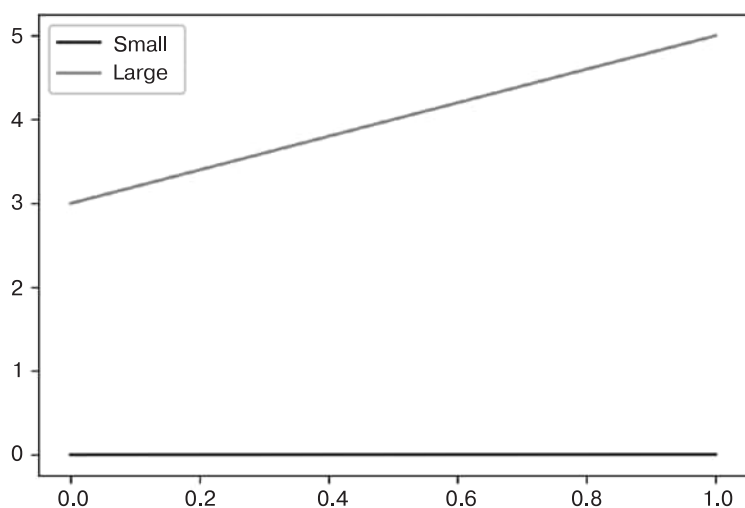
Out [7]

	Small	Large
0	2000	3000000
1	5000	5000000

Если нарисовать линейную диаграмму набора данных `df`, Matplotlib настроит масштаб графика так, что он будет охватывать значения в столбце `Large`. По этой причине тенденция в значениях столбца `Small` станет просто неразличимой:

```
In [8] df.plot()
```

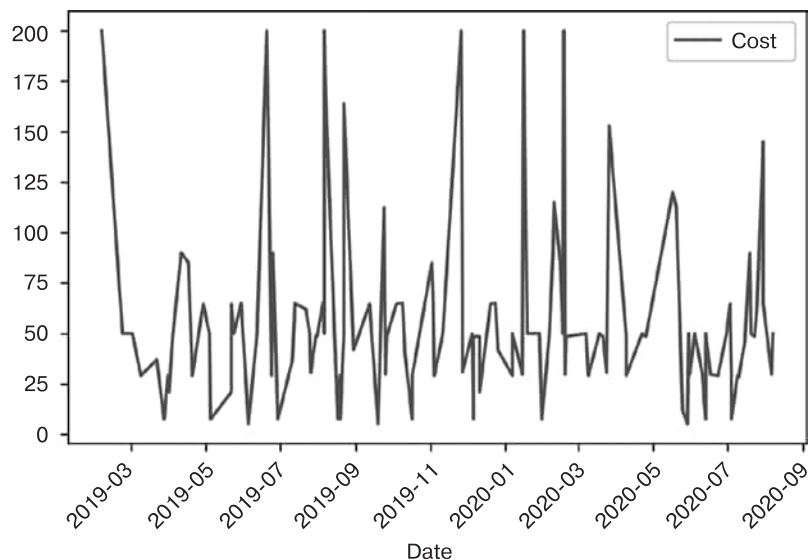
Out [8] <matplotlib.axes._subplots.AxesSubplot at 0x7fc48279b6d0>



Вернемся к нашему набору данных `space`. Метод `plot` принимает параметр `y`, в котором можно передать определение столбца `DataFrame`, значения которого Matplotlib должна отобразить. В следующем примере в параметре `y` передается столбец `Cost`, и это еще один — третий — способ нарисовать тот же график временной последовательности:

```
In [9] space.plot(y = "Cost")
```

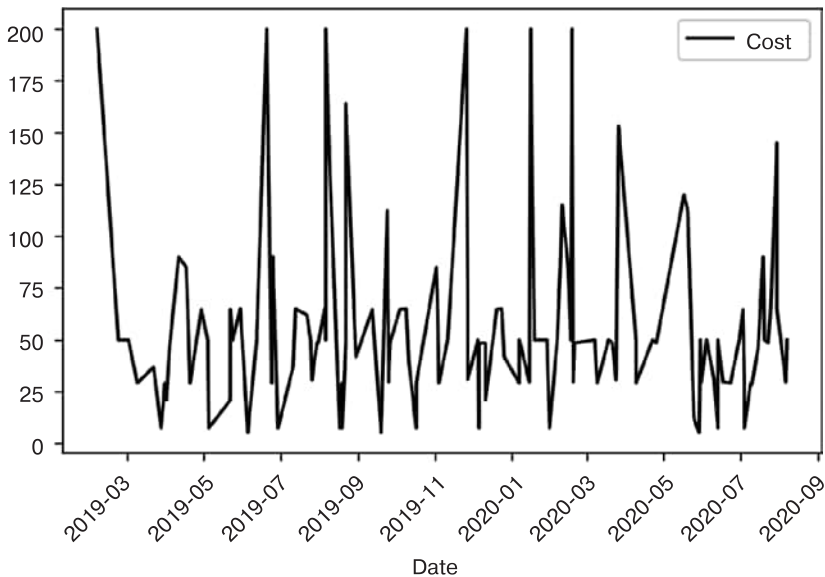
```
Out [9] <matplotlib.axes._subplots.AxesSubplot at 0x11eb0b990>
```



Мы можем использовать параметр `colormap`, чтобы изменить цветопередачу на графике. Считайте это способом настройки цветовой темы при визуализации. Параметр принимает строку с названием цветовой палитры, как она определена в библиотеке `Matplotlib`. Приведу пример использования темы `"gray"`, которая отображает линейную диаграмму в черно-белых тонах:

```
In [10] space.plot(y = "Cost", colormap = "gray")
```

```
Out [10] <matplotlib.axes._subplots.AxesSubplot at 0x11ebef350>
```



Получить список допустимых названий цветовых схем для передачи в параметре `colormap` можно вызовом метода `colormaps` из пакета `pyplot` (в нашем блокноте импортирован с псевдонимом `plt`). Обратите внимание, что некоторые из этих тем можно применять только при соблюдении определенных критериев, таких, например, как минимальное количество линий на графике:

```
In [11] print(plt.colormaps())
```

```
Out [11] ['Accent', 'Accent_r', 'Blues', 'Blues_r', 'BrBG', 'BrBG_r',
          'BuGn', 'BuGn_r', 'BuPu', 'BuPu_r', 'CMRmap', 'CMRmap_r',
          'Dark2', 'Dark2_r', 'GnBu', 'GnBu_r', 'Greens', 'Greens_r',
          'Greys', 'Greys_r', 'OrRd', 'OrRd_r', 'Oranges', 'Oranges_r',
          'PRGn', 'PRGn_r', 'Paired', 'Paired_r', 'Pastel1', 'Pastel1_r',
          'Pastel2', 'Pastel2_r', 'PiYG', 'PiYG_r', 'PuBu', 'PuBuGn',
          'PuBuGn_r', 'PuBu_r', 'PuOr', 'PuOr_r', 'PuRd', 'PuRd_r',
          'Purples', 'Purples_r', 'RdBu', 'RdBu_r', 'RdGy', 'RdGy_r',
```

```
'RdPu', 'RdPu_r', 'RdYlBu', 'RdYlBu_r', 'RdYlGn', 'RdYlGn_r',
'Reds', 'Reds_r', 'Set1', 'Set1_r', 'Set2', 'Set2_r', 'Set3',
'Set3_r', 'Spectral', 'Spectral_r', 'Wistia', 'Wistia_r', 'YlGn',
'YlGnBu', 'YlGnBu_r', 'YlGn_r', 'YlOrBr', 'YlOrBr_r', 'YlOrRd',
'YlOrRd_r', 'afmhot', 'afmhot_r', 'autumn', 'autumn_r', 'binary',
'binary_r', 'bone', 'bone_r', 'brg', 'brg_r', 'bwr', 'bwr_r',
'cividis', 'cividis_r', 'cool', 'cool_r', 'coolwarm',
'coolwarm_r', 'copper', 'copper_r', 'cubehelix', 'cubehelix_r',
'flag', 'flag_r', 'gist_earth', 'gist_earth_r', 'gist_gray',
'gist_gray_r', 'gist_heat', 'gist_heat_r', 'gist_ncar',
'gist_ncar_r', 'gist_rainbow', 'gist_rainbow_r', 'gist_stern',
'gist_stern_r', 'gist_yarg', 'gist_yarg_r', 'gnuplot',
'gnuplot2', 'gnuplot2_r', 'gnuplot_r', 'gray', 'gray_r', 'hot',
'hot_r', 'hsv', 'hsv_r', 'inferno', 'inferno_r', 'jet', 'jet_r',
'magma', 'magma_r', 'nipy_spectral', 'nipy_spectral_r', 'ocean',
'ocean_r', 'pink', 'pink_r', 'plasma', 'plasma_r', 'prism',
'prism_r', 'rainbow', 'rainbow_r', 'seismic', 'seismic_r',
'spring', 'spring_r', 'summer', 'summer_r', 'tab10', 'tab10_r',
'tab20', 'tab20_r', 'tab20b', 'tab20b_r', 'tab20c', 'tab20c_r',
'terrain', 'terrain_r', 'twilight', 'twilight_r',
'twilight_shifted', 'twilight_shifted_r', 'viridis', 'viridis_r',
'winter', 'winter_r']
```

Matplotlib поддерживает более 150 цветовых тем, кроме того, предлагает способы настройки цвета на графиках вручную.

14.3. ГИСТОГРАММЫ

Параметр `kind` метода `plot` изменяет тип диаграммы. Гистограмма — отличный выбор для отображения количества уникальных значений в наборе данных, поэтому воспользуемся ею и посмотрим, сколько космических полетов спонсировала каждая компания в наборе.

Для начала выберем столбец `Company Name` и вызовем метод `value_counts`, чтобы получить последовательность `Series` с количеством полетов, спонсированных каждой компанией:

```
In [12] space["Company Name"].value_counts()
```

```
Out [12] CASC      35
         SpaceX    25
         Roscosmos  12
         Arianespace 10
         Rocket Lab   9
         VKS RF       6
         ULA          6
```



```

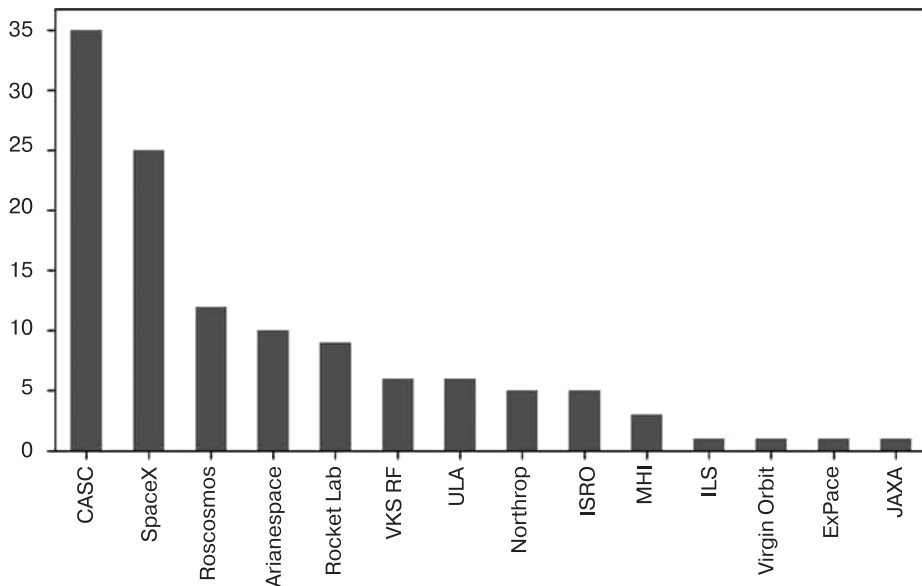
Northrop      5
ISRO          5
MHI           3
Virgin Orbit  1
JAXA          1
ILS           1
ExPace        1
Name: Company Name, dtype: int64

```

Затем вызовем метод `plot` для `Series`, передав аргумент `"bar"` в параметр `kind`. Matplotlib снова отобразит индексные метки вдоль оси *X* и значения вдоль оси *Y*. Похоже, что компания CASC спонсировала больше всего полетов, за ней следует SpaceX:

```
In [13] space["Company Name"].value_counts().plot(kind = "bar")
```

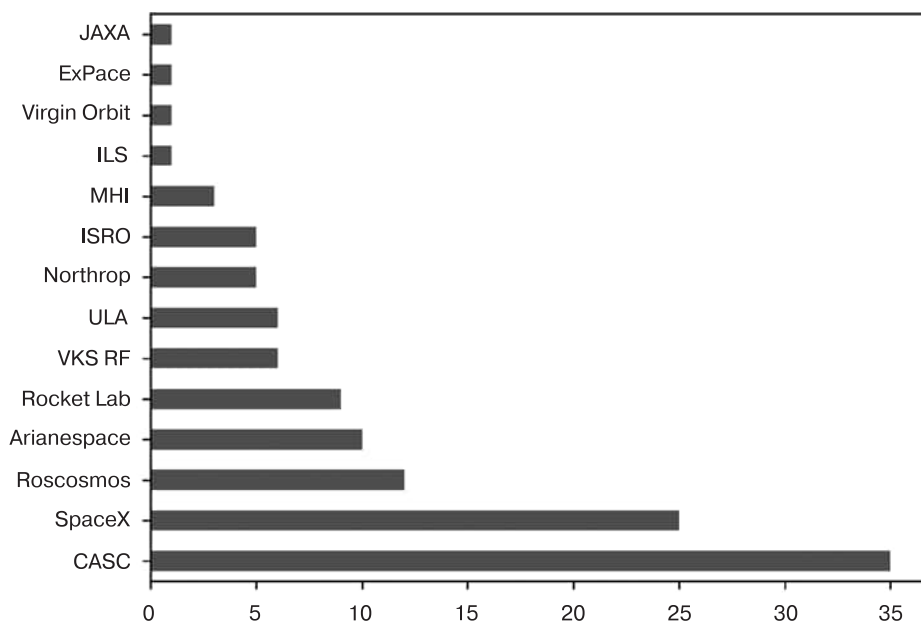
```
Out [13] <matplotlib.axes._subplots.AxesSubplot at 0x11ecd6310>
```



В общем и целом диаграмма получилась неплохая, но приходится поворачивать голову влево-вправо, чтобы прочитать метки и оценить картину в целом. А давайте попробуем изменить аргумент в параметре `kind` на `"barh"`, чтобы отобразить горизонтальную гистограмму:

```
In [14] space["Company Name"].value_counts().plot(kind = "barh")
```

```
Out [14] <matplotlib.axes._subplots.AxesSubplot at 0x11edf0190>
```



Вот так намного лучше! Теперь можно легко определить, какие компании спонсировали наибольшее количество космических полетов.

14.4. КРУГОВЫЕ ДИАГРАММЫ

Круговая диаграмма отображает цветные сегменты, складывающиеся в целый круг (подобно кусочкам пиццы). Каждый сегмент визуально отражает долю, вносимую им в общую сумму.

Используем круговую диаграмму, чтобы сравнить соотношение успешных и неудачных полетов. Столбец `Status` имеет только два уникальных значения: `"Success"` и `"Failure"`. Для начала вызовем метод `value_counts`, чтобы подсчитать количество вхождений каждого из этих значений:

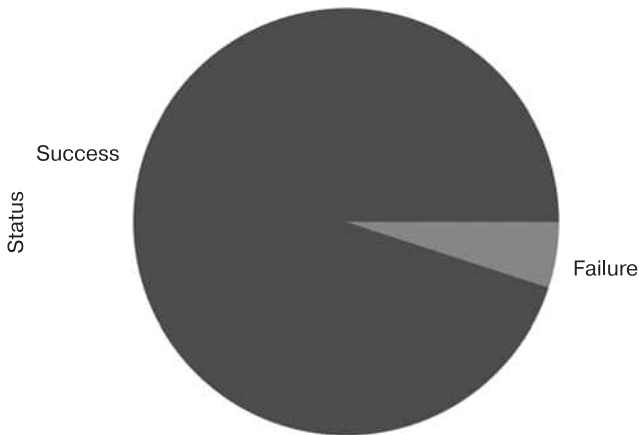
```
In [15] space["Status"].value_counts()
```

```
Out [15] Success    114
         Failure     6
         Name: Status, dtype: int64
```

А затем снова вызовем метод `plot`. Но на этот раз передадим в параметре `kind` аргумент `"pie"`:

```
In [16] space["Status"].value_counts().plot(kind = "pie")
```

```
Out [16] <matplotlib.axes._subplots.AxesSubplot at 0x11ef9ea90>
```

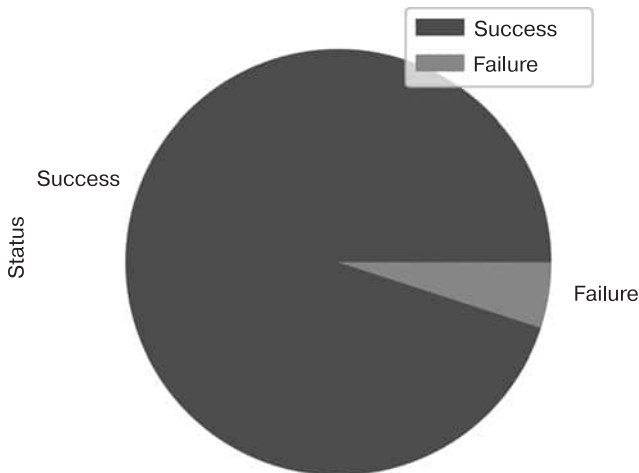


Хорошие новости! Похоже, что подавляющее большинство космических полетов было проведено успешно.

Чтобы добавить легенду в диаграмму, можем передать параметр `legend` с аргументом `True`:

```
In [17] space["Status"].value_counts().plot(kind = "pie", legend = True)
```

```
Out [17] <matplotlib.axes._subplots.AxesSubplot at 0x11eac1a10>
```



Matplotlib поддерживает множество самых разных типов диаграмм и графиков, включая гистограммы, диаграммы рассеяния и коробчатые диаграммы. В ней есть возможность использовать дополнительные параметры для настройки внешнего вида, меток, легенд и интерактивности диаграмм. Здесь мы рассмотрели лишь очень малую часть возможностей этой мощной библиотеки.

РЕЗЮМЕ

- Pandas легко интегрируется с библиотекой Matplotlib для визуализации данных. Она также хорошо сочетается с другими библиотеками построения графиков в экосистеме Python, которые широко используются для обработки и анализа информации.
- Метод `plot` объекта `Series` или `DataFrame` создает диаграмму на основе данных, хранящихся в этой структуре.
- По умолчанию Matplotlib рисует линейный график.
- Параметр `kind` метода `plot` позволяет выбрать тип отображаемой диаграммы, например линейный график, гистограмму или круговую диаграмму.
- Параметр `colormap` изменяет цветовую схему диаграммы. Matplotlib определяет десятки цветовых схем и также дает пользователям возможность настраивать параметры метода `colormaps` вручную.

Приложения

Приложение А

Установка и настройка

Добро пожаловать в раздел книги с дополнительными материалами! В этом приложении вы узнаете, как установить язык программирования Python и библиотеку pandas в операционных системах macOS и Windows. *Библиотека* (иногда библиотеки называют *пакетами*) — это набор инструментов, расширяющих функциональные возможности основного языка программирования. То есть этот пакет расширений или дополнений предлагает средства решения типичных задач, с которыми разработчики сталкиваются при работе с языком. Экосистема Python включает тысячи пакетов для таких областей, как статистика, HTTP-запросы и управление базами данных.

Зависимость — это часть программного обеспечения, которую нужно установить для запуска другой части. Pandas — это не автономный пакет; он имеет набор зависимостей, включающий библиотеки NumPy и pytz. Эти библиотеки могут потребовать установки своих собственных зависимостей. Нам не нужно понимать, что делают все эти пакеты, но следует установить их, чтобы получить возможность использовать pandas.

А.1. ДИСТРИБУТИВ ANACONDA

Библиотеки с открытым исходным кодом часто разрабатываются независимыми коллективами с разной скоростью. К сожалению, независимость циклов разработки может приводить к проблемам совместимости между версиями библиотек. Например, установка последней версии библиотеки pandas без обновления ее зависимостей может сделать ее неработоспособной.

Чтобы упростить установку и управление библиотекой `pandas` и ее зависимостями, применим дистрибутив Python под названием `Anaconda`. *Дистрибутив* — это набор программного обеспечения, объединяющий несколько приложений и их зависимости в одном простом пакете установки. `Anaconda` насчитывает более 20 миллионов пользователей и является самым популярным дистрибутивом в среде начинающих заниматься обработкой данных на Python.

`Anaconda` устанавливает Python и `conda` — мощную систему управления средой. *Среда* — это независимое изолированное окружение для выполнения кода, своего рода игровая площадка, где можно установить Python и набор пакетов. Чтобы поэкспериментировать с другой версией Python или `pandas`, с другой комбинацией пакетов или чем-то еще, можно создать новую среду `conda`. На рис. А.1 показаны три гипотетические среды `conda`, в каждой из которых используется своя версия Python.

Среда 1	Среда 2	Среда 3
Python 2.7	Python 3.9	Python 3.8
pandas 0.20.3	pandas 1.2.0	django 3.0.7
numpy 1.9.1	numpy 1.16.6	flask 1.1.12

Рис. А.1. Три среды `Anaconda` с разными версиями Python и наборами пакетов

Преимущество использования отдельных окружений — их изолированность. Изменения в одной среде не влияют на другую, так как `conda` хранит их в разных папках. Поэтому можно одновременно работать над несколькими проектами, каждый из которых требует своей конфигурации. При установке среды и пакетов `conda` также устанавливает соответствующие зависимости и гарантирует совместимость между различными версиями библиотек. Проще говоря, `conda` — это эффективный способ установить несколько версий и конфигураций инструментов для Python.

Это было лишь общее введение. А теперь приступим к делу и установим `Anaconda`. Откройте в браузере страницу www.anaconda.com/products/individual и найдите раздел загрузки мастера установки для вашей операционной системы. Там, скорее всего, вы увидите несколько версий дистрибутива `Anaconda`.

- Если есть возможность выбора между мастером установки с графическим интерфейсом и с интерфейсом командной строки, выбирайте мастер с графическим интерфейсом.

- Если есть возможность выбора версии Python, выбирайте самую последнюю версию. Как это принято в мире программного обеспечения, больший номер соответствует более новой версии. Python 3 новее Python 2, а Python 3.9 новее Python 3.8. Приступая к изучению новой технологии, всегда лучше начать с последней версии. Не волнуйтесь: если понадобится, `conda` позволит вам создать среду с более ранней версией Python.
- Пользователям Windows может быть предоставлена возможность установки 64- и 32-разрядных версий. Выбор между ними мы обсудим в разделе А.3.

В настоящее время процессы установки в операционных системах macOS и Windows различаются между собой, поэтому найдите раздел, соответствующий вашей ОС, и продолжайте чтение.

А.2. ПРОЦЕСС УСТАНОВКИ В MACOS

Рассмотрим установку Anaconda на компьютер с macOS.

А.2.1. Установка Anaconda в macOS

Загружаемый дистрибутив Anaconda состоит из одного файла `.pkg`. Имя файла, скорее всего, будет включать номер версии Anaconda и название операционной системы (например, `Anaconda3-2021.05-MacOSX-x86_64`). Найдите файл дистрибутива в своей файловой системе и дважды щелкните на нем кнопкой мыши, чтобы запустить установку.

В открывшемся окне нажмите кнопку `Continue` (Продолжить). В следующем диалоге `README` программа установки выведет краткий обзор Anaconda (рис. А.2).

В ходе установки будет создана начальная среда `conda` с именем `base`, включающая более 250 предварительно выбранных пакетов для анализа данных. Позже вы сможете создать дополнительные среды. Мастер установки также сообщает, что эта среда `base` будет активироваться при каждом запуске командной оболочки (мы обсудим этот процесс в подразделе А.2.2). Пока просто примите на веру, что это обязательная часть процесса установки, и продолжайте.

Продолжите установку, переходя последовательно все стадии, предложенные мастером. Примите лицензионное соглашение и требования к пространству. Вам будет предоставлена возможность выбрать каталог установки, как поступить — решать только вам. Обратите внимание, что дистрибутив является автономным; Anaconda устанавливается в единственный каталог. Соответственно, если вы когда-нибудь захотите удалить Anaconda, то можете просто удалить этот каталог.

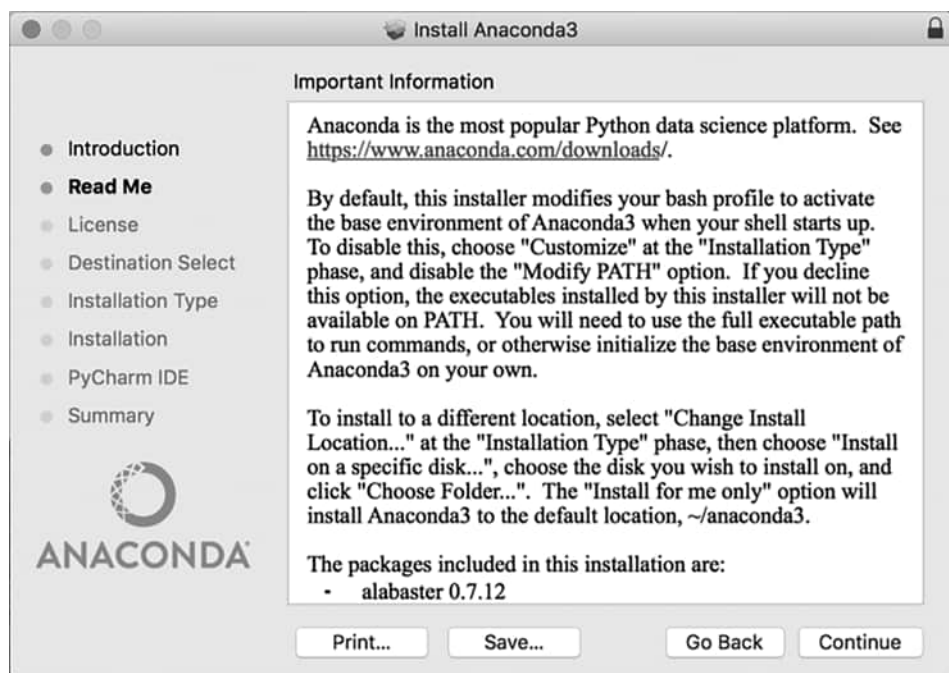


Рис. А.2. Окно мастера установки Anaconda в системе macOS

Установка может занять до нескольких минут. Когда она завершится, щелкните на кнопке **Next** (Далее) в появившейся экранной форме, чтобы выйти из программы мастера установки.

А.2.2. Запуск терминала

В состав Анаconda входит программа с графическим интерфейсом **Navigator**, которая упрощает создание окружений **conda** и управление ими. Однако, прежде чем обратиться к ней, рассмотрим более традиционное приложение **Terminal** для отправки команд диспетчеру окружений **conda**.

Terminal (Терминал) — это приложение для выполнения команд в операционной системе macOS. До появления современных графических интерфейсов пользователи взаимодействовали с компьютерами исключительно с использованием текстовых приложений. В терминал вы вводите команду, а затем нажимаете клавишу **Enter**, чтобы выполнить ее. Я бы хотел, чтобы мы освоили терминал раньше, чем **Anaconda Navigator**, потому что важно понимать детали, которые скрываются за абстракцией графического интерфейса, прежде чем полагаться на него.

Откройте окно Finder и перейдите в каталог Applications (Программы), где вы найдете приложение Terminal в папке Utilities (Утилиты). Запустите приложение. Я также рекомендую перетащить значок приложения Terminal на панель Dock для быстрого доступа.

Терминал должен показать активную среду conda в круглых скобках перед приглашением к вводу с мигающим курсором. Напоминаю, что начальная среда base была создана во время установки Anaconda. На рис. А.3 для примера показано окно программы Terminal с активированной средой base.



Рис. А.3. Окно программы Terminal на компьютере с macOS. base — это активная среда conda

Anaconda активирует диспетчер окружений conda и эту среду base всякий раз, когда запускается программа Terminal.

А.2.3. Типичные команды, доступные в терминале

Для эффективной работы в терминале достаточно запомнить всего несколько команд. В терминале можно перемещаться по каталогам файловой системы компьютера так же, как в Finder. Команда `pwd` (print working directory — «напечатать рабочий каталог») выводит имя папки, в которой мы находимся в настоящий момент:

```
(base) ~$ pwd
/Users/boris
```

Команда `ls` (list — «перечислить») выводит список файлов и папок в текущем рабочем каталоге:

```
(base) ~$ ls
Applications  Documents      Google Drive  Movies        Pictures      anaconda3
Desktop        Downloads      Library       Music         Public
```

Некоторые команды принимают при активации дополнительные флаги. *Флаг* — это конфигурационный параметр, добавляемый в командной строке после наименования команды, чтобы изменить ее поведение при выполнении. Флаги состоят из последовательностей дефисов и букв. Вот один из примеров. Команда `ls` по умолчанию не выводит скрытые папки и файлы, но мы можем добавить в команду флаг `--all`, чтобы увидеть их. Некоторые флаги поддерживаются в нескольких

синтаксических вариантах. Например, команда `ls -a` — это более короткая версия команды `ls --all`. Попробуйте выполнить их обе.

Команда `cd` (change directory — «сменить текущий каталог») выполняет переход в указанный каталог. Введите путь к каталогу сразу после имени команды, не забыв добавить пробел. В следующем примере выполняется переход в каталог `Desktop`:

```
(base) ~$ cd Desktop
```

Узнать имя текущего каталога можно с помощью команды `pwd`:

```
(base) ~/Desktop$ pwd
/Users/boris/Desktop
```

Если после команды `cd` добавить две точки, то она выполнит переход вверх на один уровень в иерархии каталогов:

```
(base) ~/Desktop$ cd ..
```

```
(base) ~$ pwd
/Users/boris
```

Терминал имеет мощную функцию автодополнения. Например, находясь в своем домашнем каталоге, введите `cd Des` и нажмите клавишу `Tab`, в результате имя каталога будет дополнено до `cd Desktop`. Терминал просматривает список доступных файлов и папок и обнаруживает, что введенному шаблону `Des` соответствует только `Desktop`. Если совпадений несколько, терминал дополнит только общую часть их названий, например, если текущий каталог содержит две папки, `Anaconda` и `Analytics`, и вы введете букву `A`, то терминал автоматически дополнит имя до `Ana`, подставив общие буквы в двух вариантах. Вам нужно будет ввести дополнительную букву и снова нажать клавишу `Tab`, чтобы терминал автоматически добавил оставшуюся часть имени папки.

Итак, вы узнали все необходимое, чтобы начать работу с диспетчером окружений `conda`. Переходите к разделу А.4, а мы пока встретимся с нашими друзьями — пользователями Windows — и настроим с ними среду `conda` еще для одной ОС!

А.3. ПРОЦЕСС УСТАНОВКИ В WINDOWS

Рассмотрим установку Anaconda на компьютер с Windows.

А.3.1. Установка Anaconda в Windows

Мастер установки Anaconda для Windows доступен в 32 и 64-разрядной версиях. Этот параметр описывает тип процессора, установленного на вашем компьютере. Если вы не знаете, какую версию программы загрузить,

откройте меню **Start** (Пуск) и выберите приложение **System Information** (Сведения о системе). В главном окне приложения вы увидите таблицу с двумя столбцами — **Item** (Элемент) и **Value** (Значение). Найдите элемент **System Type** (Тип); его значение будет включать символы **x64**, если на вашем компьютере установлена 64-разрядная версия Windows, или **x86**, если установлена 32-разрядная версия Windows. На рис. А.4 показано окно приложения **System Information** (Сведения о системе) на компьютере с Windows, выделена именно строка **System Type** (Тип).

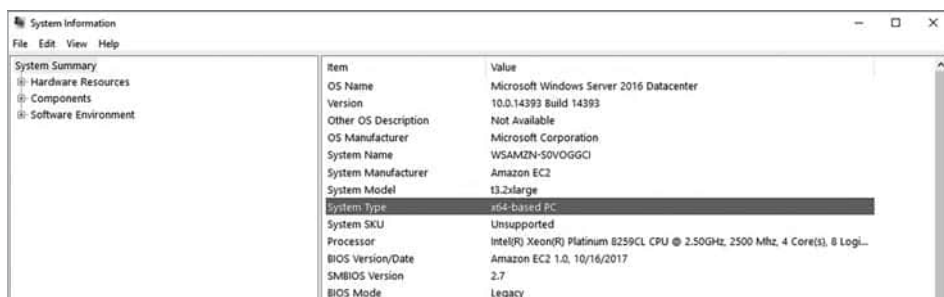


Рис. А.4. Окно приложения **System Information** (Сведения о системе) на компьютере с 64-разрядной ОС Windows

Загруженный вами дистрибутив Anaconda будет состоять из единственного выполняемого файла **.exe**. Имя файла будет включать номер версии Anaconda и название операционной системы (например, **Anaconda3-2021.05-Windows-x86_64**). Найдите файл дистрибутива в файловой системе и дважды щелкните на нем кнопкой мыши, чтобы запустить установку.

Пройдите несколько первых диалогов мастера установки. На этом пути вам будет предложено принять лицензионное соглашение, выбрать установку Anaconda для одного или всех пользователей и выбрать каталог установки. Если вы не знаете определенно, что выбрать, оставляйте значения по умолчанию.

Дойдя до диалога **Advanced Installation Options** (Дополнительные параметры установки), снимите флажок **Register Anaconda As My Default Python** (Зарегистрировать Anaconda как версию Python по умолчанию), если на вашем компьютере уже установлен Python. Если снять этот флажок, то Anaconda не будет регистрировать себя в качестве версии Python, используемой по умолчанию. А вот если вы устанавливаете Python в первый раз, оставьте этот флажок включенным.

В ходе установки будет создана начальная среда **conda** с именем **base**, включающая более 250 предварительно выбранных пакетов для анализа данных. Позже вы сможете создать дополнительные среды.

Установка может занять до нескольких минут. На рис. А.5 показано, как отображается процесс установки в окне мастера. По завершении процесса выйдите из программы установки.

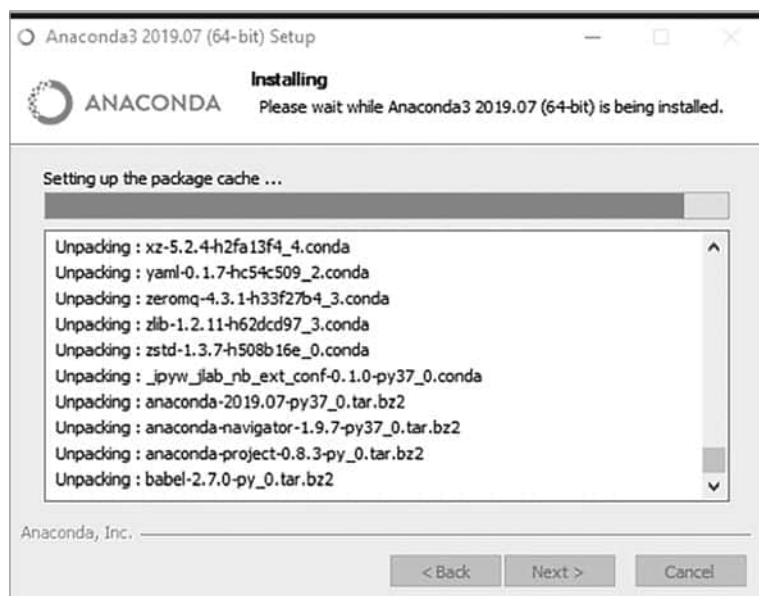


Рис. А.5. Процесс установки Anaconda на компьютер с Windows

Если когда-нибудь вы решите удалить Anaconda, то откройте меню **Start** (Пуск) и выберите **Add or Remove Programs** (Установка и удаление программ). Найдите программу Anaconda, выберите ее и нажмите кнопку **Uninstall** (Удалить), после чего следуйте инструкциям на экране, чтобы удалить дистрибутив с вашего компьютера. Обратите внимание, что этот процесс удалит все окружение **conda**, а также установленные в нем пакеты и версии Python.

А.3.2. Запуск командной оболочки Anaconda

В состав Anaconda входит программа с графическим интерфейсом **Navigator**, которая упрощает создание окружения **conda** и управление им. Однако прежде, чем запустить ее, рассмотрим более традиционное приложение для отправки команд диспетчеру окружения **conda**. Прежде чем довериться удобству графического интерфейса, важно понять, как приложение **Navigator** решает поставленные перед ним задачи «за кулисами», что стоит за красивым и удобным графическим приложением.

Anaconda Prompt — это приложение для выполнения команд в операционной системе Windows. Вы вводите команду, а затем нажимаете клавишу **Enter**, чтобы выполнить ее. До появления современных графических интерфейсов пользователи взаимодействовали с компьютерами исключительно с использованием текстовых приложений, подобных этому. Откройте меню **Start** (Пуск), найдите приложение *Anaconda Prompt* и запустите его.

Anaconda Prompt всегда должна сообщать активную среду *conda* в круглых скобках перед приглашением к вводу с мигающим курсором. В настоящий момент вы должны видеть **base** — начальную среду, созданную мастером установки *Anaconda*. На рис. А.6 для примера показано окно программы *Anaconda Prompt* с активированной средой **base**.

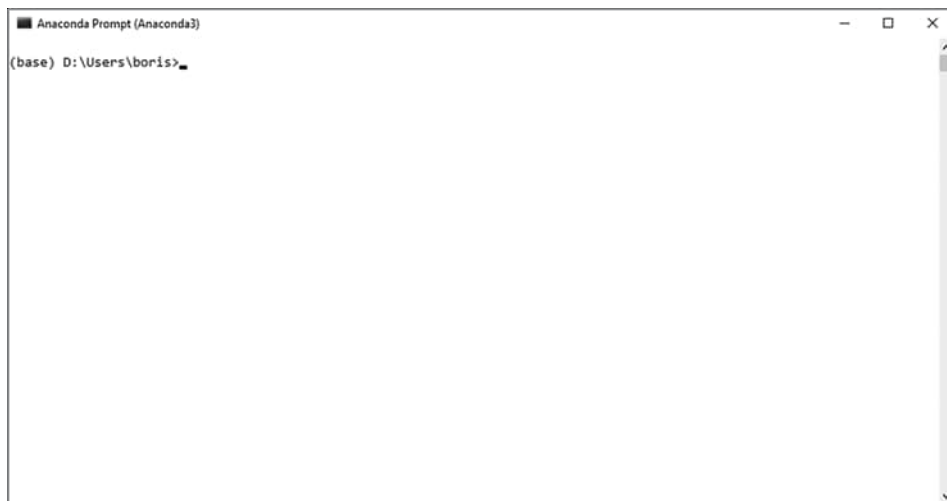


Рис. А.6. Окно программы *Anaconda Prompt* на компьютере с Windows. **base** — это активная среда *conda*

Anaconda Prompt активирует среду **base** в момент запуска. В разделе А.4 я расскажу, как создать и активировать новые окружения с помощью *conda*.

А.3.3. Типичные команды, доступные в *Anaconda Prompt*

Для эффективной работы в *Anaconda Prompt* достаточно запомнить всего несколько команд. Мы можем перемещаться по каталогам файловой системы компьютера так же, как в программе *Windows Explorer* (Проводник).

Команда **dir** (*directory* — «каталог») выводит список файлов и папок в текущем каталоге:

```
(base) C:\Users\Boris>dir
Volume in drive C is OS
Volume Serial Number is 6AAC-5705

Directory of C:\Users\Boris
08/15/2019 03:16 PM <DIR> .
08/15/2019 03:16 PM <DIR> ..
09/20/2017 02:45 PM <DIR> Contacts
08/18/2019 11:21 AM <DIR> Desktop
08/13/2019 03:50 PM <DIR> Documents
08/15/2019 02:51 PM <DIR> Downloads
09/20/2017 02:45 PM <DIR> Favorites
05/07/2015 09:56 PM <DIR> Intel
06/25/2018 03:35 PM <DIR> Links
09/20/2017 02:45 PM <DIR> Music
09/20/2017 02:45 PM <DIR> Pictures
09/20/2017 02:45 PM <DIR> Saved Games
09/20/2017 02:45 PM <DIR> Searches
09/20/2017 02:45 PM <DIR> Videos
        1 File(s) 91 bytes
       26 Dir(s) 577,728,139,264 bytes free
```

Команда `cd` (change directory — «сменить каталог») выполняет переход в указанный каталог. Введите путь к каталогу сразу после имени команды, не забыв добавить пробел между ними. В следующем примере выполняется переход в каталог `Desktop`:

```
(base) C:\Users\Boris>cd Desktop
(base) C:\Users\Boris\Desktop>
```

Если после команды `cd` добавить две точки, то она выполнит переход из текущего каталога на один уровень вверх в иерархии каталогов:

```
(base) C:\Users\Boris\Desktop>cd ..
(base) C:\Users\Boris>
```

`Anaconda Prompt` имеет мощную функцию автодополнения. Например, находясь в своем домашнем каталоге, введите `cd Des` и нажмите клавишу `Tab`, в результате имя каталога будет дополнено до `cd Desktop`. `Anaconda Prompt` просматривает список доступных файлов и папок и обнаруживает, что введенному шаблону `Des` соответствует только `Desktop`. Если совпадений несколько, `Anaconda Prompt` дополнит только общую часть их названия, например, если текущий каталог содержит две папки, `Anaconda` и `Analytics`, и вы введете букву `A`, то `Anaconda Prompt` автоматически дополнит имя до `Ana`, подставив общие буквы в обоих возможных вариантах. Вам нужно будет ввести дополнительную букву и снова нажать клавишу `Tab`, чтобы приложение `Anaconda Prompt` автоматически добавило оставшуюся часть имени папки.

На данный момент вы узнали все необходимое, чтобы начать работу с диспетчером окружения `conda`. Теперь приступим к созданию нашей первой среды `conda`!

A.4. СОЗДАНИЕ НОВЫХ ОКРУЖЕНИЙ ANACONDA

Поздравляю — вы успешно установили дистрибутив Anaconda на свой компьютер с macOS или Windows. Теперь создадим среду `conda`, которую можно использовать, работая с книгой. Обратите внимание, что примеры, представленные в этом разделе, взяты с компьютера с macOS. Экранные формы могут немного различаться в двух операционных системах, однако сами команды Anaconda одни и те же.

Откройте терминал в macOS или Anaconda Prompt в Windows. При этом должна активироваться среда по умолчанию `base`. Найдите слева от приглашения к вводу круглые скобки со словом `base` между ними.

Для начала убедимся, что успешно установили диспетчер окружения `conda`, выполнив какую-нибудь команду. Самая простая: спросить у `conda` номер версии. Обратите внимание, что у вас номер версии может отличаться от показанного в следующем примере, но если команда возвращает хоть какое-то число, значит, диспетчер `conda` успешно установлен:

```
(base) ~$ conda --version
conda 4.10.1
```

Команда `conda info` возвращает сведения технического характера о самой `conda`, включая название текущей активной среды и ее местоположение на жестком диске. Вот сокращенная версия вывода:

```
(base) ~$ conda info

active environment : base
active env location : /opt/anaconda3
shell level : 1
user config file : /Users/boris/.condarc
populated config files : /Users/boris/.condarc
conda version : 4.10.1
conda-build version : 3.18.9
python version : 3.7.4.final.0
```

Для настройки поведения команд диспетчера `conda` можно использовать флаги. *Флаг* — это конфигурационный параметр, добавляемый после команды в командной строке, чтобы изменить ее поведение при выполнении. Флаги состоят из последовательностей дефисов и букв. Флаг `--envs` заставляет команду `info` перечислить все окружения с их местоположениями на компьютере. Звездочкой (*) отмечена активная среда:

```
(base) ~$ conda info --envs
# conda environments:
#
base                  * /Users/boris/anaconda3
```


Все команды `conda` поддерживают флаг `--help`, который выводит описание самой этой команды. Добавим этот флаг в вызов команды `conda info`:

```
(base) ~$ conda info --help
порядок использования: conda info [-h] [--json] [-v] [-q] [-a] [--base] [-e]
                        [-s] [--unsafe-channels]
```

Отображает информацию о текущей установке `conda`.

Параметры:

необязательные аргументы:

<code>-h, --help</code>	Вывести это справочное сообщение и завершиться.
<code>-a, --all</code>	Показать всю информацию.
<code>--base</code>	Показать путь к среде <code>base</code> .
<code>-e, --envs</code>	Перечислить все известные окружения <code>conda</code> .
<code>-s, --system</code>	Перечислить переменные окружения.
<code>--unsafe-channels</code>	Вывести список каналов с токенами.

Параметры управления выводом, запросами и потоком выполнения:

<code>--json</code>	Всю информацию выводить в формате <code>json</code> . Подходит для использования <code>conda</code> из программ.
<code>-v, --verbose</code>	Использовать один раз для получения дополнительной информации, два раза при отладке, три раза при трассировке.
<code>-q, --quiet</code>	Не выводить индикатор прогресса.

Создадим новую среду для экспериментов. Сделать это можно с помощью команды `conda create`. Ей нужно передать флаг `--name` с именем среды. Я выбрал имя, подходящее для этой книги: `pandas_in_action`, но вы можете выбрать любое другое, которое вам нравится. Когда `conda` запросит подтверждение, введите `Y` (`yes` — «да») и нажмите `Enter`:

```
(base) ~$ conda create --name pandas_in_action
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /opt/anaconda3/envs/pandas_in_action

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# Активируйте эту среду командой
#
# $ conda activate pandas_in_action
```

```
#
# Деактивируйте активную среду командой
#
# $ conda deactivate
```

По умолчанию `conda` устанавливает в новой среде последнюю версию Python. Чтобы установить другую версию, добавьте ключевое слово `python` в конце команды, знак равенства и номер нужной версии. Следующий пример иллюстрирует, как создать среду с именем `sample` и установить в нее Python 3.7:

```
(base) ~$ conda create --name sample python=3.7
```

Удалить среду можно командой `conda env remove`. Ей нужно передать флаг `--name` с именем удаляемой среды. Следующий пример удаляет созданную нами среду `sample`:

```
(base) ~$ conda env remove --name sample
```

Теперь, создав среду `pandas_in_action`, мы можем ее активировать. Сделать это можно командой `conda activate`. Текст в скобках перед приглашением командной строки изменится, станет именем активированной среды:

```
(base) ~$ conda activate pandas_in_action
```

```
(pandas_in_action) ~$
```

Все команды `conda` выполняются в контексте активной среды. Например, если теперь попросить `conda` установить пакет Python, он будет установлен в среду `pandas_in_action`.

Нам нужно установить следующие пакеты:

- ядро библиотеки `pandas`;
- среду разработки `jupyter`, в которой мы будем писать код;
- библиотеки `bottleneck` и `numexpr` для ускорения.

Команда `conda install` загружает и устанавливает указанные пакеты в активную среду. А вот модификация команды установки: можно добавить четыре пакета сразу после команды, разделив их пробелами:

```
(pandas_in_action) ~$ conda install pandas jupyter bottleneck numexpr
```

Как упоминалось выше, эти четыре библиотеки имеют свои зависимости. Диспетчер окружений `conda` выведет список всех пакетов, которые необходимо установить. Ниже приводится сокращенная версия этого списка. Ничего страшного, если у себя вы увидите в списке другие библиотеки или номера версий, `conda` сама позаботится о совместимости.

```
Collecting package metadata (repodata.json): done
Solving environment: done
```

```
## Package Plan ##
```

```
environment location: /opt/anaconda3/envs/pandas_in_action
```

```
added / updated specs:
```

- bottleneck
- jupyter
- numexpr
- pandas

```
The following packages will be downloaded:
```

package	build	
-----	-----	
appnope-0.1.2	py38hecd8cb5_1001	10 KB
argon2-cffi-20.1.0	py38haf1e3a3_1	44 KB
async_generator-1.10	py_0	24 KB
certifi-2020.12.5	py38hecd8cb5_0	141 KB
cffi-1.14.4	py38h2125817_0	217 KB
ipython-7.19.0	py38h01d92e1_0	982 KB
jedi-0.18.0	py38hecd8cb5_0	906 KB
#... другие библиотеки		

Введите Y в ответ на запрос и нажмите Enter, чтобы установить все пакеты и их зависимости.

Если позже вы захотите вспомнить, какие пакеты установлены в среде, то выполните команду `conda list`, которая выведет полный список всех установленных библиотек с номерами их версий:

```
(pandas_in_action) ~$ conda list
```

```
# packages in environment at /Users/boris/anaconda3/envs/pandas_in_action:
#
# Name                        Version      Build Channel
jupyter                      1.0.0        py39hecd8cb5_7
pandas                      1.2.4        py39h23ab428_0
```

А если позже вы решите удалить какой-то пакет из среды, то выполните команду `conda uninstall`. Вот как будет выглядеть команда для удаления `pandas`:

```
(pandas_in_action) ~$ conda uninstall pandas
```

Теперь мы готовы исследовать нашу среду разработки. Запустите приложение Jupyter Notebook с помощью команды `jupyter notebook`:

```
(pandas_in_action) ~$ jupyter notebook
```

Эта команда запустит на вашем компьютере локальный сервер, под управлением которого будет действовать ядро Jupyter. Нам нужен постоянно работающий сервер, чтобы он мог следить за вводимым нами кодом на Python и немедленно его выполнять.

Приложение Jupyter Notebook должно открыться в веб-браузере, выбранном как браузер по умолчанию в вашей системе. Получить доступ к приложению можно также, открыв в браузере страницу `localhost:8888/`; здесь `localhost` — это имя вашего компьютера, а `8888` — номер порта, через который приложение принимает запросы. Подобно тому как док имеет несколько портов для приема кораблей, ваш компьютер (`localhost`) тоже имеет множество портов, что позволяет запускать несколько программ одновременно на локальном сервере. На рис. А.7 показан основной интерфейс приложения Jupyter Notebook со списком файлов и папок в текущем каталоге.

Интерфейс Jupyter Notebook напоминает интерфейс Finder в macOS или Windows Explorer (Проводник) в Windows. Папки и файлы организованы в алфавитном порядке. Вы можете щелкать кнопкой мыши на папках, чтобы войти в них, и использовать строку навигации вверху основного интерфейса Jupyter Notebook для выхода на более высокие уровни. Поэкспериментируйте с интерфейсом пару минут. Уяснив для себя, как работает навигация, закройте браузер.

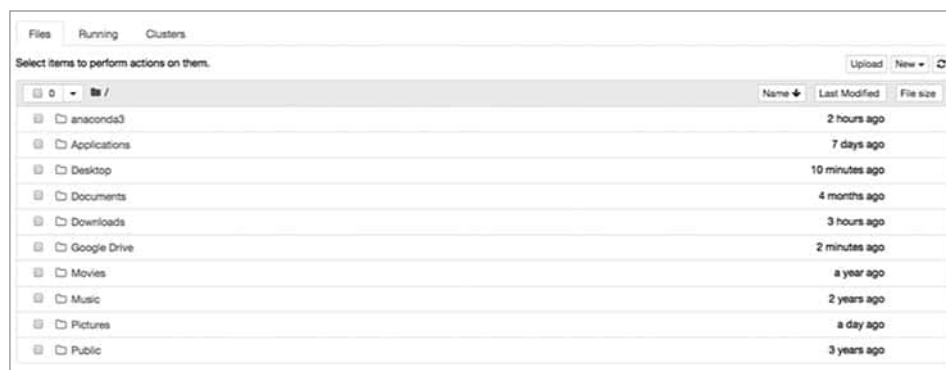


Рис. А.7. Основной интерфейс Jupyter Notebook

Обратите внимание, что после закрытия браузера сервер Jupyter продолжит работать. Чтобы остановить его, нужно дважды нажать `Ctrl+C` в терминале или Anaconda Prompt.

Примите на заметку также то, что каждый раз, когда запускается терминал в macOS или Anaconda Prompt в Windows, необходимо снова активировать среду `pandas_in_action`. Несмотря на то что среда `base` включает библиотеку `pandas`, я рекомендую создавать новую среду для каждой книги или учебника по Python,

с которыми вы работаете. Это обеспечит изоляцию зависимостей Python в разных проектах. Например, в одном руководстве может использоваться версия pandas 1.1.3, а в другом — pandas 1.2.0. При установке, обновлении и работе в изолированной среде оказывается ниже вероятность появления технических ошибок.

Напомню кратко, что нужно делать каждый раз, запуская терминал или **Anaconda Prompt**:

```
(base) ~$ conda activate pandas_in_action
(pandas_in_action) ~$ jupyter notebook
```

Первая команда активирует среду **conda**, а вторая запускает Jupyter Notebook.

A.5. ANACONDA NAVIGATOR

Anaconda Navigator — программа с графическим интерфейсом для управления окружениями **conda**. Она поддерживает не все возможности, которые предлагает инструмент командной строки **conda**, зато имеет наглядный и удобный интерфейс для создания и управления окружениями с помощью **conda**. Программа **Anaconda Navigator** находится в папке **Applications** (Программы) в **Finder** (macOS) или в меню **Start** (Пуск) (Windows). На рис. А.8 показано, как выглядит главное окно приложения **Anaconda Navigator**.

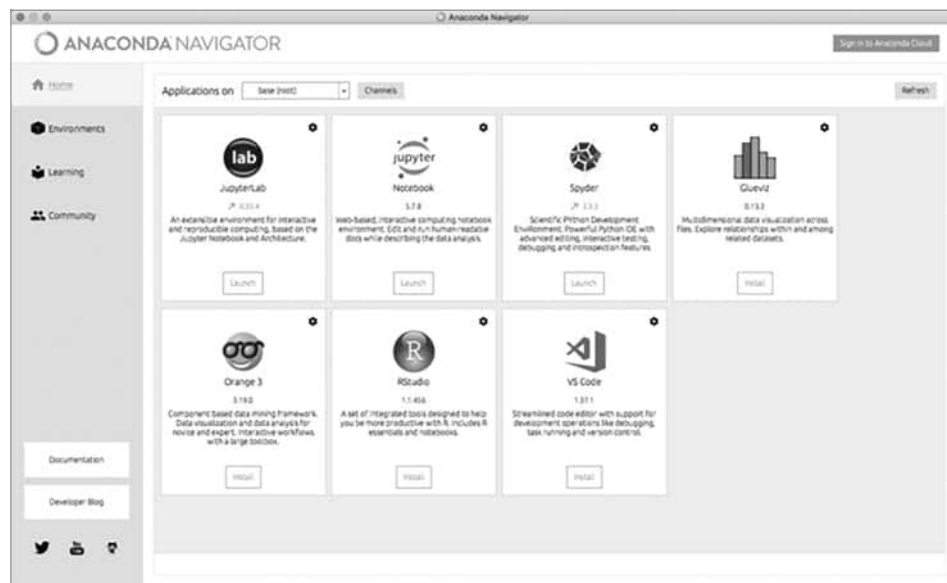


Рис. А.8. Главное окно приложения **Anaconda Navigator**

Щелкните кнопкой мыши на вкладке Environments (Окружение) на панели слева и в появившемся списке выберите нужную среду, чтобы увидеть, какие пакеты установлены, включая их описания и номера версий.

На нижней панели нажмите кнопку Create (Создать), чтобы запустить процесс создания новой среды. Дайте среде имя и выберите версию Python для установки. В появившемся диалоге вы увидите путь к папке (поле Location (Местоположение)), где conda создаст среду (рис. А.9).

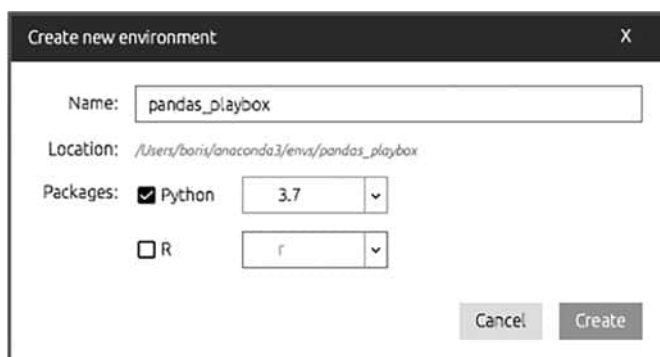


Рис. А.9. Создание новой среды Anaconda

Чтобы установить пакет, выберите среду в списке слева. Над перечнем пакетов выберите в раскрывающемся списке пункт All (Все), чтобы увидеть все пакеты (рис. А.10).

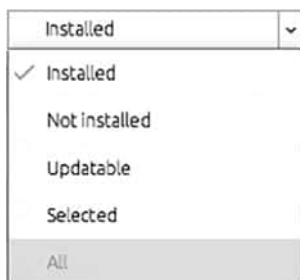


Рис. А.10. Поиск пакетов в Anaconda

В поле поиска справа введите название искомой библиотеки, например pandas. Отыщите ее в результатах поиска и установите соответствующий флажок (рис. А.11).

Затем нажмите зеленую кнопку Apply (Применить) в правом нижнем углу, чтобы установить библиотеку.

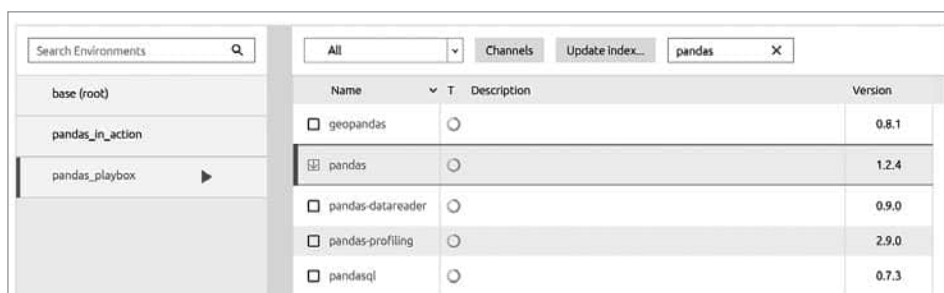


Рис. А.11. Поиск и выбор пакета pandas в Anaconda

Удалим созданную нами среду `pandas_playbox`. Она не нужна нам, потому что мы уже создали среду `pandas_in_action`. Для удаления выберите `pandas_playbox` в списке слева, затем нажмите кнопку **Remove** (Удалить) на нижней панели и еще раз — в диалоге подтверждения (рис. А.12).

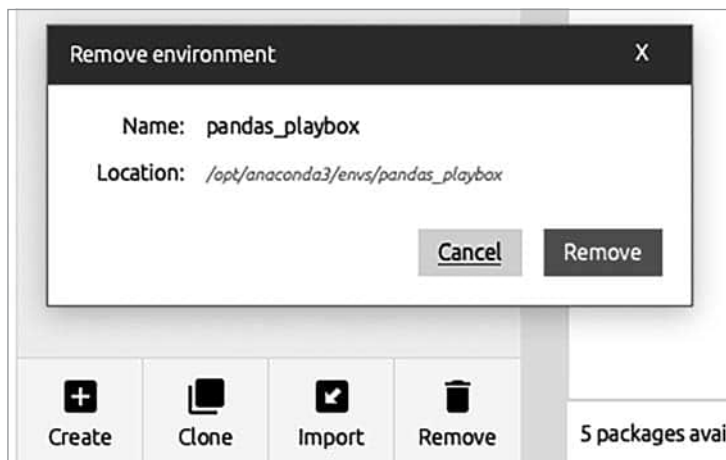


Рис. А.12. Удаление среды, созданной в Anaconda

Чтобы запустить Jupyter Notebook из Anaconda Navigator, выберите вкладку **Home** (Главная) на панели слева. На этой вкладке вы увидите плитки со значками приложений, установленных в текущей среде. В верхней части окна имеется раскрывающийся список, в котором можно выбрать активную среду `conda`. Обязательно выберите среду `pandas_in_action`, которую мы создали для этой книги. После этого можно запустить Jupyter Notebook, щелкнув на плитке со значком приложения. Это действие эквивалентно запуску Jupyter Notebook в приложении **Terminal** или **Anaconda Prompt**.

A.6. ОСНОВЫ JUPYTER NOTEBOOK

Jupyter Notebook — это интерактивная среда разработки для Python, состоящая из одной или нескольких ячеек, содержащих код на Python или разметку Markdown. *Markdown* — это стандарт форматирования текста, который можно использовать для добавления в блокноты Jupyter заголовков, текстовых абзацев, маркированных списков, встроенных изображений и многого другого. Код на Python используется для определения логики, а разметка Markdown — для оформления наших мыслей. По мере чтения книги не стесняйтесь использовать разметку Markdown, чтобы добавлять свои заметки. Полное описание Markdown доступно по адресу <https://daringfireball.net/projects/markdown/syntax>.

На первой странице блокнота Jupyter, открывшейся после запуска, щелкните на раскрывающемся списке New (Новый) вверху справа и выберите пункт Python 3, чтобы создать новый блокнот (рис. A.13).

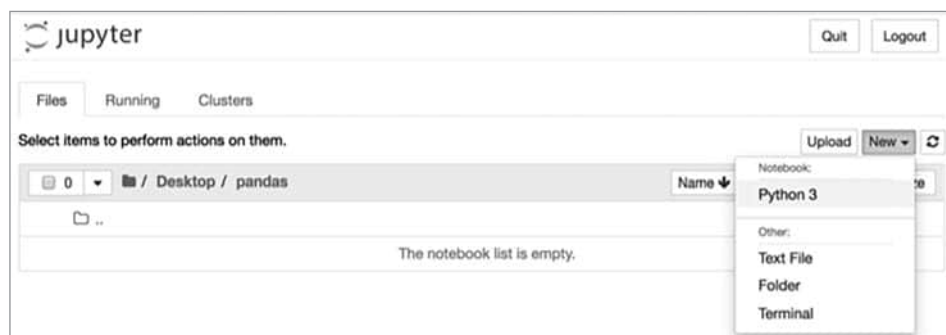


Рис. A.13. Создание нового блокнота Jupyter

Чтобы присвоить блокноту имя, щелкните на тексте *Untitled* (Без названия) вверху и введите имя в открывшемся диалоге. Jupyter Notebook сохраняет свои файлы с расширением *.ipynb*, сокращенно от IPython Notebooks, предшественника Jupyter Notebooks. Вернувшись на вкладку Jupyter Notebook, вы сможете увидеть в каталоге вновь созданный файл *.ipynb*.

Блокнот Jupyter работает в двух режимах: выполнения команд и правки. Щелчок кнопкой мыши на ячейке или нажатие клавиши *Enter*, когда ячейка находится в фокусе, запускает режим правки. В этом режиме Jupyter выделяет ячейку зеленой рамкой и интерпретирует нажатия клавиш на клавиатуре буквально. Мы используем этот режим для ввода символов в выбранную ячейку. На рис. A.14 показано, как выглядит ячейка в блокноте Jupyter в режиме редактирования.



Рис. А.14. Пустая ячейка в блокноте Jupyter в режиме правки

Ниже навигационного меню находится панель инструментов для выполнения типовых операций. В раскрывающемся списке справа отображается тип выбранной ячейки. Щелкните на этом списке и выберите Code (Код) или Markdown, чтобы изменить тип ячейки (рис. А.15).



Рис. А.15. Изменение типа ячейки в блокноте Jupyter

Одна из замечательных особенностей Jupyter Notebook — поддержка подхода к разработке методом проб и ошибок. Вы вводите в ячейку код на Python, а затем выполняете его. Jupyter выводит результат под ячейкой. Вы проверяете соответствие результата ожиданиям и продолжаете процесс. Такой подход поощряет активное экспериментирование: вам остается только нажать несколько клавиш, чтобы увидеть, что делает строка кода.

Выполним некоторый простой код на Python. Введите следующее выражение в первую ячейку блокнота, а затем нажмите кнопку Run (Запуск) на панели инструментов, чтобы выполнить его:

```
In [1]: 1 + 1
```

```
Out [1]: 2
```

Поле слева от кода (отображающее число 1 в предыдущем примере) показывает порядковый номер выполненной ячейки от момента запуска или перезапуска Jupyter Notebook. Вы можете выполнять ячейки в любом порядке и даже выполнять одну и ту же ячейку несколько раз.

Я призываю не упускать возможности поэкспериментировать, когда вы будете читать эту книгу, выполнять различные фрагменты кода в ячейках блокнота Jupyter. И не волнуйтесь, если номера выполняемых ячеек не будут совпадать с номерами в тексте книги.

Если ячейка содержит несколько строк кода, Jupyter выведет результат последнего выражения. Обратите внимание, что будет выполнен весь код в ячейке,

но мы увидим результат выполнения только последнего выражения, без индикации выполнения промежуточных команд.

```
In [2]: 1 + 1  
        3 + 2
```

```
Out [2]: 5
```

Интерпретатор — это программное обеспечение, которое анализирует исходный код на Python и выполняет его. Jupyter Notebook использует IPython (Interactive Python) — усовершенствованный интерпретатор с дополнительными функциями для повышения производительности труда разработчиков. Например, вы можете нажать клавишу **Tab** для получения справки о доступных методах и атрибутах любого объекта Python. В следующем примере показаны методы, поддерживаемые языком Python для строк. Введите любую строку, затем точку, а затем нажмите клавишу **Tab**, чтобы вывести подсказку, как показано на рис. А.16. Если вы не знакомы с основными структурами данных Python, то обращайтесь к приложению Б, где найдете исчерпывающее введение в язык.

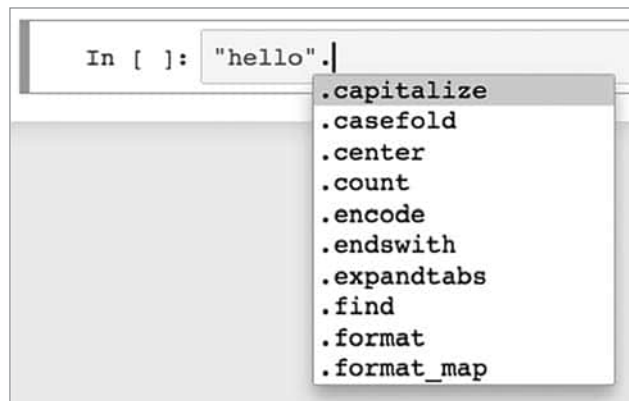


Рис. А.16. Поддержка автодополнения в Jupyter Notebook

В ячейку можно ввести код из любого количества символов, но в общем случае желательно сохранить размер ячейки, сделать код достаточно маленьким, чтобы его было проще читать и понимать. Если вы экспериментируете со сложной логикой, то разделите операции на несколько ячеек.

Чтобы выполнить код в ячейке блокнота Jupyter, можно использовать любую из двух комбинаций клавиш: **Shift+Enter**, чтобы выполнить ячейку и переместить фокус ввода на следующую ячейку, и **Ctrl+Enter**, чтобы выполнить ячейку

и сохранить фокус ввода в ней. Попрактикуйтесь в повторном выполнении первых двух ячеек, чтобы ощутить разницу в действии этих двух комбинаций.

Нажмите клавишу **Esc**, чтобы активировать командный режим, режим управления блокнотом. Доступные операции в этом режиме более глобальны; они влияют на блокнот в целом, а не на одну конкретную ячейку. В этом режиме нажатия некоторых горячих клавиш на клавиатуре интерпретируются как команды. В табл. А.1 представлено несколько полезных сочетаний клавиш, которые можно использовать в командном режиме.

Таблица А.1

Комбинация клавиш	Описание
Клавиши со стрелками вверх и вниз	Навигация по ячейкам в блокноте
a	Создать новую ячейку перед выбранной
b	Создать новую ячейку после выбранной
c	Копировать содержимое ячейки
x	Вырезать содержимое ячейки
v	Вставить содержимое скопированной или вырезанной ячейки в ячейку после выбранной
d+d	Удалить ячейку
z	Отменить удаление
y	Присвоить ячейке тип Code (Код)
m	Присвоить ячейке тип Markdown
h	Показать справочное меню, содержащее полный список коротких комбинаций клавиш
Command+S (macOS) или Ctrl+S (Windows)	Сохранить блокнот. Обратите внимание, что Jupyter Notebook поддерживает автоматическое сохранение

Чтобы очистить блокнот, выберите в главном меню **Kernel (Ядро)** пункт **Restart (Перезапуск)**. Также в этом меню доступны пункты, позволяющие очистить результаты выполнения ячеек и повторно выполнить все имеющиеся ячейки.

Допустим, мы достаточно много поэкспериментировали с блокнотом в течение дня и решили, что пора выходить из него. Jupyter Notebook продолжит выполняться в фоновом режиме даже после закрытия вкладки браузера. Чтобы остановить его, перейдите на вкладку **Running (Запустить)** вверху на начальной странице Jupyter и нажмите кнопку **Shutdown (Выключение)** рядом с названием блокнота (рис. А.17).



Рис. А.17. Остановка Jupyter Notebook

После остановки всех блокнотов следует завершить работу приложения Jupyter Notebook. Закройте вкладку браузера с приложением Jupyter. В приложении Terminal или Anaconda Prompt дважды нажмите Ctrl+C, чтобы остановить локальный сервер Jupyter.

Теперь вы готовы начать писать код на Python и pandas в Jupyter. Удачи!

Приложение Б

Экспресс-курс по языку Python

Библиотека `pandas` написана на Python — популярном языке программирования, впервые выпущенном в 1991 году голландским разработчиком Гвидо ван Россумом (Guido van Rossum). *Библиотека* (или *пакет*) — это набор инструментов, расширяющий возможности языка программирования. Библиотеки повышают производительность разработчиков, предоставляя автоматизированные решения повседневных рутинных задач, таких как подключение к базе данных, оценка качества кода и тестирование. Практически все проекты на Python используют библиотеки. В конце концов, зачем решать задачу с нуля, если кто-то уже решил ее? В каталоге Python Package Index (PyPI), централизованном онлайн-репозитории пакетов для Python, доступно более 300 000 библиотек. `Pandas` — одна из этих 300 000 библиотек; она реализует хранение и обработку сложных многомерных структур данных. Однако прежде, чем переходить к знакомству с `pandas`, желательно знать, что доступно в базовом языке.

Python — это язык объектно-ориентированного программирования (ООП). Парадигма ООП рассматривает программу как набор объектов, взаимодействующих друг с другом. *Объект* — это цифровая структура данных, хранящая информацию и предлагающая свои способы доступа к ней и управления ею. Каждый объект имеет определенную цель и сущность. Объекты можно рассматривать как актеров в пьесе, а программу — как спектакль.

Объекты можно считать цифровыми строительными блоками. Рассмотрим для примера программное обеспечение для работы с электронными таблицами, например Excel. Работая с программой на пользовательском уровне, мы можем видеть различия между книгой, листом и ячейкой. Книга содержит листы, лист содержит ячейки, а ячейки содержат значения. Мы рассматриваем эти три объекта как три отдельных контейнера бизнес-логики, каждый из которых играет

определенную роль. И взаимодействуем мы с ними по-разному. При создании объектно-ориентированных компьютерных программ разработчики мыслят аналогично, определяя и создавая «блоки», составляющие программу.

В сообществе Python часто можно услышать фразу: «Все является объектом». Это утверждение означает, что все типы данных в языке, даже простые, такие как числа и текст, реализованы как объекты. Библиотеки, такие как `pandas`, добавляют в язык новые типы объектов — дополнительные наборы строительных блоков.

Как аналитик данных, ставший инженером-программистом, я был свидетелем того, как для многих должностей в нашей отрасли знание языка Python стало требованием времени и жизненной необходимостью. По своему опыту могу сказать, что для эффективной работы с `pandas` не нужно быть продвинутым программистом. Однако базовое понимание основных механизмов Python значительно ускорит освоение библиотеки. В этом приложении освещаются ключевые основы языка, которые вам понадобятся, чтобы добиться успеха.

Б.1. ПРОСТЫЕ ТИПЫ ДАННЫХ

Данные бывают разных типов. Целое число, например 5, относится не к тому же типу, что вещественное число, например 8.46. И оба числа, 5 и 8.46, отличаются по типу данных от текстового значения, такого как "Bob".

Начнем со знакомства с основными типами данных в Python. Убедитесь, что установили дистрибутив Anaconda и настроили среду программирования `conda`, включающую Jupyter Notebook. Если вам нужна помощь, то обращайтесь за инструкциями по установке в приложение А. Активируйте среду `conda`, которую вы создали для этой книги, выполните команду `jupyter notebook` и создайте новый блокнот.

Небольшое примечание перед началом: в Python символ хеша (`#`) создает комментарий. *Комментарий* — это строка текста, которую интерпретатор Python игнорирует при обработке кода. Разработчики используют комментарии для включения дополнительных описаний в свой код. Например:

```
# Сложить два числа
1 + 1
```

Мы также можем добавить комментарий после фрагмента кода. Python игнорирует все, что следует за символом хеша в той же строке. Однако начальная часть строки с кодом (до хеша) выполняется нормально:

```
1 + 1 # Сложить два числа
```

Предыдущий пример дает результат 2, а вот в следующем примере не выводится ничего. Символ начала комментария фактически отключает строку, и Python просто игнорирует сложение.

```
# 1 + 1
```

Я использовал комментарии в ячейках кода везде в книге, чтобы сообщить дополнительную информацию о выполняемых операциях. Вам не нужно копировать комментарии в свой блокнот Jupyter.

Б.1.1. Числа

Целое число — это число без дробной части. Например, 20:

```
In [1] 20
```

```
Out [1] 20
```

Целое число может быть положительным, отрицательным или нулем. Отрицательные числа предваряются знаком минус (-):

```
In [2] -13
```

```
Out [2] -13
```

Число с плавающей точкой — это вещественное число, имеющее дробную часть. Для обозначения десятичной запятой используется символ точки. Примером числа с плавающей точкой может служить число 7.349:

```
In [3] 7.349
```

```
Out [3] 7.349
```

Целые числа и числа с плавающей точкой представляют разные типы данных в Python или, что то же самое, разные объекты. Внешне они различаются наличием десятичной точки. Например, значение 5.0 — это объект с плавающей точкой, тогда как 5 — это целочисленный объект.

Б.1.2. Строки

Строка — это текст из нуля или более символов. Строка объявляется заключением фрагмента текста в пару одинарных, двойных или тройных кавычек. Каждый из трех вариантов имеет свои особенности, но для начинающих осваивать Python они несущественны. Мы будем использовать двойные кавычки в этой книге. Jupyter Notebook выводит все три вида строк совершенно одинаково:

```
In [4] 'Good morning'
```

```
Out [4] 'Good morning'
```

```
In [5] "Good afternoon"
```

```
Out [5] 'Good afternoon'
```

```
In [6] "" "Good night" ""
```

```
Out [6] 'Good night'
```

Строки могут включать не только буквы, но также цифры, пробелы и другие символы. Рассмотрим следующий пример, где строка включает в себя семь букв, знак доллара, две цифры, пробел и восклицательный знак:

```
In [7] "$15 dollars!"
```

```
Out [7] '$15 dollars!'
```

Строки можно различать по наличию окружающих их кавычек. Многих новичков смущает такое значение, как "5" — строка, содержащая один цифровой символ. Но это действительно строка текста, здесь "5" не является целым числом.

Пустая строка не содержит никаких символов. Ее можно создать с помощью пары кавычек, между которыми ничего нет:

```
In [8] ""
```

```
Out [8] ''
```

Под длиной строки подразумевается количество содержащихся в ней символов. Строка "Monkey business", например, имеет длину 15 символов; в слове **Monkey** шесть символов, в слове **business** — восемь и еще один символ — это пробел между словами.

Каждому символу в строке Python присваивает порядковый номер. Этот номер называется *индексом*, и нумерация индексов начинается с 0. В строке "car":

- "с" имеет индекс 0;
- "а" имеет индекс 1;
- "r" имеет индекс 2.

Индекс конечного символа в строке всегда на единицу меньше длины строки. Строка "car" имеет длину 3, а ее конечный символ имеет индекс 2. Начало отсчета индексов с нуля обычно сбивает с толку начинающих разработчиков, это правило трудно усвоить с первого раза, потому что в школе нас учат начинать считать с 1.

Мы можем извлечь из строки любой символ, указав его индекс. Для этого нужно после строки добавить пару квадратных скобок и значение индекса между ними. Следующий пример извлекает символ `h` в слове `Python`. Символ `h` — это четвертый символ в последовательности, поэтому он имеет индекс 3:

```
In [9] "Python"[3]
```

```
Out [9] 'h'
```

Чтобы извлечь символы из конца строки, можно использовать отрицательные индексы в квадратных скобках (начинать отсчет с `-1` в направлении от конца строки). Значение `-1` соответствует последнему символу, `-2` — предпоследнему и т. д. Следующий пример извлекает четвертый символ с конца в слове `Python`: символ `t`:

```
In [10] "Python"[-4]
```

```
Out [10] 't'
```

Обращение `"Python"[2]` даст в результате тот же символ `t`, что и последний пример выше по тексту.

Для извлечения нескольких символов из строки можно использовать специальный синтаксис, выполняющий операцию извлечения *среза*. Чтобы получить срез, нужно поместить в квадратные скобки два числа, разделенные двоеточием. Левое число определяет начальный индекс, а правое — конечный индекс, причем символ с конечным индексом не включается в срез. Это довольно сложно уяснить, я знаю.

Следующий пример извлекает все символы, начиная с символа с индексом 2 (включительно) и заканчивая символом с индексом 5 (но не включая его). В результате в срез попадают символы `t` с индексом 2, `h` с индексом 3 и `o` с индексом 4:

```
In [11] "Python"[2:5]
```

```
Out [11] 'tho'
```

Если начальный индекс равен 0, его можно опустить, не указывать в командной строке, и результат от этого не изменится. Выбирайте любой вариант синтаксиса, какой вам больше нравится:

```
In [12] # Следующие две строки эквивалентны
        "Python"[0:4]
        "Python"[:4]
```

```
Out [12] 'Pyth'
```

Вот еще один прием: чтобы извлечь символы от какого-то начального индекса до конца строки, опустите конечный индекс. В следующем примере показаны два варианта извлечения символов, начиная с `h` (индекс 3) до конца строки `"Python"`:

```
In [13] # Следующие две строки эквивалентны
        "Python"[3:6]
        "Python"[3:]
```

```
Out [13] 'hon'
```

Можно также опустить оба индекса и оставить только двоеточие, чтобы получить срез «от начала до конца», то есть полную копию строки:

```
In [14] "Python"[:]
```

```
Out [14] 'Python'
```

В операции извлечения среза допускается смешивать положительные и отрицательные индексы. Давайте получим срез от символа с индексом 1 (`y`) до последнего символа в строке (`n`):

```
In [15] "Python"[1:-1]
```

```
Out [15] 'ytho'
```

Можно также передать необязательное третье число, чтобы установить величину *шага* — количество позиций до следующего извлекаемого символа. Следующий пример извлекает символы из позиций от 0 (включительно) до 6 (не включая его) с шагом 2. В результате получается срез, включающий символы `P`, `t` и `o`, находящиеся в позициях 0, 2 и 4:

```
In [16] "Python"[0:6:2]
```

```
Out [16] 'Pto'
```

Вот еще один интересный трюк: в третьем числе можно передать `-1`, чтобы получить срез в обратном направлении, от конца строки к началу. В результате получится перевернутая строка:

```
In [17] "Python"[::-1]
```

```
Out [17] 'nohtyP'
```

Операцию получения среза удобно использовать для извлечения фрагментов текста из больших строк. Тема работы с текстовыми объектами подробно рассмотрена в главе 6.

Б.1.3. Логические значения

Логический (или *булев*) тип данных представляет собой логическое выражение истины. Данные этого типа могут иметь только одно из двух значений: `True` или `False`. Булев тип назван в честь английского математика и философа Джорджа Буля (George Boole). Обычно данные этого типа моделируют отношение «или-или»: да или нет, включено или выключено, действительно или недействительно, активно или неактивно и т. д.

```
In [18] True
```

```
Out [18] True
```

```
In [19] False
```

```
Out [19] False
```

Логические значения часто получаются в результате вычислений или операций сравнения, которые мы увидим в подразделе Б.2.2.

Б.1.4. Объект `None`

Объект `None` представляет ничто, то есть отсутствие значения. Этот тип, как и логический, сложно понять, потому что он более абстрактный, чем конкретное значение, скажем целое число.

Предположим, что мы решили в течение недели ежедневно измерять температуру воздуха, но забыли сделать это в пятницу. Показания температуры для шести из семи дней будут целыми числами. Но как выразить температуру за пропущенный день? Мы можем записать, например, «отсутствует», «неизвестно» или просто поставить прочерк. Объект `None` в языке Python моделирует эту же идею, выраженную в значении данных. В языке должно быть что-то, с помощью чего можно было бы сообщить об отсутствии значения. Для этого требуется объект, который объявляет, что значение отсутствует, не существует или не требуется. Jupyter Notebook ничего не выводит, если попытаться заполнить ячейку с `None`:

```
In [20] None
```

Значения `None`, как и логические значения, обычно не создаются непосредственно, а получаются в результате некоторых манипуляций. Работая с книгой, мы исследуем этот объект более подробно.

Б.2. ОПЕРАТОРЫ

Оператор — это символ, выполняющий операцию. Классическим примером из начальной школы может служить оператор сложения: знак плюс (+). Значения, с которыми работает оператор, называются *операндами*. В выражении $3 + 5$:

- $+$ — это оператор;
- 3 и 5 — операнды.

В этом разделе мы рассмотрим различные математические и логические операторы, встроенные в Python.

Б.2.1. Математические операторы

Введем математическое выражение, показанное в начале раздела. Jupyter выведет результат непосредственно под ячейкой:

```
In [21] 3 + 5
```

```
Out [21] 8
```

Для удобочитаемости принято добавлять пробелы слева и справа от оператора. Следующие два примера иллюстрируют вычитание ($-$) и умножение ($*$):

```
In [22] 3 - 5
```

```
Out [22] -2
```

```
In [23] 3 * 5
```

```
Out [23] 15
```

$**$ — оператор возведения в степень. Следующий пример возводит число 3 в степень 5 (число 3 умножается само на себя 5 раз):

```
In [24] 3 ** 5
```

```
Out [24] 243
```

Оператор $/$ выполняет деление. Следующий пример делит 3 на 5:

```
In [25] 3 / 5
```

```
Out [25] 0.6
```

В математической терминологии результат деления одного числа на другое называется *частным*. Деление с помощью оператора $/$ всегда возвращает частное с плавающей точкой, даже если делитель делит делимое нацело:

```
In [26] 18 / 6
```

```
Out [26] 3.0
```

Деление нацело — это альтернативный тип деления, при котором из частного принудительно удаляется дробная часть. Выполняется такое деление оператором `//`, и в результате получается целое частное. Следующий пример демонстрирует различия между этими операторами деления:

```
In [27] 8 / 3
```

```
Out [27] 2.6666666666666665
```

```
In [28] 8 // 3
```

```
Out [28] 2
```

Оператор *деления по модулю* (`%`) возвращает остаток от деления нацело. Например, при делении 5 на 3 получается остаток 2:

```
In [29] 5 % 3
```

```
Out [29] 2
```

Операторы сложения и умножения можно использовать и со строками. Знак плюс объединяет две строки. На техническом языке этот процесс называется *конкатенацией*.

```
In [30] "race" + "car"
```

```
Out [30] 'racecar'
```

Оператор умножения повторяет строку указанное число раз:

```
In [31] "Mahi" * 2
```

```
Out [31] 'MahiMahi'
```

Тип объекта определяет поддерживаемые им операции и операторы. Например, мы можем делить целые числа, но не можем делить строки. Ключевым навыком в ООП является определение типа объекта и операций, которые над ним можно выполнять.

Мы можем объединить две строки или сложить два числа. Но что произойдет, если попытаться сложить строку и число?

```
In [32] 3 + "5"
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-9-d4e36ca990f8> in <module>
----> 1 3 + "5"
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Ошибка! Этот пример — наше первое знакомство с ошибками в Python. Он демонстрирует одну из нескольких десятков ошибок, встроенных в язык и контролируемых им. Технически ошибки называются *исключениями*. Как и все остальное в Python, исключение — это объект. Всякий раз, когда мы допускаем синтаксическую или логическую ошибку, Jupyter Notebook выводит сообщение, включающее название ошибки и номер строки, вызвавшей ее. Для обозначения факта появления исключения часто используется технический термин «*возбуждение (а также “вызов”, “генерация”) исключения*». Мы могли бы сказать так: «при попытке сложить число и строку Python сгенерировал исключение».

Python генерирует исключение `TypeError` при попытке выполнить операцию, не поддерживаемую типом данных. В предыдущем примере Python увидел число и знак плюс и предположил, что далее последует еще одно число, но вместо числа он получил строку, которую нельзя сложить с целым числом. В разделе Б.4.1 я покажу, как преобразовать целое число в строку (и наоборот).

Б.2.2. Операторы проверки на равенство и неравенство

Python считает два объекта равными, если они содержат одно и то же значение. Проверить равенство двух объектов можно, поместив их по разные стороны от оператора проверки на равенство (`==`). Этот оператор возвращает `True`, если два объекта равны. Напомню, что `True` — это логическое значение.

```
In [33] 10 == 10
```

```
Out [33] True
```

Будьте внимательны: оператор проверки равенства состоит из двух знаков равенства. Один знак равенства в языке Python выполняет совершенно другую операцию, которую мы рассмотрим в разделе Б.3.

Оператор проверки равенства возвращает `False`, если два объекта не равны. Итак, результатом операции могут явиться только логические значения `True` и `False`:

```
In [34] 10 == 20
```

```
Out [34] False
```

Вот несколько примеров применения оператора проверки равенства к строкам:

```
In [35] "Hello" == "Hello"
```

```
Out [35] True
```

```
In [36] "Hello" == "Goodbye"
```

```
Out [36] False
```

При сравнении строк учитывается регистр символов. В следующем примере одна строка начинается с заглавной **H**, а другая — со строчной **h**, поэтому Python считает эти две строки неравными:

```
In [37] "Hello" == "hello"
```

```
Out [37] False
```

Оператор неравенства (**!=**) выполняет обратную проверку; он возвращает **True**, если два объекта не равны. Например, числа 10 и 20 не равны:

```
In [38] 10 != 20
```

```
Out [38] True
```

Аналогично строка "Hello" не равна строке "Goodbye":

```
In [39] "Hello" != "Goodbye"
```

```
Out [39] True
```

Оператор неравенства возвращает **False**, если два объекта равны:

```
In [40] 10 != 10
```

```
Out [40] False
```

```
In [41] "Hello" != "Hello"
```

```
Out [41] False
```

Python поддерживает математическое сравнение чисел. Оператор **<** сравнивает два операнда и возвращает **True**, если операнд слева меньше операнда справа. Следующий пример проверяет, меньше ли число **-5**, чем **3**:

```
In [42] -5 < 3
```

```
Out [42] True
```

Оператор **>** сравнивает два операнда и возвращает **True**, если операнд слева больше операнда справа. Следующий пример проверяет, больше ли число **5**, чем **7**; результатом является значение **False**:

```
In [43] 5 > 7
```

```
Out [43] False
```

Оператор **<=** сравнивает два операнда и возвращает **True**, если операнд слева меньше операнда справа или равен ему. Следующий пример проверяет, действительно ли число **11** меньше числа **11** или равно ему:

```
In [44] 11 <= 11
```

```
Out [44] True
```

Оператор `>=` сравнивает два операнда и возвращает `True`, если операнд слева больше операнда справа или равен ему. Следующий пример проверяет, действительно ли число 4 больше числа 5 или равно ему:

```
In [45] 4 >= 5
```

```
Out [45] False
```

Pandas позволяет применять подобные сравнения к целым столбцам данных, но об этом подробнее говорится в главе 5.

Б.3. ПЕРЕМЕННЫЕ

Переменная — это имя, присвоенное объекту; переменную можно сравнить с почтовым адресом дома, потому что имя уникально идентифицирует объект. Имена переменных должны быть четкими и ясно описывать данные, которые хранит объект, а также его назначение в приложении. Имя `revenues_for_quarter4` лучше имени `r` или `r4`.

Присваивание переменной объекту производится с помощью оператора присваивания, одиночного знака равенства (`=`). В следующем примере объектам данных четырех разных типов (строка, целое число, число с плавающей точкой и логическое значение) назначаются четыре переменные (`name`, `age`, `high_school_gpa` и `is_handsome`):

```
In [46] name = "Boris"
        age = 28
        high_school_gpa = 3.7
        is_handsome = True
```

Выполнение ячейки с присваиванием переменной не дает никаких результатов в Jupyter Notebook, но впоследствии назначенную переменную можно использовать в любой ячейке. Переменная является заменой значения, которое она содержит:

```
In [47] name
```

```
Out [47] 'Boris'
```

Имя переменной должно начинаться с буквы или символа подчеркивания. За первой буквой могут следовать буквы, цифры или символы подчеркивания.

Как следует из названия, переменные могут содержать значения, меняющиеся в ходе выполнения программы. Назначим переменной `age` из примера выше новое значение — 35. После выполнения ячейки связь переменной `age` с ее прежним значением, 28, будет потеряна:


```
In [48] age = 35
      age
```

```
Out [48] 35
```

Одну и ту же переменную можно использовать по обе стороны оператора присваивания. Python всегда сначала вычисляет выражение справа от знака равенства. В следующем примере Python увеличивает значение `age`, прибавляя к переменной число `10`. Полученная сумма, `45`, сохраняется в переменной `age`:

```
In [49] age = age + 10
      age
```

```
Out [49] 45
```

Python — это язык с *динамической типизацией*, то есть переменные в нем ничего не знают о типах данных, которые они представляют, не закрепляются за ними. Переменная — это имя-прототип, которое может представлять, замещать любой объект в программе. Только объект знает свой тип данных. Поэтому мы можем переназначать переменные объектам разных типов. В следующем примере переменная `high_school_gpa`, первоначально связанная со значением с плавающей точкой, переназначается строке `"A+"`:

```
In [50] high_school_gpa = "A+"
```

При попытке сослаться на несуществующую переменную Python генерирует исключение `NameError`:

```
In [51] last_name
-----
NameError                                Traceback (most recent call last)
<ipython-input-5-elaeda7b4fde> in <module>
----> 1 last_name

NameError: name 'last_name' is not defined
```

Обычно можно столкнуться с исключением `NameError`, допустив опечатку в имени переменной. Этого исключения не нужно бояться; просто исправьте написание имени и снова выполните ячейку.

Б.4. ФУНКЦИИ

Функция — это процедура, состоящая из одного или нескольких шагов. Функцию можно представить как кулинарный рецепт на языке программирования, содержащий последовательность инструкций, дающих ожидаемый результат. Функции обеспечивают возможность повторного использования кода в программном обеспечении. Поскольку функция определяет конкретную часть

бизнес-логики от начала до конца, ее можно повторно использовать, когда потребуется выполнить одну и ту же операцию несколько раз.

Функции сначала объявляются, а затем выполняются. В объявлении записываются шаги, которые должна выполнить функция. Чтобы выполнить функцию, мы запускаем ее. Если придерживаться нашей кулинарной аналогии, то объявление функции эквивалентно записи рецепта, а выполнение — приготовлению блюда по этому рецепту. На техническом языке запуск функции называется *вызовом*.

Б.4.1. Аргументы и возвращаемые значения

Python включает более 65 встроенных функций. Кроме того, мы можем объявить свои собственные функции. Рассмотрим пример. Встроенная функция `len` возвращает длину указанного объекта. Понятие длины зависит от типа данных; для строки это количество символов.

Мы вызываем функцию, вводя ее имя и пару круглых скобок. Точно так же, как в кулинарный рецепт могут подставляться ингредиенты, в вызов функции можно передавать входные данные, называемые *аргументами*. Передаваемые аргументы перечисляются внутри круглых скобок через запятую.

Функция `len` принимает один аргумент: объект, длину которого нужно вычислить. В следующем примере функции `len` передается строковый аргумент `"Python is fun"`:

```
In [52] len("Python is fun")
```

```
Out [52] 13
```

Приготовление по рецепту дает окончательный результат — готовое блюдо. Точно так же функция создает конечный результат, называемый *возвращаемым значением*. В предыдущем примере мы вызвали функцию `len`, ее единственным аргументом была строка `"Python is fun"`, а возвращаемым значением стало число 13.

Вот и все! Функция — это процедура, которая вызывается с аргументами или без них и возвращает некоторое значение.

Вот еще три популярные встроенные функции в Python:

- `int` — преобразует аргумент в целое число;
- `float` — преобразует аргумент в число с плавающей точкой;
- `str` — преобразует аргумент в строку.

Следующие три примера демонстрируют, как действуют эти функции. В первом примере вызывается функция `int` со строковым аргументом `"20"`, которая

возвращает целочисленное значение 20. Сможете вы сами определить, какие аргументы принимают и какие значения возвращают остальные две функции?

```
In [53] int("20")
```

```
Out [53] 20
```

```
In [54] float("14.3")
```

```
Out [54] 14.3
```

```
In [55] str(5)
```

```
Out [55] '5'
```

Вот еще одна распространенная ошибка, контролируемая Python: если функции передать аргумент с правильным типом данных, но с неподходящим значением, то Python сгенерирует исключение `ValueError`. В примере ниже функция `int` получает строку (аргумент подходящего типа), из которой нельзя извлечь целое число:

```
In [56] int("xyz")
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-6-ed77017b9e49> in <module>
----> 1 int("xyz")
```

```
ValueError: invalid literal for int() with base 10: 'xyz'
```

Еще одна популярная встроенная функция — `print`, которая выводит текст на экран. Она принимает произвольное количество аргументов. Эту функцию удобно использовать для наблюдения за значением переменной во время выполнения программы. В следующем примере функция `print` вызывается четыре раза с переменной `value`, значение которой изменяется несколько раз:

```
In [57] value = 10
        print(value)

        value = value - 3
        print(value)

        value = value * 4
        print(value)

        value = value / 2
        print(value)
```

```
Out [57] 10
         7
         28
         14.0
```

Если функция принимает несколько аргументов, они должны отделяться друг от друга запятыми. Разработчики часто добавляют пробел после каждой запятой для удобства.

Когда мы передаем функции `print` несколько аргументов, она выводит их последовательно. Обратите внимание, что в следующем примере три значения напечатаны через пробел:

```
In [58] print("Cherry", "Strawberry", "Key Lime")
```

```
Out [58] Cherry Strawberry Key Lime
```

Параметр — это имя, данное ожидаемому аргументу функции. Каждый аргумент в вызове функции соответствует какому-то ее параметру. В предыдущих примерах мы передали функции `print` последовательность аргументов без указания имен параметров.

При передаче некоторых аргументов необходимо явно указывать имена соответствующих параметров. Например, параметр `sep` (separator — «разделитель») функции `print` определяет строку, которую Python вставляет между печатаемыми значениями. Мы должны явно указать имя параметра `sep`, чтобы передать аргумент со строкой-разделителем. Присваивание аргумента именованному параметру функции производится с помощью знака равенства. В следующем примере выводятся те же три строки, но при этом функции `print` указывается, что она должна разделять их восклицательными знаками:

```
In [59] print("Cherry", "Strawberry", "Key Lime", sep = "!")
```

```
Out [59] Cherry!Strawberry!Key Lime
```

Но вернемся к предыдущему примеру, на шаг назад. Почему перед этим три значения были напечатаны через пробел?

Аргумент по умолчанию — это резервное значение, которое Python передает в параметре, если оно не указано явно в вызове функции. Параметр `sep` функции `print` имеет аргумент по умолчанию " ". Если вызвать функцию `print` без аргумента для параметра `sep`, то Python автоматически передаст этому параметру строку с одним пробелом. Следующие две строки кода дают один и тот же результат:

```
In [60] # Следующие две строки эквивалентны
        print("Cherry", "Strawberry", "Key Lime")
        print("Cherry", "Strawberry", "Key Lime", sep=" ")
```

```
Out [60] Cherry Strawberry Key Lime
        Cherry Strawberry Key Lime
```

Параметры, такие как `sep`, мы называем *именованными аргументами*. При передаче соответствующих аргументов мы должны указывать имена параметров.

Python требует, чтобы именованные аргументы передавались после последовательных. Вот еще один пример вызова функции `print`, где в параметре `sep` передается другой строковый аргумент:

```
In [61] print("Cherry", "Strawberry", "Key Lime", sep="!*")
```

```
Out [61] Cherry!*Strawberry!*Key Lime
```

Параметр `end` функции `print` настраивает строку, которая будет добавлена в конец вывода. Аргументом по умолчанию этого параметра является `"\n"`, специальный символ, который Python распознает как перенос строки. В следующем примере мы явно передаем тот же аргумент `"\n"` в параметре `end`:

```
In [62] print("Cherry", "Strawberry", "Key Lime", end="\n")
        print("Peach Cobbler")
```

```
Out [62] Cherry Strawberry Key Lime
        Peach Cobbler
```

В вызов функции можно передать несколько именованных аргументов. Правила при этом остаются прежними: аргументы должны отделяться друг от друга запятыми. В следующем примере функция `print` вызывается дважды. Первый вызов выводит три аргумента, разделяя их знаком `!"` и заканчивая вывод строкой `****`. Поскольку первый вызов не осуществляет перевода строки, вывод второго вызова продолжается там, где завершился первый:

```
In [63] print("Cherry", "Strawberry", "Key Lime", sep="!", end="****")
        print("Peach Cobbler")
```

```
Out [63] Cherry!Strawberry!Key Lime****Peach Cobbler
```

Приостановитесь ненадолго и подумайте о форматировании кода в предыдущем примере. Длинные строки трудно читать, особенно когда в вызов функции передается несколько параметров. Для решения этой проблемы сообщество Python предлагает несколько способов форматирования кода. Один из вариантов — поместить все аргументы в отдельную строку:

```
In [64] print(
        "Cherry", "Strawberry", "Key Lime", sep="!", end="****"
    )
```

```
Out [64] Cherry!Strawberry!Key Lime****
```

Другой вариант — поместить каждый аргумент в отдельную строку:

```
In [65] print(
        "Cherry",
        "Strawberry",
        "Key Lime",
    )
```

```

        sep="!",
        end="***",
    )

```

Out [65] Cherry!Strawberry!Key Lime***361 Functions

Все три примера кода этого функционала технически допустимы. Python поддерживает множество способов форматирования кода, и в этой книге я использую несколько вариантов, преследуя главную цель — удобочитаемость. Вы не обязаны следовать правилам форматирования, которых придерживаюсь я. И я постараюсь сообщать вам, какие различия в представлении кода являются техническими, а какие — эстетическими.

Б.4.2. Пользовательские функции

Мы можем объявлять свои собственные функции в наших программах. Цель функции — зафиксировать фрагмент бизнес-логики в виде единой процедуры, пригодной для многократного использования. В кругах разработчиков часто звучит мантра *DRY*, которая расшифровывается как *don't repeat yourself* — «не повторяйся». Эта аббревиатура предупреждает, что повторение, реализация одной и той же логики или поведения в разных фрагментах кода может привести к нестабильной работе программы. Чем больше мест, где имеется повторяющийся код, тем больше правок вам придется вносить, если требования изменятся. Функции решают проблему повторной отработки частей программы.

Рассмотрим пример. Предположим, что мы метеорологи, работающие с данными о погоде. Нам часто приходится в программе преобразовывать температуру из градусов Фаренгейта в градусы Цельсия. Существует простая формула преобразования, и написание функции для преобразования температуры из одной шкалы в другую — хорошая идея, потому что это позволит изолировать логику преобразования и повторно использовать ее по мере необходимости.

Определение функции начинается с ключевого слова `def`. За `def` следует имя функции, пара круглых скобок и двоеточие. Имена функций и переменных, состоящие из нескольких слов, желательно оформлять в соответствии с соглашением о змеиной нотации, когда слова разделяются символом подчеркивания, из-за чего имя напоминает змею. Назовем нашу функцию `convert_to_fahrenheit`:

```
def convert_to_fahrenheit():
```

Напомню, что *параметр* — это имя ожидаемого аргумента функции. Функция `convert_to_fahrenheit` должна принимать единственный параметр: температуру по Цельсию. Назовем этот параметр `celsius_temp`:

```
def convert_to_fahrenheit(celsius_temp):
```

Определив параметр при объявлении функции, мы должны передать аргумент для этого параметра при ее вызове. То есть всякий раз, вызывая `convert_to_fahrenheit`, мы должны передать значение для `celsius_temp`.

Далее нужно определить последовательность действий — шагов, которые должна выполнять функция. Шаги объявляются в теле функции, в разделе кода, располагаются с отступом под ее именем. В Python отступы устанавливают отношения между конструкциями в программе. Тело функции — это пример *блока* — части кода, вложенного в другую часть кода. Согласно PEP-8¹, руководству по стилю оформления кода, принятому в сообществе Python, каждая строка в блоке должна иметь отступ из четырех пробелов:

```
def convert_to_fahrenheit(celsius_temp):
    # Эта строка с отступом принадлежит функции
    # То же относится и к этой строке

# Эта строка не имеет отступа и не принадлежит функции convert_to_fahrenheit
```

Параметры функции можно использовать в ее теле. В нашем примере мы можем использовать параметр `celsius_temp` в любом месте в теле функции `convert_to_fahrenheit`.

В теле функции также можно объявлять переменные. Эти переменные называются *локальными*, потому что они привязаны к области видимости внутри функции. Python уничтожает локальные переменные, как только функция завершает работу.

Теперь реализуем программно логику преобразования! Согласно формуле преобразования температуры из градусов Цельсия в градусы Фаренгейта нужно градусы Цельсия умножить на 9/5 и прибавить к результату 32:

```
def convert_to_fahrenheit(celsius_temp):
    first_step = celsius_temp * (9 / 5)
    fahrenheit_temperature = first_step + 32
```

На данный момент наша функция правильно вычисляет температуру по Фаренгейту, но она не передает результат обратно вызывающей программе. Чтобы сделать это, нужно использовать ключевое слово `return`. Вернем результат внешнему миру:

```
In [66] def convert_to_fahrenheit(celsius_temp):
        first_step = celsius_temp * (9 / 5)
        fahrenheit_temperature = first_step + 32
        return fahrenheit_temperature
```

¹ PEP 8 — Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008>. (Перевод руководства на русский язык можно найти по адресу <https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html>. — *Примеч. пер.*)

Наша функция готова и мы можем проверить ее! Для вызова пользовательских функций используется тот же синтаксис с парой круглых скобок, что и для встроенных функций Python. В следующем примере показан вызов функции `convert_to_fahrenheit` с аргументом `10`. Python выполняет тело функции с параметром `celsius_temp`, равным `10`. Функция возвращает значение `50.0`:

```
In [67] convert_to_fahrenheit(10)
```

```
Out [67] 50.0
```

Так же как и для встроенных функций, вместо позиционного аргумента можно передать именованный. В следующем примере явно указывается имя параметра `celsius_temp`. Он эквивалентен предыдущему примеру:

```
In [68] convert_to_fahrenheit(celsius_temp = 10)
```

```
Out [68] 50.0
```

Хотя применение именованных аргументов не является обязательным, они помогают внести ясность, улучшить читаемость программы. В предыдущем примере становится проще понять, какие входные данные передаются функции `convert_to_fahrenheit`.

Б.5. МОДУЛИ

Модуль — это один файл с кодом на языке Python. *Стандартная библиотека* Python содержит более 250 модулей, встроенных в язык для повышения эффективности труда разработчика. Они предлагают поддержку множества технических операций, например служат для математических вычислений, анализа аудиофайлов и выполнения запросов к URL. Чтобы уменьшить потребление памяти программой, Python по умолчанию не загружает эти модули, мы должны явно импортировать нужные модули вручную.

Синтаксис импорта встроенных модулей и внешних пакетов идентичен: сначала вводится ключевое слово `import`, за ним следует имя модуля или пакета. Импортируем модуль `datetime` из стандартной библиотеки Python, реализующий операции с датами и временем:

```
In [69] import datetime
```

Псевдоним — это альтернативное имя импортируемого модуля — короткая ссылка, которую можно назначить модулю, чтобы не приходилось вводить его полное имя при обращении к его операциям. Выбор псевдонима зависит только от наших предпочтений, но многие разработчики на Python используют хорошо

zareкомендовавшие себя и устоявшиеся псевдонимы. Например, модуль `datetime` часто импортируется с псевдонимом `dt`. Назначение псевдонима производится с помощью ключевого слова `as`:

```
In [70] import datetime as dt
```

Теперь для ссылки на модуль `datetime` можно использовать более короткий псевдоним `dt`.

Б.6. КЛАССЫ И ОБЪЕКТЫ

Все типы данных, которые мы исследовали до сих пор, — целые числа, числа с плавающей точкой, логические значения, строки, исключения, функции и даже модули — являются объектами. *Объект* — это цифровая структура данных, контейнер для хранения, использования и управления данными определенного типа.

Класс — это макет, калька для создания объектов. Классы можно рассматривать как шаблоны, на основе которых Python строит объекты.

Объект, созданный из класса, называется *экземпляром* класса, а акт создания объекта из класса называется *созданием экземпляра*.

Встроенная функция `type` возвращает класс объекта, который был передан ей в качестве аргумента. В следующем примере функция `type` вызывается дважды с двумя разными строками: `"peanut butter"` и `"jelly"`. Содержимое строк отличается, но они созданы на основе одного и того же шаблона — класса `str`. Они обе являются строками:

```
In [71] type("peanut butter")
```

```
Out [71] str
```

```
In [72] type("jelly")
```

```
Out [72] str
```

Эти примеры достаточно просты и понятны. Функция `type` может пригодиться, когда нельзя с уверенностью определить, какому классу принадлежит объект. Если вы вызываете некоторую нестандартную функцию и не уверены, какой тип объекта она возвращает, то передайте ее возвращаемое значение функции `type`, чтобы выяснить это.

Литерал — это сокращенный синтаксис создания объекта из класса. Вы уже видели один пример — двойные кавычки, которые создают строки (`"hello"`). Более сложные объекты создаются иначе.

Модуль `datetime`, который мы импортировали в разделе Б.5, имеет класс `date`, моделирующий календарную дату. Предположим, что вам нужно представить день рождения Леонардо да Винчи, 15 апреля 1452 года, в виде объекта `date`.

Чтобы создать экземпляр из класса, нужно ввести имя класса и пару круглых скобок. Инструкция `date()`, например, создаст объект `date` из класса `date`. Синтаксис создания экземпляра класса идентичен вызову функции. Иногда при создании экземпляра объекта можно передать аргументы конструктору — функции, которая создает объекты. Первые три аргумента конструктора `date` представляют год, месяц и день, которые будут храниться в объекте `date`. Для нашего примера достаточно трех аргументов:

```
In [73] da_vinci_birthday = dt.date(1452, 4, 15)
        da_vinci_birthday
```

```
Out [73] datetime.date(1452, 4, 15)
```

Теперь у нас есть переменная `da_vinci_birthday`, хранящая объект `date`, представляющий дату: 15 апреля 1452 года.

Б.7. АТТРИБУТЫ И МЕТОДЫ

Атрибут — это часть внутренних данных, принадлежащих объекту, характеристика или деталь, раскрывающая информацию об объекте. Мы обращаемся к атрибутам объекта с помощью точечного синтаксиса. Примерами атрибутов объекта `date` могут служить `day`, `month` и `year`:

```
In [74] da_vinci_birthday.day
```

```
Out [74] 15
```

```
In [75] da_vinci_birthday.month
```

```
Out [75] 4
```

```
In [76] da_vinci_birthday.year
```

```
Out [76] 1452
```

Метод — это действие или команда, которую можно применить к объекту. Метод можно рассматривать как функцию, принадлежащую объекту. *Атрибуты* определяют *состояние* объекта, а методы — его поведение. Так же как функция, метод может принимать аргументы и возвращать значение.

Чтобы вызвать метод, нужно ввести его имя и пару круглых скобок. Не забудьте добавить точку между объектом и именем метода, как и в случае атрибутов. При-

мером метода объекта `date` может служить `weekday`. Метод `weekday` возвращает день недели в виде целого числа, 0 обозначает воскресенье, а 6 — субботу:

```
In [77] da_vinci_birthday.weekday()
```

```
Out [77] 3
```

Леонардо да Винчи родился в среду!

Простота и возможность повторного использования таких методов, как, например, `weekday`, — вот главная причина существования объекта `date`. Представьте, как сложно было бы смоделировать логику работы с датами в форме текстовых строк. Подумайте, как сложно было бы жить, если бы каждому разработчику приходилось создавать свое программное решение. Ой-ей-ей! Разработчики Python поняли, что пользователям придется работать с датами, а программистам удобнее и быстрее работать с готовыми шаблонами типовой обработки дат, и создали повторно используемый класс `date` для моделирования этой реальной конструкции.

Главный вывод из вышесказанного: стандартная библиотека Python предлагает разработчикам множество вспомогательных классов и функций для решения типовых задач. Однако по мере усложнения программ становится все труднее моделировать реальные идеи, используя только базовые объекты Python. Для решения этой проблемы разработчики добавили в язык возможность пользователям определять свои собственные объекты, моделирующие бизнес-логику определенной специфической предметной области. Эти объекты обычно объединяются в библиотеки. Именно так была создана и библиотека `pandas` — набор дополнительных классов для решения конкретных задач в области анализа данных.

Б.8. МЕТОДЫ СТРОК

Строковый объект имеет набор собственных методов. Вот несколько примеров.

Метод `upper` возвращает новую строку со всеми символами в верхнем регистре:

```
In [78] "Hello".upper()
```

```
Out [78] "HELLO"
```

Методы также можно вызывать относительно переменных. Напомню, что *переменная* — это имя-прототип, заменитель для объекта. Встретив в программе имя переменной, Python заменяет ее объектом, на который ссылается эта переменная. В следующем примере метод `upper` вызывается для строки, на

которую ссылается переменная `greeting`. Результат получается такой же, как и в предыдущем примере:

```
In [79] greeting = "Hello"
        greeting.upper()
```

```
Out [79] "HELLO"
```

Все объекты делятся на две категории: изменяемые и неизменяемые. Содержимое *изменяемого* объекта можно изменить. Изменить содержимое *неизменяемого* объекта невозможно. Строки, числа и логические значения являются примерами неизменяемых объектов, их невозможно изменить после создания. Строка `"Hello"`, проставленная явно в программе или присвоенная переменной, всегда будет строкой `"Hello"`. Как и число 5 всегда будет числом 5.

В предыдущем примере вызов метода `upper` не изменил исходную строку `"Hello"`, назначенную переменной `greeting`, а вернул новую строку со всеми буквами в верхнем регистре. Мы можем вывести переменную `greeting`, чтобы убедиться, что ее содержимое не изменилось:

```
In [80] greeting
```

```
Out [80] 'Hello'
```

Строка — неизменяемый объект, поэтому методы строки не могут изменить ее содержимого. В разделе Б.9 мы рассмотрим некоторые примеры изменяемых объектов.

Метод `lower` возвращает новую строку со всеми символами в нижнем регистре:

```
In [81] "1611 BROADWAY".lower()
```

```
Out [81] '1611 broadway'
```

Существует также метод `swapcase`, возвращающий новую строку, в которой каждый символ имеет регистр, противоположный регистру соответствующего символа в исходной строке. Буквы верхнего регистра преобразуются в нижний регистр, а буквы нижнего регистра — в верхний:

```
In [82] "uPsIdE dOwN".swapcase()
```

```
Out [82] 'UpSiDe Down'
```

Методы могут принимать аргументы. Взглянем на метод `replace`, который меняет местами все вхождения подстроки указанной последовательностью символов. Он действует подобно функции «Найти и заменить» в текстовом редакторе. Метод `replace` принимает два аргумента:

- искомую подстроку;
- значение для замены.

В следующем примере все вхождения "S" заменяются на "\$":

```
In [83] "Sally Sells Seashells by the Seashore".replace("S", "$")
```

```
Out [83] '$ally $ells $eashells by the $eashore'
```

В этом примере:

- "Sally Sells Seashells by the Seashore" — это исходный *объект* строки;
- `replace` — это вызываемый *метод* строки;
- "S" — *первый аргумент*, передаваемый в вызов метода `replace`;
- "\$" — *второй аргумент*, передаваемый в вызов метода `replace`;
- "\$ally \$ells \$eashells by the \$eashore" — *значение, возвращаемое* методом `replace`.

Метод может возвращать значение, тип которого отличается от типа исходного объекта. Например, метод `isspace` вызывается для строки, но возвращает логическое значение: `True`, если строка состоит только из пробелов; `False` — в противном случае.

```
In [84] " ".isspace()
```

```
Out [84] True
```

```
In [85] "3 Amigos".isspace()
```

```
Out [85] False
```

Строки имеют семейство методов для удаления пробельных символов. Метод `rstrip` (right strip — «отбросить справа») удаляет пробельные символы в конце строки:

```
In [86] data = "    10/31/2019 "
        data.rstrip()
```

```
Out [86] '    10/31/2019'367 String methods
```

Метод `lstrip` (left strip — «отбросить слева») удаляет пробельные символы в начале строки:

```
In [87] data.lstrip()
```

```
Out [87] '10/31/2019  '
```

Метод `strip` удаляет пробельные символы с обоих концов строки:

```
In [88] data.strip()
```

```
Out [88] '10/31/2019'
```

Метод `capitalize` преобразует первый символ строки в верхний регистр. Этот метод удобно использовать при работе с именами, названиями географических объектов или организаций:

```
In [89] "robert".capitalize()
```

```
Out [89] 'Robert'
```

Метод `title` преобразует в верхний регистр первый символ каждого слова в строке, определяя границы между словами по пробелам:

```
In [90] "once upon a time".title()
```

```
Out [90] 'Once Upon A Time'
```

Мы можем вызывать несколько методов подряд в одной инструкции. Этот стиль называется *цепочкой методов*. В следующем примере метод `lower` возвращает новый строковый объект, для которого тут же вызывается метод `title`. Значение, возвращаемое методом `title`, — это еще один новый строковый объект:

```
In [91] "BENJAMIN FRANKLIN".lower().title()
```

```
Out [91] 'Benjamin Franklin'
```

Ключевое слово `in` проверяет, существует ли подстрока в другой строке. Искомая подстрока помещается слева от ключевого слова `in`, а строка, в которой выполняется поиск, — справа. Операция возвращает логическое значение:

```
In [92] "tuna" in "fortunate"
```

```
Out [92] True
```

```
In [93] "salmon" in "fortunate"
```

```
Out [93] False
```

Метод `startswith` проверяет, начинается ли строка с указанной подстроки:

```
In [94] "factory".startswith("fact")
```

```
Out [94] True
```

Метод `endswith` проверяет, заканчивается ли строка с указанной подстроки:

```
In [95] "garage".endswith("rage")
```

```
Out [95] True
```

Метод `count` подсчитывает количество вхождений подстроки в строку. Следующий пример подсчитывает количество вхождений символа `e` в строку `"celebrate"`:

```
In [96] "celebrate".count("e")
```

```
Out [96] 3
```

Методы `find` и `index` определяют индекс символа или подстроки. Они возвращают индекс первого найденного вхождения аргумента. Напомню, что нумерация индексов начинается с 0. В следующем примере определяется индекс первой буквы `e` в строке `"celebrate"`. В данном случае возвращается индекс 1:

```
In [97] "celebrate".find("e")
```

```
Out [97] 1
```

```
In [98] "celebrate".index("e")
```

```
Out [98] 1
```

В чем разница между методами `find` и `index`? Если строка не содержит искомой подстроки, то `find` вернет `-1`, а `index` сгенерирует исключение `ValueError`:

```
In [99] "celebrate".find("z")
```

```
Out [99] -1
```

```
In [100] "celebrate".index("z")
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-5-bf78a69262aa> in <module>
----> 1 "celebrate".index("z")
```

```
ValueError: substring not found
```

Каждый метод предназначен для конкретной ситуации, ни один из вариантов не лучше и не хуже другого. Например, если для продолжения нормальной работы программы требуется оценить, присутствует ли в большой строке некоторая подстрока, то можно использовать метод `index` и предусмотреть реакцию программы на исключение в том случае, когда подстрока отсутствует. А если отсутствие подстроки не препятствует выполнению программы, то можно использовать метод `find`, чтобы избежать сбоя.

Б.9. СПИСКИ

Список — это контейнер для хранения объектов по порядку. Списки имеют двойное назначение: предоставить общий «контейнер» для хранения значений и обеспечить их упорядоченность. Значения в списке называются *элементами*. В других языках программирования эту структуру данных часто называют *массивом*.

Список объявляется с помощью пары квадратных скобок, внутри которых через запятую перечисляются элементы. В следующем примере создается список из пяти строк:

```
In [101] backstreet_boys = ["Nick", "AJ", "Brian", "Howie", "Kevin"]
```

Длина списка равна количеству элементов в нем. Помните функцию `len`? С ее помощью можно узнать количество участников в величайшей американской музыкальной группе Backstreet Boys:

```
In [102] len(backstreet_boys)
```

```
Out [102] 5
```

Список без элементов называется *пустым списком*. Его длина равна 0:

```
In [103] []
```

```
Out [103] []
```

Списки могут хранить элементы любых типов: строки, целые числа, числа с плавающей точкой, логические значения и т. д. Списки, содержащие элементы только одного типа, называются *гомогенными* или *однородными*. Следующие три списка однородны. Первый хранит целые числа, второй — числа с плавающей точкой, а третий — логические значения:

```
In [104] prime_numbers = [2, 3, 5, 7, 11]
```

```
In [105] stock_prices_for_last_four_days = [99.93, 105.23, 102.18, 94.45]
```

```
In [106] settings = [True, False, False, True, True, False]
```

Списки могут также хранить элементы разных типов. Списки, содержащие элементы разных типов, называются *гетерогенными* или *разнородными*. Следующий список содержит строку, целое число, логическое значение и число с плавающей точкой:

```
In [107] motley_crew = ["rhinoceros", 42, False, 100.05]
```

По аналогии со строками каждому элементу списка присваивается индекс. Индекс определяет место элемента в списке, а нумерация индексов начинается с 0. В следующем списке `Favorite_Foods` из трех элементов индексы назначены так, как указано ниже:

- "Sushi" имеет индекс 0;
- "Steak" — индекс 1;
- "Barbeque" — индекс 2.

```
In [108] favorite_foods = ["Sushi", "Steak", "Barbeque"]
```

Два коротких замечания по форматированию списков. Во-первых, Python позволяет добавлять запятую после последнего элемента списка. Запятая никак не влияет на список, это просто альтернативный синтаксис:

```
In [109] favorite_foods = ["Sushi", "Steak", "Barbeque",]
```


Во-вторых, некоторые руководства по оформлению программного кода на Python рекомендуют разбивать длинные списки так, чтобы каждый элемент занимал одну строку. Этот формат также технически не влияет на список. Вот как выглядит такой синтаксис:

```
In [110] favorite_foods = [  
        "Sushi",  
        "Steak",  
        "Barbeque",  
    ]
```

Во всех примерах этой книги я старался использовать наиболее удобочитаемый, по моему мнению стиль форматирования. Вы же можете использовать любой другой формат, удобный для вас.

Получить доступ к элементу списка можно по его индексу. Укажите индекс между парой квадратных скобок после списка (или имени переменной, которая на него ссылается):

```
In [111] favorite_foods[1]  
  
Out [111] 'Steak'
```

В подразделе Б.1.2 вы познакомились с синтаксисом извлечения срезов из строк. Этот синтаксис можно таким же образом использовать для работы со списками. Приведу пример, в котором извлекаются элементы с индексами от 1 до 3. Напомню, что в этом синтаксисе элемент с начальным индексом включается в результат, а элемент с конечным индексом — нет:

```
In [112] favorite_foods[1:3]  
  
Out [112] ['Steak', 'Barbeque']
```

Число перед двоеточием можно убрать, чтобы получить срез с начала списка. Следующий пример извлекает элементы от начала списка до элемента с индексом 2 (не включая его):

```
In [113] favorite_foods[:2]  
  
Out [113] ['Sushi', 'Steak']
```

Число после двоеточия можно убрать, чтобы получить срез до конца списка. Следующий пример извлекает элементы от элемента с индексом 2 до конца списка:

```
In [114] favorite_foods[2:]  
  
Out [114] ['Barbeque']
```

Если убрать оба числа, операция извлечения среза вернет копию полного списка:

```
In [115] favorite_foods[:]
```

```
Out [115] ['Sushi', 'Steak', 'Barbeque']
```

Наконец, можно указать в квадратных скобках третье необязательное число, чтобы организовать извлечение элементов с определенным шагом. Следующий пример извлекает элементы от элемента с индексом 0 (включительно) до элемента с индексом 3 (не включая его) с шагом 2:

```
In [116] favorite_foods[0:3:2]
```

```
Out [116] ['Sushi', 'Barbeque']
```

Во всех случаях операция извлечения среза возвращает новый список, оставляя старый неизменным.

Теперь перечислю некоторые методы списков. Метод `append` добавляет новый элемент в конец списка:

```
In [117] favorite_foods.append("Burrito")
         favorite_foods
```

```
Out [117] ['Sushi', 'Steak', 'Barbeque', 'Burrito']371 Lists
```

Помните нашу дискуссию об изменяемых и неизменяемых объектах? Список — это пример изменяемого объекта, то есть объекта, который *может* быть изменен. После создания списка можно добавлять в него новые элементы, удалять существующие или заменять их другими элементами. В предыдущем примере метод `append` изменил существующий список, на который ссылается переменная `favorite_foods`, а не создал новый.

Для сравнения: строка является примером неизменяемого объекта. Когда вызывается такой метод, как `upper`, Python возвращает новую строку, а исходная строка остается в неприкосновенности. Неизменяемые объекты, повторюсь, не могут быть изменены.

Списки имеют множество методов, изменяющих их. Метод `extend` добавляет несколько элементов в конец списка. Он принимает один аргумент — список со значениями для добавления:

```
In [118] favorite_foods.extend(["Tacos", "Pizza", "Cheeseburger"])
         favorite_foods
```

```
Out [118] ['Sushi', 'Steak', 'Barbeque', 'Burrito', 'Tacos', 'Pizza',
           'Cheeseburger']
```

Метод `insert` вставляет элемент в указанную позицию в списке. Его первый аргумент — индекс позиции, куда следует вставить новый элемент, а второй аргумент — сам новый элемент. Python вставляет новое значение в указанную позицию и сдвигает все следующие за ним элементы. В следующем примере строка "Pasta" вставляется в позицию 2, при этом элемент "Barbeque" и все последующие сдвигаются, увеличивают свой индекс на одну позицию. Вот так:

```
In [119] favorite_foods.insert(2, "Pasta")
         favorite_foods
```

```
Out [119] ['Sushi',
           'Steak',
           'Pasta',
           'Barbeque',
           'Burrito',
           'Tacos',
           'Pizza',
           'Cheeseburger']
```

Ключевое слово `in` позволяет проверить присутствие элемента в списке. Элемент `Pizza` присутствует в нашем списке `favorite_foods`, а `Caviar` — нет:

```
In [120] "Pizza" in favorite_foods
```

```
Out [120] True
```

```
In [121] "Caviar" in favorite_foods
```

```
Out [121] False
```

Оператор `not in` проверяет отсутствие элемента в списке. Он является полной противоположностью оператору `in`:

```
In [122] "Pizza" not in favorite_foods
```

```
Out [122] False
```

```
In [123] "Caviar" not in favorite_foods
```

```
Out [123] True
```

Метод `count` находит количество вхождений элемента в список:

```
In [124] favorite_foods.append("Pasta")
         favorite_foods
```

```
Out [124] ['Sushi',
           'Steak',
           'Pasta',
```

```
'Barbeque',
'Burrito',
'Tacos',
'Pizza',
'Cheeseburger',
'Pasta']
```

```
In [125] favorite_foods.count("Pasta")
```

```
Out [125] 2
```

Метод `remove` удаляет первое вхождение элемента из списка. Обратите внимание, что последующие вхождения элемента не удаляются:

```
In [126] favorite_foods.remove("Pasta")
         favorite_foods
```

```
Out [126] ['Sushi',
           'Steak',
           'Barbeque',
           'Burrito',
           'Tacos',
           'Pizza',
           'Cheeseburger',
           'Pasta']
```

Удалим второе вхождение строки `"Pasta"` в конце списка. Метод `pop` удаляет и возвращает, выводит в качестве результата последний элемент из списка:

```
In [127] favorite_foods.pop()
```

```
Out [127] 'Pasta'
```

```
In [128] favorite_foods
```

```
Out [128] ['Sushi', 'Steak', 'Barbeque', 'Burrito', 'Tacos', 'Pizza',
           'Cheeseburger']
```

Метод `pop` также принимает целочисленный аргумент с индексом значения, которое Python должен удалить. В следующем примере удаляется значение `"Barbeque"` в позиции 2, а строка `"Burrito"` и все последующие элементы сдвигаются, приобретают индекс, уменьшенный на одну позицию:

```
In [129] favorite_foods.pop(2)
```

```
Out [129] 'Barbeque'373 Lists
```

```
In [130] favorite_foods
```

```
Out [130] ['Sushi', 'Steak', 'Burrito', 'Tacos', 'Pizza', 'Cheeseburger']
```

Список может хранить любые объекты, включая другие списки. В следующем примере объявляется список с тремя вложенными списками. Каждый вложенный список содержит три целых числа:

```
In [131] spreadsheet = [  
        [1, 2, 3],  
        [4, 5, 6],  
        [7, 8, 9]  
    ]
```

Давайте на секунду задержимся на последнем примере. Видите ли вы какие-либо параллели с электронными таблицами? Вложенные списки — это один из способов представления многомерных табличных коллекций данных. Внешний список можно рассматривать как лист, а каждый внутренний — как строку данных.

Б.9.1. Итерации по спискам

Список — это пример объекта-коллекции. Он способен хранить множество значений — *коллекцию*. Под *итерациями* подразумевается последовательное перемещение по элементам объекта-коллекции.

Наиболее распространенный способ итераций по элементам списка — цикл `for`. Вот как выглядит его синтаксис:

```
for variable_name in some_list:  
    # Выполнить некоторое действие с элементом
```

Цикл `for` состоит из нескольких компонентов:

- ключевое слово `for`;
- имя переменной цикла, в которой будет сохраняться каждый следующий элемент списка;
- ключевое слово `in`;
- список, через который выполняются итерации;
- блок кода, выполняемый в каждой итерации; в этом блоке кода можно ссылаться на имя переменной цикла.

Напомню, что *блок кода* — это часть кода с отступом от левого края. В Python отступы устанавливают отношения между конструкциями в программе. Блок кода, следующий за именем функции, определяет логику работы этой функции. Точно так же блок, следующий за заголовком цикла `for`, определяет логику, выполняемую в каждой итерации.

Приведу пример, который выполняет итерации по списку из четырех строк и выводит длину каждой из них:

```
In [132] for season in ["Winter", "Spring", "Summer", "Fall"]:
          print(len(season))
```

```
Out [132] 6
          6
          6
          4
```

Этот цикл выполняет четыре итерации. Переменная `season` поочередно принимает значения "Winter", "Spring", "Summer" и "Fall". В каждой итерации текущая строка передается в вызов функции `len`. Функция `len` возвращает число, которое затем выводится на экран.

Предположим, что мы решили сложить длины строк. Для этого нужно объединить, сочетать цикл `for` с некоторыми другими концепциями Python. В следующем примере сначала создается переменная `letter_count`, инициализированная нулем и предназначенная для хранения накопленной суммы. Внутри блока цикла `for` определяется длина текущей строки с помощью функции `len` и прибавляется к промежуточной сумме. По завершении цикла `for` мы выводим значение `letter_count`:

```
In [133] letter_count = 0
          for season in ["Winter", "Spring", "Summer", "Fall"]:
              letter_count = letter_count + len(season)
          letter_count
```

```
Out [133] 22
```

Цикл `for` — наиболее распространенный способ выполнения итераций по спискам. Однако Python поддерживает также другой синтаксис, который мы обсудим в подразделе Б.9.2.

Б.9.2. Генераторы списков

Генераторы списков¹ предлагают компактный синтаксис создания списка из объекта-коллекции. Предположим, что у нас есть список из шести чисел:

```
In [134] numbers = [4, 8, 15, 16, 23, 42]
```

Допустим, что нам потребовалось создать новый список с квадратами этих чисел. Другими словами, мы хотим применить некоторую операцию к каждому элементу исходного списка. Одним из решений является перебор элементов исходного списка, возведение их в квадрат и добавление результатов в новый

¹ Их иногда еще называют списковыми включениями. — *Примеч. пер.*

список. Напомню, что добавить элемент в конец списка можно с помощью метода `append`:

```
In [135] squares = []
        for number in numbers:
            squares.append(number ** 2)
        squares
```

```
Out [135] [16, 64, 225, 256, 529, 1764]
```

Используя синтаксис генераторов списков, можно создать список квадратов одной строкой кода. Для этого нужно использовать пару квадратных скобок. Внутри скобок сначала описать операцию для применения к каждому элементу исходной коллекции, а затем указать саму исходную коллекцию, откуда будут извлекаться элементы.

Следующий пример все так же выполняет итерации по списку чисел и сохраняет каждый из них в переменной `number`. Перед ключевым словом `for` описывается операция, которая должна применяться к переменной `number` в каждой итерации. В генераторах списков логика вычисления `number ** 2` указывается в начале, а логика `for` — в конце:

```
In [136] squares = [number ** 2 for number in numbers]
        squares
```

```
Out [136] [16, 64, 225, 256, 529, 1764]
```

Применение генераторов списков считается в Python почти «идиоматическим» способом создания новых списков на основе существующих структур данных. В коллекцию идиоматических способов входит множество рекомендуемых практик, сгенерированных и накопленных разработчиками Python с течением времени.

Б.9.3. Преобразование строки в список и обратно

Теперь мы знакомы со списками и строками, поэтому посмотрим, как можно использовать их вместе. Предположим, что в нашей программе есть строка, которая содержит адрес:

```
In [137] empire_state_bldg = "20 West 34th Street, New York, NY, 10001"
```

И нам понадобилось разбить этот адрес на более мелкие компоненты: название улицы, города, штата и почтовый индекс. Обратите внимание, что нужные нам компоненты адреса в строке разделены запятыми.

Метод `split` строк разбивает строку на части по указанному *разделителю* — последовательности из одного или нескольких символов, обозначающих границу.

472 Приложения

В следующем примере метод `split` разбивает `empire_state_building` по запятым и возвращает список, состоящий из более коротких строк:

```
In [138] empire_state_bldg.split(",")
```

```
Out [138] ['20 West 34th Street', ' New York', ' NY', ' 10001']
```

Этот код является шагом в правильном направлении. Но обратите внимание, что последние три элемента содержат пробелы в начале. Мы, конечно, могли бы применить к каждому элементу метод `strip`, чтобы удалить пробелы, но проще будет добавить пробел в аргумент метода `split`, определяющий разделитель:

```
In [139] empire_state_bldg.split(", ")
```

```
Out [139] ['20 West 34th Street', 'New York', 'NY', '10001']
```

Так мы успешно разбили строку на список строк.

Есть возможность выполнить и обратную операцию. Предположим, что мы храним адрес в списке и хотим объединить элементы списка в одну строку:

```
In [140] chrysler_bldg = ["405 Lexington Ave", "New York", "NY", "10174"]
```

Сначала нужно объявить строку, которую Python должен вставлять между соседними элементами списка. А после этого можно вызвать строковый метод `join` и передать ему список в качестве аргумента. Python объединит элементы списка, разделив их указанным разделителем. В следующем примере используется разделитель из запятой и пробела:

```
In [141] ", ".join(chrysler_bldg)
```

```
Out [141] '405 Lexington Ave, New York, NY, 10174'
```

Методы `split` и `join` удобно использовать при работе с текстовыми данными, которые часто необходимо разделять и объединять.

Б.10. КОРТЕЖИ

Кортеж — это структура данных, подобная списку. Кортеж тоже хранит элементы по порядку, но, в отличие от списка, он неизменяемый. Внутри кортежа после его создания нельзя добавлять, удалять или заменять элементы.

Единственным техническим требованием к определению кортежа является перечисление нескольких элементов через запятую. В следующем примере является кортеж из трех элементов:

```
In [142] "Rock", "Pop", "Country"
```

```
Out [142] ('Rock', 'Pop', 'Country')
```


Однако обычно объявление кортежа заключается в круглые скобки. Такой синтаксис позволяет легко отличать кортежи от других структур:

```
In [143] music_genres = ("Rock", "Pop", "Country")
         music_genres
```

```
Out [143] ('Rock', 'Pop', 'Country')
```

Определить длину кортежа можно с помощью функции `len`:

```
In [144] len(music_genres)
```

```
Out [144] 3
```

Чтобы объявить кортеж с одним элементом, нужно поставить запятую после этого элемента. Заключительная запятая используется интерпретатором Python для идентификации кортежей. Сравните различия в следующих двух выходных данных. В первом примере запятая не используется, и Python воспринимает значение как строку.

```
In [145] one_hit_wonders = ("Never Gonna Give You Up")
         one_hit_wonders
```

```
Out [145] 'Never Gonna Give You Up'
```

Для сравнения: следующий синтаксис возвращает кортеж. Да, в Python единственный символ может иметь большое значение:

```
In [146] one_hit_wonders = ("Never Gonna Give You Up",)
         one_hit_wonders
```

```
Out [146] ('Never Gonna Give You Up',)
```

Функция `tuple` создает пустой кортеж, то есть кортеж, не имеющий элементов:

```
In [147] empty_tuple = tuple()
         empty_tuple
```

```
Out [147] ()
```

```
In [148] len(empty_tuple)
```

```
Out [148] 0
```

По аналогии со списками к элементам кортежа можно обращаться по индексам, можно также выполнять итерации по элементам кортежа с помощью цикла `for`. Единственное, чего нельзя сделать, так это изменить кортеж. Вследствие своей неизменяемости кортежи не поддерживают такие методы, как `append`, `pop` и `insert`.

Если имеется упорядоченный набор элементов и известно, что он не изменится, то для его хранения лучше использовать кортеж, а не список.

Б.11. СЛОВАРИ

Списки и кортежи являются структурами данных, оптимальными для хранения объектов по порядку. Но для установления связей между объектами и последующей работы с ними нужна другая структура данных.

Представьте меню ресторана. Каждый пункт меню — это уникальный идентификатор, по которому мы можем определить соответствующую цену. Пункт меню и стоимость соответствующего блюда связаны. Порядок элементов неважен, потому что связываются две части данных.

Словарь — это изменяемый неупорядоченный набор пар «ключ/значение». Пара, как указано только что, состоит из ключа и значения. Каждый ключ служит идентификатором значения. Ключи должны быть уникальными, а значения могут повторяться.

Словари объявляются с помощью пары фигурных скобок (`{}`). В следующем примере создается пустой словарь:

```
In [149] {}
```

```
Out [149] {}
```

Смоделируем ресторанное меню на Python. Внутри фигурных скобок мы свяжем ключ с его значением с помощью двоеточия (`:`). В следующем примере объявляется словарь с одной парой «ключ/значение». Строковому ключу `"Cheeseburger"` присваивается значение с плавающей точкой `7.99`:

```
In [150] { "Cheeseburger": 7.99 }
```

```
Out [150] {'Cheeseburger': 7.99}
```

При объявлении словаря с несколькими парами «ключ/значение» они должны отделяться друг от друга запятыми. Расширим наш словарь `menu` и добавим в него еще две пары «ключ/значение». Обратите внимание, что значения ключей `"French Fries"` и `"Soda"` одинаковы:

```
In [151] menu = {"Cheeseburger": 7.99, "French Fries": 2.99, "Soda": 2.99}  
          menu
```

```
Out [151] {'Cheeseburger': 7.99, 'French Fries': 2.99, 'Soda': 2.99}
```

Определить количество пар «ключ/значение» в словаре можно с помощью встроенной функции `len`:

```
In [152] len(menu)
```

```
Out [152] 3
```

Для получения значений из словаря нужно использовать ключи. Поместите пару квадратных скобок с ключом сразу после словаря. Синтаксис идентичен синтаксису извлечения элемента из списка по индексу. В следующем примере извлекается значение, соответствующее ключу "French Fries":

```
In [153] menu["French Fries"]
```

```
Out [153] 2.99
```

В списке индекс элемента всегда является числом. В словаре ключ неизменяемый, но может быть любого типа: целым числом, числом с плавающей точкой, строкой, логическим значением и т. д.

Если указанного ключа в словаре нет, Python генерирует исключение `KeyError`. `KeyError` — еще один пример встроенной ошибки Python:

```
In [154] menu["Steak"]
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-19-0ad3e3ec4cd7> in <module>
----> 1 menu["Steak"]
```

```
KeyError: 'Steak'
```

Как обычно, регистр символов имеет значение. Если хотя бы один символ не совпадает, Python не сможет найти ключ. Ключа "soda" в нашем словаре нет. Есть только ключ "Soda":

```
In [155] menu["soda"]
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-20-47940ceca824> in <module>
----> 1 menu["soda"]
```

```
KeyError: 'soda'
```

Получить значение по ключу можно также с помощью метода `get`:

```
In [156] menu.get("French Fries")
```

```
Out [156] 2.99
```

Преимущество метода `get` в том, что он возвращает `None`, если искомый ключ отсутствует, а не вызывает ошибку. Напомню, что `None` — это объект, который Python использует для обозначения отсутствия чего-либо или пустого значения. Значение `None` не выводится в Jupyter Notebook. Но мы можем передать вызов метода `get` в качестве аргументов в функцию `print`, чтобы заставить Python напечатать строковое представление `None`:

```
In [157] print(menu.get("Steak"))
```

```
Out [157] None
```

Во втором аргументе можно передать методу `get` значение по умолчанию, которое будет возвращено, если он не найдет указанный ключ в словаре. В следующем примере выполняется попытка получить значение ключа `"Steak"`, который отсутствует в словаре `menu`, но вместо `None` Python возвращает указанное нами значение `99.99`:

```
In [158] menu.get("Steak", 99.99)
```

```
Out [158] 99.99
```

Словарь — это изменяемая структура данных. Мы можем добавлять и удалять пары «ключ/значение» из словаря после его создания. Чтобы добавить новую пару «ключ/значение», нужно указать желаемый ключ в квадратных скобках и присвоить ему значение с помощью оператора присваивания (`=`):

```
In [159] menu["Taco"] = 0.99
          menu
```

```
Out [159] {'Cheeseburger': 7.99, 'French Fries': 2.99, 'Soda': 1.99,
          'Taco': 0.99}
```

Если ключ уже существует в словаре, то Python заменит его прежнее значение. В следующем примере значение `7.99` ключа `"Cheeseburger"` меняется на `9.99`:

```
In [160] print(menu["Cheeseburger"])
          menu["Cheeseburger"] = 9.99
          print(menu["Cheeseburger"])
```

```
Out [160] 7.99
          9.99
```

Метод `pop` удаляет пару «ключ/значение» из словаря; он принимает ключ в качестве аргумента, удаляет из словаря и возвращает его значение. Если указанного ключа в словаре нет, то Python сгенерирует исключение `KeyError`:

```
In [161] menu.pop("French Fries")
```

```
Out [161] 2.99
```

```
In [162] menu
```

```
Out [162] {'Cheeseburger': 9.99, 'Soda': 1.99, 'Taco': 0.99}
```

Ключевое слово `in` проверяет наличие указанного ключа в словаре:

```
In [163] "Soda" in menu
```

```
Out [163] True
```

```
In [164] "Spaghetti" in menu
```

```
Out [164] False
```

Проверить присутствие некоторого значения в словаре можно с помощью метода `values`. Он возвращает объект, подобный списку, который содержит найденные значения. В сочетании с методом `values` можно использовать оператор `in`:

```
In [165] 1.99 in menu.values()
```

```
Out [165] True
```

```
In [166] 499.99 in menu.values()
```

```
Out [166] False
```

Метод `values` возвращает объект, похожий на список, но не являющийся ни списком, ни кортежем, ни словарем. Однако нам совсем не обязательно точно знать тип этого объекта. Нас интересует только возможность работать с ним. Оператор `in` проверяет присутствие указанного значения в объекте, а объект, возвращаемый методом `values`, знает, как обслужить оператор `in`.

Б.11.1. Итерации по словарям

Мы всегда должны помнить, что пары «ключ/значение» хранятся в словаре без какого-то определенного порядка. Если вам нужна структура данных, поддерживающая упорядоченное хранение элементов, то используйте список или кортеж. Если вам нужно связать пары объектов, используйте словарь.

Пусть мы не гарантируем определенную упорядоченность элементов словаря, мы тем не менее можем использовать цикл `for` для их обработки по одному за итерацию. Метод `items` словаря выдает в каждой итерации кортеж с двумя элементами. Этот кортеж содержит ключ и соответствующее ему значение. Для сохранения ключа и значения можно объявить несколько переменных после ключевого слова `for`. В следующем примере переменная `state` будет получать ключ словаря в каждой итерации, а переменная `capital` — соответствующее значение:

```
In [167] capitals = {
            "New York": "Albany",
            "Florida": "Tallahassee",
            "California": "Sacramento"
        }

        for state, capital in capitals.items():
            print("The capital of " + state + " is " + capital + ".")

        The capital of New York is Albany.
        The capital of Florida is Tallahassee.
        The capital of California is Sacramento.
```

В первой итерации Python возвращает кортеж ("New York", "Albany"). Во второй — кортеж ("Florida", "Tallahassee") и т. д.

Б.12. МНОЖЕСТВА

Объекты списков и словарей помогают решить задачи упорядочения и создания ассоциаций. Множества помогают удовлетворить еще одну общую потребность: уникальность. *Множество* — это неупорядоченный изменяемый набор уникальных элементов. Он не может хранить повторяющиеся элементы.

Множества определяются с помощью пары фигурных скобок, внутри которых через запятую перечисляются элементы. В следующем примере объявляется множество из шести чисел:

```
In [168] favorite_numbers = { 4, 8, 15, 16, 23, 42 }
```

Внимательные читатели могут заметить, что синтаксис объявления множеств идентичен синтаксису объявления словарей. Python различает эти два типа объектов по наличию или отсутствию пар «ключ/значение».

Поскольку Python интерпретирует пустую пару фигурных скобок как пустой словарь, единственный способ создать пустое множество — использовать встроенную функцию `set`:

```
In [169] set()
```

```
Out [169] set()
```

Вот несколько полезных методов множеств. Метод `add` добавляет новый элемент в множество:

```
In [170] favorite_numbers.add(100)
         favorite_numbers
```

```
Out [170] {4, 8, 15, 16, 23, 42, 100}
```

Python добавит элемент в множество, только если в нем еще нет этого элемента. В следующем примере делается попытка добавить 15 в `favorite_numbers`. Python обнаруживает, что 15 уже присутствует в множестве, и просто ничего не делает:

```
In [171] favorite_numbers.add(15)
         favorite_numbers
```

```
Out [171] {4, 8, 15, 16, 23, 42, 100}
```

Множества не поддерживают понятия упорядоченности. При попытке получить элемент множества по индексу Python сгенерирует исключение `TypeError`:

```
In [172] favorite_numbers[2]
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-17-e392cd51c821> in <module>
----> 1 favorite_numbers[2]
```

```
TypeError: 'set' object is not subscriptable
```

Python генерирует исключение `TypeError` при любой попытке применить операцию, недопустимую для объекта. Множества хранят элементы без определенного порядка, поэтому они не имеют индексов.

Кроме предотвращения дублирования элементов, множества идеально подходят для выявления сходств и различий между двумя коллекциями данных. Определим два набора строк:

```
In [173] candy_bars = { "Milky Way", "Snickers", "100 Grand" }
         sweet_things = { "Sour Patch Kids", "Reeses Pieces", "Snickers" }
```

Метод `intersection` возвращает новое множество с элементами, присутствующими в обоих указанных множествах. Ту же логику реализует оператор `&`. Как показывает следующий пример, множества `candy_bars` и `sweet_things` содержат только один общий элемент — строку `"Snickers"`:

```
In [174] candy_bars.intersection(sweet_things)
```

```
Out [174] {'Snickers'}
```

```
In [175] candy_bars & sweet_things
```

```
Out [175] {'Snickers'}
```

Метод `union` возвращает новое множество, включающее все элементы из двух указанных множеств. Ту же логику реализует оператор `|`. Обратите внимание, что повторяющиеся значения, такие как строка `"Snickers"` в этом примере, присутствуют в новом множестве только в одном экземпляре:

```
In [176] candy_bars.union(sweet_things)
```

```
Out [176] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Snickers', 'Sour
Patch Kids'}
```

```
In [177] candy_bars | sweet_things
```

```
Out [177] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Snickers', 'Sour
Patch Kids'}
```

Метод `difference` возвращает новое множество с элементами, присутствующими в множестве, для которого вызван этот метод, но отсутствующими в множестве, переданном в качестве аргумента. Ту же логику реализует оператор `-`. Как показывает следующий пример, строки `"100 Grand"` и `"Milky Way"` присутствуют в множестве `candy_bars` и отсутствуют в `sweet_things`:

```
In [178] candy_bars.difference(sweet_things)
```

```
Out [178] {'100 Grand', 'Milky Way'}
```

```
In [179] candy_bars - sweet_things
```

```
Out [179] {'100 Grand', 'Milky Way'}
```

Метод `symmetric_difference` возвращает новое множество с элементами, присутствующими в каком-то одном множестве, но не в обоих сразу. Ту же логику реализует оператор `^`:

```
In [180] candy_bars.symmetric_difference(sweet_things)
```

```
Out [180] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Sour Patch Kids'}
```

```
In [181] candy_bars ^ sweet_things
```

```
Out [181] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Sour Patch Kids'}
```

Это все, что я хотел рассказать! Вы довольно много узнали о Python: типы данных, функции, итерации и многое другое. Ничего страшного, если вы не запомнили всех деталей. Просто возвращайтесь к этому приложению всякий раз, когда нужно будет освежить в памяти основные механизмы Python. Материалы основных глав книги, описывающих работу с библиотекой `pandas`, содержат множество ссылок на идеи, изложенные в этом приложении.

Приложение В

Экспресс-курс по библиотеке NumPy

Библиотека с открытым исходным кодом NumPy (Numerical Python) используется библиотекой pandas и предоставляет мощный объект `ndarray` для хранения однородных (гомогенных) n -мерных массивов. Это довольно мощная конструкция, давайте разберем ее. *Массив* — это упорядоченный набор значений, подобный списку Python. *Однородный* (или гомогенный) означает, что массив может хранить значения только какого-то одного типа данных. «*N-мерный*» означает, что массив может иметь любое количество измерений. (Об измерениях мы поговорим в разделе В.1.) NumPy была разработана специалистом по обработке данных Трэвисом Олифантом (Travis Oliphant), который основал Anaconda, компанию, которая распространяет дистрибутив Python. А именно его мы использовали для создания нашей среды разработки.

С помощью NumPy можно генерировать наборы случайных данных любого размера и формы; на самом деле официальная документация pandas широко освещает это. Базовое знакомство с библиотекой NumPy здесь призвано помочь вам лучше понять механику работы pandas.

В.1. ИЗМЕРЕНИЯ

Под *измерениями* понимается количество точек отсчета, необходимых для извлечения одного значения из структуры данных. Рассмотрим коллекцию результатов измерения температуры воздуха в нескольких городах в определенный день (табл. В.1)¹.

¹ Температура в таблицах примера представлена в градусах по шкале Фаренгейта. — *Примеч. пер.*

Таблица В.1

Город	Температура
Нью-Йорк	38
Чикаго	36
Сан-Франциско	51
Майами	73

Если бы я попросил вас найти конкретную температуру в этом наборе данных, вам потребовалась бы только одна точка отсчета: название города (например, «Сан-Франциско») или его порядковый номер (например, «третий город в списке»). Таким образом, эта таблица представляет одномерный набор данных.

Сравните эту таблицу с набором данных температур для нескольких городов за несколько дней (табл. В.2).

Таблица В.2

Город	Понедельник	Вторник	Среда	Четверг	Пятница
Нью-Йорк	38	41	35	32	35
Чикаго	36	39	31	27	25
Сан-Франциско	51	52	50	49	53
Майами	73	74	72	71	74

Сколько точек отсчета потребуется сейчас, чтобы извлечь конкретное значение из этого набора данных? Ответ: 2. Нужно знать город и день недели (например, «Сан-Франциско в четверг») или номер строки и номер столбца (например, «строка 3 и столбец 4»). Ни город, ни день недели по отдельности не являются достаточными идентификаторами, потому что каждому из них соответствует несколько значений в наборе данных. Комбинация города и дня недели (или, что то же самое, строки и столбца) однозначно определяет единственное значение на их пересечении; соответственно, этот набор данных является двумерным.

Число измерений, идентифицирующих значение, не зависит от количества строк и столбцов в наборе данных. Таблица с 1 миллионом строк и 1 миллионом столбцов по-прежнему будет двумерной. Нам по-прежнему потребуется комбинация строки и столбца, чтобы извлечь значение.

Каждая дополнительная точка отсчета добавляет еще одно измерение. Мы могли бы собирать температуры в течение двух недель (табл. В.3 и В.4).

Названий города и дня недели больше недостаточно для извлечения одного значения. Теперь нужны три точки отсчета (номер недели, город и день), соответственно, этот набор данных можно классифицировать как трехмерный.

Таблица В.3. Неделя 1

Город	Понедельник	Вторник	Среда	Четверг	Пятница
Нью-Йорк	38	41	35	32	35
Чикаго	36	39	31	27	25
Сан-Франциско	51	52	50	49	53
Майами	73	74	72	71	74

Таблица В.4. Неделя 2

Город	Понедельник	Вторник	Среда	Четверг	Пятница
Нью-Йорк	40	42	38	36	28
Чикаго	32	28	25	31	25
Сан-Франциско	49	55	54	51	48
Майами	75	78	73	76	71

В.2. ОБЪЕКТ NDARRAY

Для начала создадим новый блокнот Jupyter и импортируем библиотеку NumPy, которой обычно назначается псевдоним `np`:

```
In [1] import numpy as np
```

NumPy способна генерировать как случайные, так и неслучайные данные. Начнем с простой задачи: создадим набор последовательных чисел из определенного диапазона.

В.2.1. Создание набора последовательных чисел с помощью метода `arange`

Функция `arange` возвращает одномерный объект `ndarray` с набором последовательных числовых значений. Когда `arange` вызывается с одним аргументом, NumPy использует в качестве нижней границы диапазона число `0`. А первый аргумент в этом случае задает верхнюю границу, число, на котором заканчивается диапазон. Значение верхней границы не включается в диапазон; NumPy дойдет до этого значения, но не включит его. При вызове с аргументом `3`, например, `arange` создаст `ndarray` со значениями `0`, `1` и `2`:

```
In [2] np.arange(3)
```

```
Out [2] array([0, 1, 2])
```

Если функции `arange` передать два аргумента, то они будут интерпретироваться как нижняя и верхняя границы диапазона. Значение нижней границы будет включено в сгенерированный набор, а значение верхней границы, как нам уже довелось видеть в предыдущих главах, — нет. Обратите внимание, что в следующем примере NumPy включает в набор 2, но не включает 6:

```
In [3] np.arange(2, 6)
```

```
Out [3] array([2, 3, 4, 5])
```

Первые два аргумента `arange` соответствуют именованным параметрам `start` и `stop`. Мы можем передать аргументы, явно указав имена параметров. Предыдущий и следующий примеры кода создают один и тот же массив:

```
In [4] np.arange(start = 2, stop = 6)
```

```
Out [4] array([2, 3, 4, 5])
```

Необязательный третий параметр `step` функции `arange` определяет интервал между соседними значениями, что помогает рассматривать эту концепцию с математической точки зрения: сначала в набор добавляется значение нижней границы, затем в него последовательно добавляются значения через указанный интервал, пока не будет достигнута верхняя граница. Пример ниже создает диапазон от 0 до 111 (не включая 111) с интервалом 10:

```
In [5] np.arange(start = 0, stop = 111, step = 10)
```

```
Out [5] array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110])
```

Сохраним последний массив в переменной `tens`:

```
In [6] tens = np.arange(start = 0, stop = 111, step = 10)
```

Теперь переменная `tens` ссылается на объект `ndarray`, содержащий 12 чисел.

В.2.2. Атрибуты объекта `ndarray`

Объект `ndarray` имеет свой набор атрибутов и методов. Напомню, что *атрибут* — это элемент данных, принадлежащих объекту. *Метод* — это команда, которую можно отправить объекту.

Атрибут `shape` возвращает кортеж, описывающий измерения массива. Длина кортежа `shape` равна количеству измерений `ndarray`. Как показывает следующий пример, `tens` — это одномерный массив с 12 значениями:

```
In [7] tens.shape
```

```
Out [7] (12,)
```

Количество измерений объекта `ndarray` можно также узнать с помощью атрибута `ndim`:

```
In [8] tens.ndim
```

```
Out [8] 1
```

Атрибут `size` возвращает количество элементов в массиве:

```
In [9] tens.size
```

```
Out [9] 12
```

Далее посмотрим, как можно манипулировать объектом `ndarray` в переменной `tens` с 12 элементами.

В.2.3. Метод `reshape`

В настоящее время наш массив `tens` с 12 элементами (если помните, целочисленные значения через 10) имеет только одно измерение. Мы можем получить любой его элемент с помощью всего одной точки отсчета — позиции в очередности:

```
In [10] tens
```

```
Out [10] array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110])
```

Нам может понадобиться преобразовать существующий одномерный массив в многомерный с другой формой. Предположим, что наши 12 значений представляют результаты измерений за четыре дня, по три каждый день. О таких данных проще рассуждать, если они организованы в форму 4×3 , а не 12×1 .

Метод `reshape` возвращает новый объект `ndarray` с формой, соответствующей аргументам метода. Например, массив `tens` преобразуется в новый двумерный массив с четырьмя строками и тремя столбцами:

```
In [11] tens.reshape(4, 3)
```

```
Out [11] array([[ 0, 10, 20],
                 [ 30, 40, 50],
                 [ 60, 70, 80],
                 [ 90, 100, 110]])
```

Количество аргументов, передаваемых в вызов `reshape`, должно соответствовать количеству измерений в новом `ndarray`:

```
In [12] tens.reshape(4, 3).ndim
```

```
Out [12] 2
```

Мы должны гарантировать, что произведение аргументов будет равно количеству элементов в исходном массиве. Значения 4 и 3 являются допустимыми аргументами, потому что их произведение равно 12, а `tens` содержит 12 значений. Другой допустимый пример — двумерный массив с двумя строками и шестью столбцами:

```
In [13] tens.reshape(2, 6)

Out [13] array([[ 5, 15, 25, 35, 45, 55],
                [ 65, 75, 85, 95, 105, 115]])
```

Если исходный массив нельзя преобразовать в запрошенную форму, то NumPy генерирует исключение `ValueError`. В следующем примере библиотека не смогла уместить 12 значений из `tens` в новый массив 2×5 :

```
In [14] tens.reshape(2, 5)

Out [14]
-----
ValueError                                Traceback (most recent call last)
<ipython-input-68-5b9588276555> in <module>
----> 1 tens.reshape(2, 5)

ValueError: cannot reshape array of size 12 into shape (2,5)
```

Может ли `ndarray` хранить больше двух измерений? Может. Передадим третий аргумент в вызов `reshape`, чтобы убедиться в этом. В следующем примере одномерный массив `tens` преобразуется в трехмерный массив $2 \times 3 \times 2$:

```
In [15] tens.reshape(2, 3, 2)

Out [15] array([[[ 5, 15],
                  [ 25, 35],
                  [ 45, 55]],
                [[ 65, 75],
                  [ 85, 95],
                  [105, 115]]])
```

Проверим значение атрибута `ndim` нового массива: уточним, действительно ли структура данных имеет три измерения:

```
In [16] tens.reshape(2, 3, 2).ndim

Out [16] 3
```

Мы также можем передать в вызов `reshape` аргумент `-1`, чтобы NumPy автоматически вычислила размер неизвестного измерения. В следующем примере передаются аргументы 2 и `-1`, а NumPy автоматически определяет, что новый двумерный массив должен иметь форму 2×6 :

```
In [17] tens.reshape(2, -1)

Out [17] array([[ 0, 10, 20, 30, 40, 50],
                [ 60, 70, 80, 90, 100, 110]])
```

А в примере ниже библиотека автоматически определяет, что новый объект `ndarray` должен иметь форму $2 \times 3 \times 2$:

```
In [18] tens.reshape(2, -1, 2)
```

```
Out [18] array([[ 0, 10],
                [ 20, 30],
                [ 40, 50]],
               [[ 60, 70],
                [ 80, 90],
                [100, 110]])
```

В вызов `reshape` можно передать неизвестный размер только для одного измерения.

Метод `reshape` возвращает новый объект `ndarray`. Исходный массив остается в неприкосновенности. То есть наш массив `tens` по-прежнему имеет форму 1×12 .

В.2.4. Функция `randint`

Функция `randint` генерирует одно или несколько случайных чисел из диапазона. При передаче одного аргумента он возвращает случайное целое число от 0 до указанного значения. Следующий пример возвращает случайное значение из диапазона от 0 до 5 (не включая его):

```
In [19] np.random.randint(5)
Out [19] 3
```

Функции `randint` можно передать два аргумента, чтобы объявить явно нижнюю границу, входящую в диапазон, и верхнюю границу, не входящую в диапазон. NumPy выберет число из описанного таким образом диапазона:

```
In [20] np.random.randint(1, 10)
```

```
Out [20] 9
```

А что, если нам понадобится сгенерировать массив случайных целых чисел? Для этого можно передать в вызов `randint` третий аргумент, указав желаемую форму массива. Чтобы создать одномерный массив, в качестве третьего аргумента можно передать либо одно целое число, либо список с одним элементом:

```
In [21] np.random.randint(1, 10, 3)
```

```
Out [21] array([4, 6, 3])
```

```
In [22] np.random.randint(1, 10, [3])
```

```
Out [22] array([9, 1, 6])
```

А вот для создания многомерного `ndarray` в третьем аргументе нужно передать список, описывающий количество значений в каждом измерении. Приведу пример, в котором создается двумерный массив 3×5 со значениями от 1 до 10 (не включая 10):

```
In [23] np.random.randint(1, 10, [3, 5])
```

```
Out [23] array([[2, 9, 8, 8, 7],
               [9, 8, 7, 3, 2],
               [4, 4, 5, 3, 9]])
```

Если нужно создать `ndarray` с большим количеством измерений, в списке можно указать любое желаемое количество значений. Например, список с тремя значениями создаст трехмерный массив.

В.2.5. Функция `randn`

Функция `randn` возвращает `ndarray` со случайными значениями из стандартного нормального распределения. Каждый следующий аргумент функции задает количество значений в соответствующем измерении. Если передать один аргумент, `randn` вернет одномерный `ndarray`. В следующем примере создается массив 1×3 (одна строка и три столбца):

```
In [24] np.random.randn(3)
```

```
Out [24] array([-1.04474993,  0.46965268, -0.74204863])
```

Если передать два аргумента, то `randn` вернет двумерный `ndarray` и т. д. Например, в коде ниже создается двумерный массив 2×4 :

```
In [25] np.random.randn(2, 4)
```

```
Out [25] array([[-0.35139565,  1.15677736,  1.90854535,  0.66070779],
               [-0.02940895, -0.86612595,  1.41188378, -1.20965709]])
```

А теперь создадим трехмерный массив $2 \times 4 \times 3$. Этот массив можно представить как два набора данных, каждый из которых состоит из четырех строк и трех столбцов:

```
In [26] np.random.randn(2, 4, 3)
```

```
Out [26] array([[[ 0.38281118,  0.54459183,  1.49719148],
                 [-0.03987083,  0.42543538,  0.11534431],
                 [-1.38462105,  1.54316814,  1.26342648],
                 [ 0.6256691 ,  0.51487132,  0.40268548]],
                [[-0.24774185, -0.64730832,  1.65089833],
                 [ 0.30635744,  0.21157744, -0.5644958 ],
                 [ 0.35393732,  1.80357335,  0.63604068],
                 [-1.5123853 ,  1.20420021,  0.22183476]])
```


Семейство функций `rand` дает уникальную возможность генерировать фиктивные числовые данные. К тому же можно создавать фиктивные данные различных типов и категорий, такие как имена, адреса или номера кредитных карт. Дополнительную информацию по этой теме см. в приложении Г.

В.3. ОБЪЕКТ NAN

Для представления отсутствующего или недопустимого значения библиотека NumPy использует специальный объект `nan`. Аббревиатура `nan` расшифровывается как *not a number* («не число»). Это универсальный термин для обозначения отсутствующих данных. Объект `nan` часто упоминается в книге, в частности, когда описывается импорт в структуры данных `pandas` наборов с отсутствующими значениями. На данный момент можно получить доступ к объекту `nan` напрямую как к атрибуту пакета `np`:

```
In [27] np.nan
```

```
Out [27] nan
```

Объект `nan` не равен никакому значению:

```
In [28] np.nan == 5
```

```
Out [28] False
```

Значение `nan` также не равно самому себе и другим `nan`. С точки зрения NumPy значения `nan` отсутствуют. Нельзя с уверенностью сказать, что два отсутствующих значения равны, поэтому предполагается, что они разные.

```
In [29] np.nan == np.nan
```

```
Out [29] False
```

Вот и все! Это были самые важные сведения о библиотеке NumPy, которую `pandas` использует «за кулисами», без визуального отображения.

Когда у вас появится свободная минутка, загляните в документацию по `pandas` (https://pandas.pydata.org/docs/user_guide/10min.html). Там вы найдете много примеров использования NumPy для генерации случайных данных.

Приложение Г

Генерирование фиктивных данных с помощью *Faker*

Faker — это библиотека для создания фиктивных данных. Она специализируется на создании списков имен, номеров телефонов, улиц, адресов электронной почты и т. п. Используя ее и библиотеку *NumPy*, способную генерировать случайные числовые данные, можно быстро создавать моделирующие наборы данных любого размера, формы и типа. Если вы хотите попрактиковаться в применении библиотеки *pandas*, но не можете найти подходящий набор данных, то *Faker* сможет предложить вам идеальное решение. В этом приложении мы рассмотрим все, что нужно знать, чтобы начать работу с этой библиотекой.

Г.1. УСТАНОВКА FAKER

Для начала установим библиотеку *Faker* в нашу среду *conda*. В приложении *Terminal* в *macOS* или *Anaconda Prompt* в *Windows* активируйте среду *conda*, которую вы настроили для этой книги. Когда я создавал окружение в приложении А, я назвал его `pandas_in_action`:

```
conda activate pandas_in_action
```

Если вы забыли, как называется ваша среда, то выполните команду `conda info --envs`, чтобы получить список имеющихся окружений.

После активации среды установите библиотеку *Faker* командой `conda install`:

```
conda install faker
```

Когда будет предложено подтвердить действие, введите Y, чтобы ответить «Да», и нажмите клавишу Enter. Anaconda загрузит и установит библиотеку. По завершении установки запустите Jupyter Notebook и создайте новый блокнот.

Г.2. НАЧАЛО РАБОТЫ С FAKER

Рассмотрим некоторые основные функции Faker, а затем объединим ее с библиотекой NumPy для создания набора данных `DataFrame` с 1000 строк. Сначала импортируем библиотеки `pandas` и `NumPy` и присвоим им соответствующие псевдонимы (`pd` и `np`). Также импортируем библиотеку `faker`:

```
In [1] import pandas as pd
      import numpy as np
      import faker
```

Пакет `faker` экспортирует класс `Faker` (обратите внимание на заглавную букву F в наименовании класса). Напомню, что класс — это макет объекта, шаблон структуры данных. `Series` и `DataFrame` — два примера классов из библиотеки `pandas`, а `Faker` — образец класса из библиотеки `Faker`.

Создадим экземпляр класса `Faker`, добавив после имени класса пару круглых скобок, и затем сохраним полученный объект `Faker` в переменной `fake`:

```
In [2] fake = faker.Faker()
```

Объект `Faker` имеет множество методов экземпляра, каждый из которых возвращает случайное значение из заданной категории. Метод экземпляра `name`, например, возвращает строку с полным именем человека:

```
In [3] fake.name()
```

```
Out [3] 'David Lee'
```

Учитывая, что `Faker` генерирует случайные данные, вы на своем компьютере, скорее всего, получите иной результат. Это совершенно нормально.

Чтобы получить мужские и женские полные имена, можно воспользоваться методами `name_male` и `name_female` соответственно:

```
In [4] fake.name_male()
```

```
Out [4] 'James Arnold'
```

```
In [5] fake.name_female()
```

```
Out [5] 'Brianna Hall'
```

492 Приложения

Чтобы получить только имя или только фамилию, используйте методы `first_name` и `last_name`:

```
In [6] fake.first_name()
```

```
Out [6] 'Kevin'
```

```
In [7] fake.last_name()
```

```
Out [7] 'Soto'
```

Есть также методы `first_name_male` и `first_name_female`, возвращающие только имена, мужские или женские:

```
In [8] fake.first_name_male()
```

```
Out [8] 'Brian'
```

```
In [9] fake.first_name_female()
```

```
Out [9] 'Susan'
```

Как видите, библиотека `Faker` предлагает простой, но мощный синтаксис. Вот еще один пример. Предположим, нам нужно сгенерировать несколько случайных адресов. Для этого можно использовать метод `address`, возвращающий строку с полным адресом: улицей, городом, штатом и почтовым индексом:

```
In [10] fake.address()
```

```
Out [10] '6162 Chase Corner\nEast Ronald, SC 68701'
```

Обратите внимание, что адрес полностью фиктивный; вы не найдете его на карте. `Faker` просто следует соглашениям о том, как должен выглядеть типичный адрес.

Заметьте также, что `Faker` отделяет название улицы от остальной части адреса символом перевода строки (`\n`). При желании можно передать сгенерированную строку в функцию `print` и вывести адрес в нескольких строках:

```
In [11] print(fake.address())
```

```
Out [11] 602 Jason Ways Apt. 358
         Hoganville, NV 37296
```

С помощью методов `street_address`, `city`, `state` и `postcode` можно сгенерировать отдельные элементы адреса:

```
In [12] fake.street_address()
```

```
Out [12] '58229 Heather Walk'
```

```
In [13] fake.city()
```

```
Out [13] 'North Kristinside'
```

```
In [14] fake.state()
```

```
Out [14] 'Oklahoma'
```

```
In [15] fake.postcode()
```

```
Out [15] '94631'
```

Вот еще один набор методов, с ним можно генерировать данные, связанные с бизнесом. Следующие методы возвращают случайные название компании, рекламный слоган, должность и URL:

```
In [16] fake.company()
```

```
Out [16] 'Parker, Harris and Sutton'
```

```
In [17] fake.catch_phrase()
```

```
Out [17] 'Switchable systematic task-force'
```

```
In [18] fake.job()
```

```
Out [18] 'Copywriter, advertising'394 APPENDIX D Generating fake data with Faker
```

```
In [19] fake.url()
```

```
Out [19] 'https://www.gutierrez.com/'
```

Faker также может генерировать случайные адреса электронной почты, номера телефонов и кредитных карт:

```
In [20] fake.email()
```

```
Out [20] 'sharon13@taylor.com'
```

```
In [21] fake.phone_number()
```

```
Out [21] '680.402.4787'
```

```
In [22] fake.credit_card_number()
```

```
Out [22] '4687538791240162'
```

На сайте Faker (<https://faker.readthedocs.io/en/master>) вы найдете исчерпывающее описание методов экземпляра объекта Faker. Библиотека группирует методы в категории, такие как адрес, автомобиль и банк. На рис. Г.1 показан пример страницы из документации Faker.

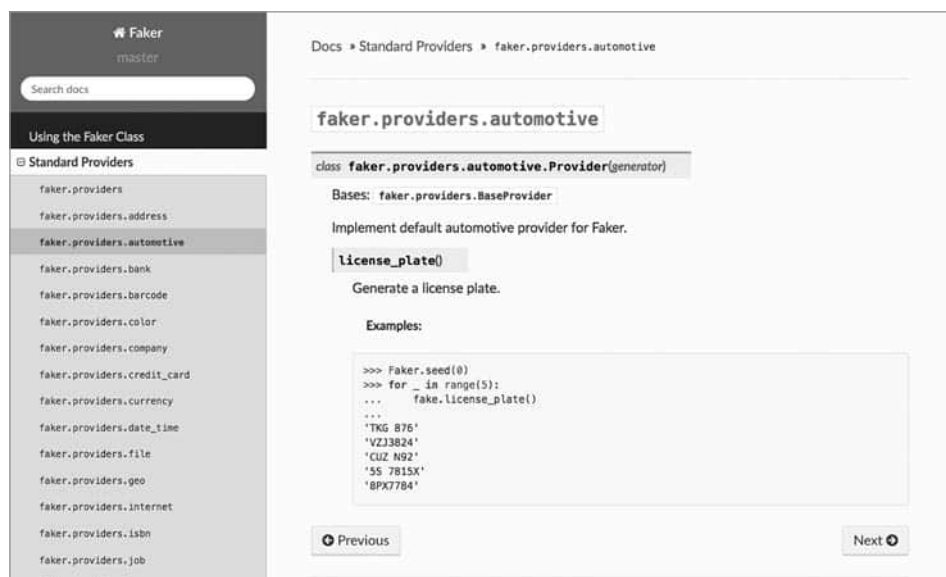


Рис. Г.1. Пример страницы из документации на официальном сайте Faker

Найдите время и познакомьтесь с доступными категориями Faker. Внесите небольшое разнообразие, чтобы сделать набор фиктивных данных для отладки вашего проекта намного более интригующим.

Г.3. ЗАПОЛНЕНИЕ НАБОРА ДАННЫХ DATAFRAME ФИКТИВНЫМИ ЗНАЧЕНИЯМИ

Теперь, узнав, как с помощью Faker генерировать фиктивные значения, воспользуемся этой библиотекой и создадим свой набор данных. Наша цель — создать `DataFrame` из 1000 строк с четырьмя столбцами: `Name`, `Company`, `Email` и `Salary`.

Вот как мы решим эту задачу: используем цикл `for`, выполняющий 1000 итераций, и в каждой итерации сгенерируем фиктивные имя, название компании и адрес электронной почты. Обратимся к `NumPy`, чтобы сгенерировать случайное число, представляющее величину зарплаты.

Для организации итераций можно использовать функцию `range` из стандартной библиотеки Python. Эта функция принимает целочисленный аргумент и возвращает итерируемую последовательность возрастающих чисел, начиная с 0 и заканчивая переданным в нее аргументом (но не включая последний). Следующий пример использует цикл `for` для перебора значений в диапазоне от 0 (включительно) до 5 (не включая его):

```
In [23] for i in range(5):  
        print(i)
```

```
Out [23] 0  
        1  
        2  
        3  
        4
```

Чтобы сгенерировать желаемый набор данных с 1000 строк, используем `range(1000)`.

Конструктор класса `DataFrame` принимает различные входные данные в своем параметре `data`, включая список словарей. Pandas отображает каждый ключ словаря в столбец `DataFrame`, а каждое значение — в значение для этого столбца. Вот как примерно должен выглядеть набор входных данных:

```
[  
    {  
        'Name': 'Ashley Anderson',  
        'Company': 'Johnson Group',  
        'Email': 'jessicabrooks@whitaker-crawford.biz',  
        'Salary': 62883  
    },  
    {  
        'Name': 'Katie Lee',  
        'Company': 'Ward-Aguirre',  
        'Email': 'kennethbowman@fletcher.com',  
        'Salary': 102971  
    }  
    # ... и еще 998 словарей  
]
```

Просматривая данные, сгенерированные библиотекой `Faker`, можно заметить некоторые логические несоответствия. Например, первого человека в наборе зовут `Ashley Anderson`, а соответствующий ему адрес электронной почты имеет вид `jessicabrooks@whitakercrawford.biz`. Этот казус связан со случайной природой данных в `Faker`. Мы не будем беспокоиться о подобных несостыковках. Однако, если у вас появится желание сделать набор данных более «жизненным», можете объединить вызовы методов из библиотеки `Faker` с обычным кодом на Python, генерирующим любые желаемые значения. Например, можно запросить у `Faker` имя ("`Morgan`") и фамилию ("`Robinson`"), а затем объединить эти две строки, чтобы сформировать более реалистичный адрес электронной почты ("`MorganRobinson@gmail.com`"):

```
In [24] first_name = fake.first_name_female()  
        last_name = fake.last_name()  
        email = first_name + last_name + "@gmail.com"  
        email
```

```
Out [24] 'MorganRobinson@gmail.com'
```

Но вернемся к нашей задаче. Воспользуемся генератором списков с функцией `range`, чтобы создать список из 1000 словарей. В каждом словаре объявим одни и те же четыре ключа: "Name", "Company", "Email" и "Salary". Для получения первых трех значений используем методы `name`, `company` и `email` объекта `Faker`. Напомню, что Python будет вызывать эти методы в каждой итерации, поэтому каждое новое значение будет отличаться от предыдущего. Чтобы получить значение для ключа "Salary", используем функцию `randint` из библиотеки `NumPy`. С ее помощью будем генерировать случайные целые числа в диапазоне от 50 000 до 200 000. Подробное описание этой функции вы найдете в приложении В.

```
In [25] data = [
    { "Name": fake.name(),
      "Company": fake.company(),
      "Email": fake.email(),
      "Salary": np.random.randint(50000, 200000)
    }
    for i in range(1000)
]
```

Итак, переменная `data` хранит список с 1000 словарей. Последний наш шаг — передать список словарей конструктору `DataFrame` из библиотеки `pandas`:

```
In [26] df = pd.DataFrame(data = data)
df
```

```
Out [26]
```

	Name	Company	Email	Salary
0	Deborah Lowe	Williams Group	ballbenjamin@gra...	147540
1	Jennifer Black	Johnson Inc	bryannash@carlso...	135992
2	Amy Reese	Mitchell, Hughes...	ajames@hotmail.com	101703
3	Danielle Moore	Porter-Stevens	logan76@ward.com	133189
4	Jennifer Wu	Goodwin Group	vray@boyd-lee.biz	57486
...
995	Joseph Stewart	Rangel, Garcia a...	sbrown@yahoo.com	123897
996	Deborah Curtis	Rodriguez, River...	smithedward@yaho...	51908
997	Melissa Simmons	Stevenson Ltd	frederick96@hous...	108791
998	Tracie Martinez	Morales-Moreno	caseycurry@lopez...	181615
999	Phillip Andrade	Anderson and Sons	anthony23@glover...	198586

```
1000 rows x 4 columns
```

Теперь у вас есть набор данных `DataFrame` с 1000 строк случайных данных, на котором можно попрактиковаться. Обязательно загляните в документацию для `Faker` и `NumPy`, чтобы узнать, какие другие типы случайных данных они могут генерировать.

Приложение Д

Регулярные выражения

Регулярное выражение (часто сокращенно RegEx) — это шаблон, описывающий структуру искомого текста. Он определяет логическую последовательность символов, которую компьютер должен отыскать в строке.

Вот простой пример. Почти наверняка вам приходилось использовать функцию поиска в своем веб-браузере. Чтобы начать поиск, в большинстве веб-браузеров можно нажать Ctrl+F (в Windows) или Command+F (в macOS). Браузер откроет диалог, в котором вводится искомая последовательность символов. Затем браузер отыщет эти символы на веб-странице. На рис. Д.1 показан пример окна браузера после успешного завершения поиска строки *romance* в содержимом страницы.



Рис. Д.1. Поиск строки *romance* в Google Chrome

Функция поиска в Chrome — это простой пример использования регулярных выражений. Этот инструмент имеет свои ограничения. Например, мы можем искать символы только в том порядке, в котором они появляются. Мы можем отыскать последовательность символов `cat`, но мы не можем объявить такое условие, как буква `c`, или `a`, или `t`. Так вот регулярные выражения делают возможным и такой динамический, условный поиск.

Регулярное выражение описывает, как выглядит искомое содержимое во фрагменте текста. Мы можем искать такие символы, как буквы, цифры или пробелы, а также использовать специальные символы для объявления условий. Вот, к примеру, что нам может понадобиться найти:

- любые две цифры подряд;
- последовательность из трех или более буквенных символов, за которыми следует пробел;
- символ `s`, но только в начале слова.

В этом приложении мы рассмотрим применение регулярных выражений в Python, а затем используем наши знания для поиска в наборе данных. Регулярным выражениям посвящены целые учебники и учебные курсы в колледжах, мы лишь вскользь коснемся этой сложной области. Начать использовать регулярные выражения легко, сложно овладеть ими в совершенстве.

Д.1. ВВЕДЕНИЕ В МОДУЛЬ RE

Для начала создадим новый блокнот Jupyter Notebook. Импортируем `pandas` и специальный модуль `re`. Модуль `re` (regular expressions — «регулярные выражения») является частью стандартной библиотеки Python:

```
In [1] import re
      import pandas as pd
```

В модуле `re` имеется функция `search`, которая ищет подстроку в строке. Функция принимает два аргумента: искомую последовательность и строку, в которой эту последовательность нужно найти. Код следующего примера ищет строку `"flower"` в строке `"field of flowers"`:

```
In [2] re.search("flower", "field of flowers")
```

```
Out [2] <re.Match object; span=(9, 15), match='flower'>
```

Функция `search` возвращает объект `Match`, если находит указанную последовательность символов в целевой строке. Объект `Match` хранит информацию о том, какая часть содержимого соответствует шаблону поиска и где она находится

в целевой строке. Предыдущий вывод отработки кода сообщает, что искомая последовательность `flower` найдена в целевой строке и находится в позициях с 9-й по 15-ю. Первый индекс соответствует первому символу найденной последовательности, а второй — первому символу, находящемуся за последним символом найденной последовательности. Если пронумеровать символы в строке `"field of flowers"`, то мы увидим, что в позиции с индексом 9 находится строчная буква `f` в слове `flowers`, а в позиции с индексом 15 — символ `s` в слове `flowers`.

Функция `search` возвращает `None`, если по заданному шаблону ничего не найдено. По умолчанию Jupyter Notebook ничего не выводит для значения `None`. Но мы можем передать вызов `search` в аргумент функции `print`, чтобы заставить Jupyter вывести это значение:

```
In [3] print(re.search("flower", "Barney the Dinosaur"))
```

```
Out [3] None
```

Функция `search` возвращает только первое совпадение, найденное в целевой строке. Найти все совпадения можно с помощью функции `findall`. Эта функция принимает те же два аргумента — искомую последовательность и целевую строку — и возвращает список строк, соответствующих искомой последовательности. В следующем примере `findall` находит два совпадения с шаблоном поиска `"flower"` в `"Picking flowers in the flower field"`:

```
In [4] re.findall("flower", "Picking flowers in the flower field")
```

```
Out [4] ['flower', 'flower']
```

Обратите внимание, что поиск выполняется с учетом регистра символов.

Д.2. МЕТАСИМВОЛЫ

Теперь объявим более сложный шаблон поиска, добавив в него элементы регулярных выражений. Для начала присвоим длинную строку переменной `sentence`. В следующем примере строка разбита на несколько строк для удобочитаемости, но вы можете ввести ее в одну строку:

```
In [5] sentence = "I went to the store and bought " \
                  "5 apples, 4 oranges, and 15 plums."
        sentence
```

```
Out [5] 'I went to the store and bought 5 apples, 4 oranges, and 15 plums.'
```

Внутри регулярного выражения можно использовать *метасимволы* — специальные символы, используемые для построения шаблонов поиска. Метасимвол `\d`, например, соответствует любой цифре. Допустим, мы хотим идентифицировать

все цифры в нашей строке "sentence". В следующем примере вызывается функция `findall` с регулярным выражением `"\d"` в качестве шаблона поиска:

```
In [6] re.findall("\d", sentence)
```

```
Out [6] ['5', '4', '1', '5']
```

Функция возвращает список из четырех цифр, встреченных в "sentence", в том порядке, в каком они появляются:

- 5 в 5 apples;
- 4 в 4 oranges;
- 1 в 15 plums;
- 5 в 15 plums.

Вот вы и познакомились с первым метасимволом! Используя простой символ `\d`, мы создали шаблон, который соответствует любой цифре в целевой строке.

Прежде чем двигаться дальше, хочу отметить два момента.

- Когда список содержит много элементов, Jupyter Notebook предпочитает выводить каждый элемент на отдельной строке. Такой стилистический подход облегчает чтение результатов, но требует много места. Чтобы заставить Jupyter выводить списки в обычном виде, разрывая строку только после вывода определенного количества символов, мы с этого момента будем передавать вызов функции `findall` в качестве аргумента во встроенную функцию `print`.
- Аргументы с регулярными выражениями будем передавать в функцию `findall` в виде неформатированных (raw) строк. Такие строки Python интерпретирует буквально. Это поможет предотвратить конфликты между регулярными выражениями и экранированными последовательностями. Рассмотрим последовательность символов `\b`. Она имеет символическое функциональное значение в простой строке и совсем другое значение в регулярном выражении. Используя неформатированные строки, мы сообщаем интерпретатору Python, что он должен рассматривать `\b` буквально, как пару символов — обратный слеш, за которым следует символ `b`, — а не функционал. Это гарантирует, что Python будет правильно интерпретировать метасимволы в регулярных выражениях.

Неформатированные строки объявляются добавлением символа `r` перед двойными кавычками. Перепишем предыдущий пример с вызовом функции `print` и неформатированной строкой:

```
In [7] print(re.findall(r"\d", sentence))
```

```
Out [7] ['5', '4', '1', '5']
```

Чтобы объявить в регулярном выражении обратное условие, достаточно изменить регистр метасимвола. Например, если `\d` означает «совпадение с любым цифровым символом», то `\D` означает «совпадение с любым нецифровым символом». К нецифровым символам относятся буквы, пробел, запятая и другие знаки. Ниже продемонстрировано использование метасимвола `\D` для идентификации всех нецифровых символов в `sentence`:

```
In [8] print(re.findall(r"\D", sentence))
```

```
Out [8] ['I', ' ', 'w', 'e', 'n', 't', ' ', 't', 'o', ' ', 't', 'h', 'e', ' ',
        's', 't', 'o', 'r', 'e', ' ', 'a', 'n', 'd', ' ', 'b', 'o',
        'u', 'g', 'h', 't', ' ', 'a', 'p', 'p', 'l', 'e', 's', ' ',
        ' ', 'o', 'r', ' ', 'a', 'n', 'g', 'e', 's', ' ', ' ', 'a', 'n',
        'd', ' ', ' ', 'p', 'l', 'u', 'm', 's', '.']
```

Теперь, познакоившись с основами регулярных выражений, рассмотрим дополнительные метасимволы и попробуем построить более сложные шаблоны поиска. Вот еще один пример. Метасимвол `\w` соответствует любому символу слова, к которым относятся буквы, цифры и символ подчеркивания:

```
In [9] print(re.findall(r"\w", sentence))
```

```
Out [9] ['I', 'w', 'e', 'n', 't', 't', 'o', 't', 'h', 'e', 's', 't', 'o',
        'r', 'e', 'a', 'n', 'd', 'b', 'o', 'u', 'g', 'h', 't', '5', 'a',
        'p', 'p', 'l', 'e', 's', '4', 'o', 'r', 'a', 'n', 'g', 'e', 's',
        'a', 'n', 'd', '1', '5', 'p', 'l', 'u', 'm', 's']
```

Обратный ему метасимвол `\W` соответствует любому символу, не являющемуся символом слова. К символам, не являющимся символами слов, относятся пробел, запятая и точка:

```
In [10] print(re.findall(r"\W", sentence))
```

```
Out [10] [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
        ' ', ' ', ' ', '.']
```

Метасимвол `\s` соответствует любому пробельному символу:

```
In [11] print(re.findall(r"\s", sentence))
```

```
Out [11] [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
        ' ', ' ', ' ', '.']
```

Обратный ему метасимвол `\S` соответствует любому непробельному символу:

```
In [12] print(re.findall(r"\S", sentence))
```

```
Out [12] ['I', 'w', 'e', 'n', 't', 't', 'o', 't', 'h', 'e', 's', 't', 'o',
        'r', 'e', 'a', 'n', 'd', 'b', 'o', 'u', 'g', 'h', 't', '5', 'a',
        'p', 'p', 'l', 'e', 's', ' ', '4', 'o', 'r', 'a', 'n', 'g', 'e',
        's', ' ', ' ', 'a', 'n', 'd', '1', '5', 'p', 'l', 'u', 'm', 's', '.']
```

Чтобы найти конкретный символ, его нужно объявить в шаблоне буквально. В следующем примере выполняется поиск всех вхождений буквы `t`. Это тот же синтаксис, который мы использовали в первых примерах в этом приложении:

```
In [13] print(re.findall(r"t", sentence))
```

```
Out [13] ['t', 't', 't', 't', 't']
```

Чтобы найти последовательность символов, добавьте их по порядку в шаблон. В следующем примере выполняется поиск букв `to` в строке `"sentence"`. Функция `findall` находит два совпадения (в слове `to` и в слове `store`):

```
In [14] print(re.findall(r"to", sentence))
```

```
Out [14] ['to', 'to']
```

Метасимвол `\b` объявляет границу слова. *Граница слова* указывает, где должен находиться искомый символ относительно пробела. В следующем примере выполняется поиск по шаблону `"\bt"`. Его логика интерпретируется так: «любой символ `t` после границы слова» или, что то же самое, «любой символ `t` после пробела». Этот шаблон соответствует символам `t` в `to` и `the`:

```
In [15] print(re.findall(r"\bt", sentence))
```

```
Out [15] ['t', 't']
```

Теперь поменяем символы местами. Шаблон `"t\b"` соответствует «любому символу `t` перед границей слова» или, что то же самое, «любому символу `t` перед пробелом». Для этого шаблона `findall` находит другие символы `t` — в конце слов `went` и `bought`:

```
In [16] print(re.findall(r"t\b", sentence))
```

```
Out [16] ['t', 't']
```

Обратный метасимвол `\B` объявляет границу не слова. Например, `"\Bt"` интерпретируется как «любой символ `t`, не следующий за границей слова» или, что то же самое, «любой символ `t`, не следующий за пробелом»:

```
In [17] print(re.findall(r"\Bt", sentence))
```

```
Out [17] ['t', 't', 't']
```

Предыдущий пример обнаружил символы `t` в словах `went`, `store` и `bought`. Python игнорировал символы `t` в словах `to` и `the`, потому что они следуют за границей слова.

Д.3. РАСШИРЕННЫЕ ШАБЛОНЫ ПОИСКА

Выше я отметил, что *метасимвол* — это символ, описывающий искомую последовательность в регулярном выражении. В разделе Д.2 вы познакомились с метасимволами `\d`, `\w`, `\s` и `\b`, представляющими цифры, символы слов, пробельные символы и границы слов. Теперь я представлю несколько новых метасимволов, а затем мы объединим их в сложный поисковый запрос.

Метасимвол точки (`.`) соответствует любому символу:

```
In [18] soda = "coca cola."
      soda
```

```
Out [18] 'coca cola.'
```

```
In [19] print(re.findall(r".", soda))
```

```
Out [19] ['c', 'o', 'c', 'a', ' ', 'c', 'o', 'l', 'a', '.']
```

На первый взгляд этот метасимвол не выглядит особенно полезным, но в сочетании с другими символами он творит чудеса. Например, регулярное выражение `"с."` соответствует символу `с`, за которым следует любой символ. Таких совпадений в нашей строке три:

```
In [20] print(re.findall(r"с.", soda))
```

```
Out [20] ['co', 'ca', 'co']
```

А если понадобится символ точки в строке? В этом случае точку в регулярном выражении нужно экранировать обратным слешем. Шаблон `"\."` в примере ниже соответствует точке в конце строки `"soda"`:

```
In [21] print(re.findall(r"\.", soda))
```

```
Out [21] ['.']
```

Выше мы видели, что в регулярном выражении можно указывать точные искомые последовательности символов. В следующем примере мы ищем точную последовательность `co`:

```
In [22] print(re.findall(r"co", soda))
```

```
Out [22] ['co', 'co']
```

А как быть, если понадобится отыскать либо символ `с`, либо символ `о`? Для этого можно заключить символы в квадратные скобки. Совпадения будут включать любое вхождение `с` или `о` в целевой строке:

504 Приложения

```
In [23] print(re.findall(r"[co]", soda))
```

```
Out [23] ['c', 'o', 'c', 'c', 'o']
```

Порядок символов в квадратных скобках не влияет на результат:

```
In [24] print(re.findall(r"[oc]", soda))
```

```
Out [24] ['c', 'o', 'c', 'c', 'o']
```

Допустим, нам потребовалось отыскать любые символы, расположенные между `c` и `l` в алфавите. Для этого можно было бы перечислить все алфавитные символы в квадратных скобках:

```
In [25] print(re.findall(r"[cdefghijkl]", soda))
```

```
Out [25] ['c', 'c', 'c', 'l']
```

Однако есть более удачное решение — использовать символ дефиса (`-`) для объявления диапазона символов. Следующий пример дает тот же результат, что и предыдущий:

```
In [26] print(re.findall(r"[c-l]", soda))
```

```
Out [26] ['c', 'c', 'c', 'l']
```

Теперь посмотрим, как отыскать несколько вхождений символа, идущих подряд. Рассмотрим строку `"bookkeeper"`:

```
In [27] word = "bookkeeper"  
        word
```

```
Out [27] 'bookkeeper'
```

Чтобы найти точно два символа `e`, можно явно указать их в искомой последовательности:

```
In [28] print(re.findall(r"ee", word))
```

```
Out [28] ['ee']
```

Совпадение с несколькими вхождениями, идущими подряд, можно также описать в шаблоне парой фигурных скобок, объявив между ними количество совпадений. Следующий пример отыскивает два символа `e` в слове `bookkeeper`:

```
In [29] print(re.findall(r"e{2}", word))
```

```
Out [29] ['ee']
```

Если попробовать поискать три символа `e`, идущих подряд, с помощью шаблона `"e{3}"`, то `findall` вернет пустой список, потому что в `bookkeeper` нет последовательностей с тремя символами `e`:


```
In [30] print(re.findall(r"e{3}", word))
```

```
Out [30] []
```

В фигурных скобках можно перечислить через запятую два числа. В таком случае первое число будет определять минимальное количество вхождений, а второе — максимальное. В следующем примере выполняется поиск от одного до трех вхождений символа `e` в строке. Первое совпадение — это последовательность символов `ee` в `bookkeeper`, а второе совпадение — последняя буква `e` в `bookkeeper`:

```
In [31] print(re.findall(r"e{1,3}", word))
```

```
Out [31] ['ee', 'e']
```

Рассмотрим этот пример подробнее. Шаблон соответствует последовательности от одного до трех символов `e`. Когда `findall` находит совпадение, она продолжает просматривать строку, пока шаблон поиска не перестанет совпадать с символами строки. Регулярное выражение сначала просмотрит символы `bookk` по отдельности. Ни один из них не соответствует шаблону, поэтому поиск продолжается дальше. Затем будет найдено совпадение шаблона с первой буквой `e`. Это совпадение пока не может быть отмечено как окончательное, потому что следующий символ тоже может быть буквой `e`, поэтому регулярное выражение проверяет следующий символ. Этот символ действительно является еще одной буквой `e`, соответствующей критериям поиска. Затем обнаруживается символ `p`, который не соответствует шаблону, и регулярное выражение объявляет найденное совпадение как `ee`, а не как два отдельных символа `e`. Та же логика повторяется для буквы `e` ближе к концу строки.

Мы уже довольно далеко продвинулись вперед в освоении регулярных выражений, но все предыдущие примеры были в основном теоретическими. А как можно использовать регулярные выражения при работе с наборами данных реального мира?

Представьте, что мы открыли горячую линию поддержки клиентов и храним расшифровки телефонных звонков. У нас может быть такое сообщение:

```
In [32] transcription = "I can be reached at 555-123-4567. "\
                        "Look forward to talking to you soon."
transcription
```

```
Out [32] 'I can be reached at 555-123-4567. Look forward to talking to you
soon.'
```

Допустим, нам нужно извлечь номер телефона из каждого такого сообщения. Сообщения могут быть самые разные, однако очевидно, что все телефонные номера имеют некоторый общий шаблон, опишем его ниже.

1. Три цифры.
2. Дефис.
3. Три цифры.
4. Дефис.
5. Четыре цифры.

Регулярные выражения имеют одну замечательную особенность — они позволяют отыскать совпадение с этим шаблоном независимо от содержимого строки. В следующем примере объявим самое сложное регулярное выражение в сравнении с предыдущими. Мы просто объединили в нем метасимволы и символы, чтобы реализовать описанный выше шаблон.

1. `\d{3}` соответствует ровно трем цифрам.
2. `-` соответствует дефису.
3. `\d{3}` соответствует ровно трем цифрам.
4. `-` соответствует дефису.
5. `\d{4}` соответствует ровно четырем цифрам.

```
In [33] print(re.findall(r"\d{3}-\d{3}-\d{4}", transcription))
```

```
Out [33] ['555-123-4567']
```

Вуаля!

Есть еще один удобный метасимвол — `+`, который добавляет к символу или метасимволу, следующему перед ним, условие «один или несколько». `\d+`, например, ищет одну или несколько цифр подряд. Мы можем упростить предыдущий пример, используя метасимвол `+`. Следующее регулярное выражение реализует другой шаблон поиска, но возвращает тот же результат.

1. Одна или несколько цифр, идущих подряд.
2. Дефис.
3. Одна или несколько цифр, идущих подряд.
4. Дефис.
5. Одна или несколько цифр, идущих подряд.

```
In [34] print(re.findall(r"\d+ - \d+ - \d+", transcription))
```

```
Out [34] ['555-123-4567']
```

Вот так с помощью одной строки кода мы теперь можем извлечь телефонный номер из самых разных сообщений, и это здорово.

Д.4. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ И PANDAS

В главе 6 представлен объект `StringMethods` для управления последовательностями `Series` строк. Объект доступен через атрибут `str`, и многие его методы способны принимать регулярные выражения в качестве аргументов, что значительно расширяет возможности методов. Давайте попрактикуемся в применении регулярных выражений к реальному набору данных.

Набор данных `ice_cream.csv` — это коллекция видов мороженого четырех популярных брендов (Ben & Jerry's, Haagen-Dazs, Breyers и Talenti). Каждая строка в наборе данных включает название бренда, вид и описание:

```
In [35] ice_cream = pd.read_csv("ice_cream.csv")
        ice_cream.head()
```

```
Out [35]
```

	Brand	Flavor	Description
0	Ben and Jerry's	Salted Caramel Core	Sweet Cream Ice Cream with Blon...
1	Ben and Jerry's	Netflix & Chilll'd™	Peanut Butter Ice Cream with Sw...
2	Ben and Jerry's	Chip Happens	A Cold Mess of Chocolate Ice Cr...
3	Ben and Jerry's	Cannoli	Mascarpone Ice Cream with Fudge...
4	Ben and Jerry's	Gimme S'more!™	Toasted Marshmallow Ice Cream w...

ПРИМЕЧАНИЕ

Набор данных `ice_cream` — это модифицированная версия набора данных, доступного на сайте Kaggle (<https://www.kaggle.com/tysonpo/ice-cream-dataset>). В данных здесь есть опечатки и несоответствия; я сохранил их, чтобы вы могли видеть, какие несоответствия могут присутствовать в реальных данных. Я предлагаю вам подумать, как можно оптимизировать и почистить эти данные с помощью методов, описанных в главе 6.

Допустим, мне стало любопытно, сколько разных шоколадных вкусов можно найти в этом наборе. Наша задача — найти все слова, которые следуют сразу за строкой `"Chocolate"` в столбце `Description`. Для этого можно использовать метод `str.extract` объектов `Series`. Он принимает регулярное выражение и возвращает `DataFrame` с найденными совпадениями.

Составим наше регулярное выражение. Начнем с границы слова (`\b`). Затем опишем буквальное слово `"Chocolate"`. Далее добавим пробельный символ (`\s`). Наконец, опишем совпадение с одним или несколькими символами слова в строке (`\w+`), чтобы захватить все буквенно-цифровые символы, предшествующие пробелу или точке. Используя такой подход, получаем выражение `"\bChocolate\s\w+"`.

По техническим причинам необходимо заключить регулярное выражение в круглые скобки перед передачей в качестве аргумента в вызов метода `str.extract`.

Метод поддерживает расширенный синтаксис поиска по нескольким регулярным выражениям, а круглые скобки ограничивают его одним:

```
In [36] ice_cream["Description"].str.extract(r"(\bChocolate\s\w+)").head()
```

```
Out [36]
-----
0      NaN
1      NaN
2  Chocolate Ice
3      NaN
4  Chocolate Cookie
```

Пока все идет нормально. Наша последовательность **Series** включает такие совпадения, как **Chocolate Ice** в индексной позиции 2 и **Chocolate Cookie** в индексной позиции 4; она также включает значения **NaN** для строк, в которых не было найдено совпадений с шаблоном. Вызовем метод **dropna**, чтобы удалить строки с отсутствующими значениями:

```
In [37] (
    ice_cream["Description"]
    .str.extract(r"(\bChocolate\s\w+)")
    .dropna()
    .head()
)
```

```
Out [37]
-----
2    Chocolate Ice
4    Chocolate Cookie
8    Chocolate Ice
9    Chocolate Ice
13   Chocolate Cookie
```

Так мы еще на шаг приблизились к решению.

Теперь преобразуем **DataFrame** в **Series**. По умолчанию метод **str.extract** возвращает **DataFrame**, реализуя поддержку поиска по нескольким шаблонам. Мы же можем использовать метод **squeeze**, чтобы превратить **DataFrame** с одним столбцом в последовательность **Series**. Возможно, вы помните, как мы применяли параметр **squeeze** при вызове функции импорта **read_csv**; метод **squeeze** дает тот же результат:

```
In [38] (
    ice_cream["Description"]
    .str.extract(r"(\bChocolate\s\w+)")
    .dropna()
    .squeeze()
)
```

```

        .squeeze()
        .head()
    )

```

```

Out [38] 2      Chocolate Ice
         4      Chocolate Cookie
         8      Chocolate Ice
         9      Chocolate Ice
        13      Chocolate Cookie
        Name: Chocolate, dtype: object

```

Наша цепочка вызовов методов становится довольно длинной, поэтому присвоим текущую последовательность `Series` переменной `Chocolate_flavors`:

```

In [39] chocolate_flavors = (
        ice_cream["Description"]
        .str.extract(r"(\bChocolate\s\w+)")
        .dropna()
        .squeeze()
    )

```

Конечная цель, напомним, — определить, какие ингредиенты следуют за словом `Chocolate`. Вызовем метод `str.split`, чтобы разбить каждую строку по пробелам. Однако вместо строки с одним пробелом мы передадим аргумент с регулярным выражением. Напомним, что метасимвол `\s` соответствует одному пробельному символу:

```

In [40] chocolate_flavors.str.split(r"\s").head()

```

```

Out [40] 2      [Chocolate, Ice]
         4      [Chocolate, Cookie]
         8      [Chocolate, Ice]
         9      [Chocolate, Ice]
        13      [Chocolate, Cookie]
        Name: 0, dtype: object

```

Метод `str.get` извлекает значение из соответствующей позиции в каждом списке в последовательности `Series`. В следующем примере мы извлекаем второй элемент (индекс 1) из каждого списка, или, что то же самое, слово, следующее за `Chocolate` в исходной строке:

```

In [41] chocolate_flavors.str.split(r"\s").str.get(1).head()

```

```

Out [41] 2      Ice
         4      Cookie
         8      Ice
         9      Ice
        13      Cookie
        Name: Chocolate, dtype: object

```

Ради любопытства вызовем метод `value_counts`, чтобы увидеть наиболее часто встречающиеся слова, следующие за словом `Chocolate` во всех видах мороженого. Неудивительно, что `Ice` является победителем по частоте встречаемости. За ним с большим отставанием следует слово `Cookie`:

```
In [42] chocolate_flavors.str.split(r"\s").str.get(1).value_counts()
```

```
Out [42] Ice          11
         Cookie       4
         Chip         3
         Cookies      2
         Sandwich     2
         Malt         1
         Mint         1
         Name: Chocolate, dtype: int64
```

Регулярные выражения предлагают богатый возможностями способ поиска в тексте по шаблонам. Я надеюсь, что вы достаточно хорошо поняли преимущества регулярных выражений и получили представление о том, как применять их при использовании различных методов в `pandas`.

Борис Пасхавер
Pandas в действии

Перевели с английского Л. Киселева, И. Пальти

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Пителимов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Куликова</i>
Корректоры	<i>Е. Павлович, Н. Терех</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2022. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 18.10.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 700. Заказ 0000.